HAL
open science

Optimization of Image B-spline Interpolation for GPU Architectures

Thibaud Briand, Axel Davy

To cite this version:

Thibaud Briand, Axel Davy. Optimization of Image B-spline Interpolation for GPU Architectures. Image Processing On Line, 2019, 9, pp.183-204. 10.5201/ipol.2019.257 . hal-02262502

HAL Id: hal-02262502

https://hal.science/hal-02262502v1

Submitted on 2 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

2015/06/16 v0.5.1 IPOL article class

Published in Image Processing On Line on 2019–08–02.
Submitted on 2019–04–11, accepted on 2019–07–22.
ISSN 2105–1232 © 2019 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2019.257

Optimization of Image B-spline Interpolation for GPU Architectures

Thibaud Briand[1,2], Axel Davy[2]

[1] Université Paris-Est, LIGM (UMR CNRS 8049), ENPC, F-77455 Marne-la-Vallée, France
(thibaud.briand@enpc.fr)
[2] Université Paris-Saclay, CMLA (UMR CNRS 8536), France
(axel.davy@ens-cachan.fr)

Abstract

Interpolation is a vital piece of many image processing pipelines. In this document we present how to optimize the implementation of image B-spline interpolation for GPU architectures. The implementation is based on the one proposed by Briand and Monasse in 2018 and works for orders up to 11. The two main optimizations consist in: (1) transposing the B-spline coefficients before the prefiltering of the rows and (2) dividing columns into subregions in order to use more threads. We assess the impact of the floating point precision and of using high B-spline orders.

Source Code

The OpenCL implementation of the code that we provide is the one which has been peer reviewed and accepted by IPOL. The source code, the code documentation, and the online demo are accessible at the IPOL web page of the article[1]. Usage instructions are included in the `README.txt` file of the archive.

Keywords: interpolation; splines; linear filtering; GPU; optimization

1 Introduction

Interpolation consists in constructing new data points within the range of a discrete set of known data points. In image processing it is commonly expressed as the problem of recovering from a discrete image $\underline{u}$ the underlying continuous signal $u : \mathbb{R}^2 \to \mathbb{R}$ verifying the interpolation condition

$$\forall (k,l), \quad u(k,l) = \underline{u}_{k,l}. \tag{1}$$

A continuous image representation is handy when one wishes to implement numerically an operator that is initially defined in the continuous domain. In particular, applying a geometric transformation to an image (like for example a rotation) requires an interpolation method.

[1] https://doi.org/10.5201/ipol.2019.257

About B-spline interpolation. The spline representation has been widely used since the 1990s. A spline of degree n is a continuous piece-wise polynomial function of degree n of a real variable with derivatives up to order $n-1$. This representation has the advantage of being equally justifiable on a theoretical and practical basis [15]. It can model the physical process of drawing a smooth curve and it is well adapted to signal processing thanks to its optimality properties. The B-spline representation [12] is particularly handy, as the continuous underlying signal is expressed as the result of a convolution between the B-spline kernel, that is compactly supported, and the discrete parameters of the representation, namely the B-spline coefficients. One of the strongest arguments in favor of the B-spline interpolation is that it approaches the Shannon-Whittaker interpolation as the order increases [1].

The determination of the B-spline coefficients is performed in a prefiltering step that, in general, can be done by solving a band diagonal system of linear equations [7, 9]. For uniformly spaced data points, which is most of the time the case in signal processing, Unser et al. proposed in [16] an efficient and stable algorithm based on linear filtering that works for any order. More details, in particular regarding the determination of the interpolator parameters, were provided by the authors in later publications [17, 18]. All the information, theoretical and practical, required to perform efficiently B-spline interpolation for any order and any boundary extension can be found in [4]. Two slightly different prefiltering algorithms, whose precision is proven to be controlled thanks to a rigorous boundary handling, were proposed.

Choice of the interpolation method and GPU. In practice, the choice of the interpolation method, among all the possibilities [6, 14], depends on the context. A compromise is made between interpolation quality and computational efficiency. Although B-spline interpolation (of high-order) provides high-quality results for images, faster but lower quality methods such as bilinear or bicubic interpolations are usually preferred.

Recently, graphics processing units (GPUs) have been widely used for image processing. Dedicated GPUs have memory bandwidth that can be an order of magnitude faster than CPU RAM. GPUs are able to perform hardware image bilinear interpolation, though with limited 1/16th precision for the position. Even for integrated GPUs (on laptops typically), the GPU can perform significantly more floating point operations per second (FLOPS) than the CPU cores combined. See Table 1 for some examples of configurations and comparative performance.

Accelerator	Description	Bandwidth	Float ops	Double ops
0	INTEL i7-6600U CPU (POCL 1.3)	19 GB/s	24 G/s	28 G/s
1	INTEL i7-6600U GPU (NEO 18.21.10858)	23 GB/s	380 G/s	95 G/s
2	AMD Radeon RX 480 (2766.4)	209 GB/s	5.7 T/s	279 G/s
3	NVIDIA TITAN V (384.130)	610 GB/s	13.7 T/s	6.9 T/s

Table 1: Compared configurations: The first accelerator is a 4-core CPU, while the other accelerators are GPUs with different levels of performance. The bandwidth and number of operations per second were obtained with Clpeak, which estimates the actual peak figures which can be obtained with OpenCL.

While previously some GPU implementations of cubic B-spline interpolation have been publicly available [13, 5, 11], to our knowledge none is able to handle higher orders (except [13] that supports orders up to 5). Moreover no analysis has been done for the floating-point precision (single or double) required for these interpolations. Due to memory bandwidth considerations and double precision computation costs, most GPU programs choose to operate on floats. Besides delivering a GPU implementation for B-spline interpolation from orders 0 to 11, we shall study the impact of the computation and storage precision on the results.

Contributions. We propose an efficient GPU implementation of B-spline interpolation for orders up to 11, which can handle both single and double floating point precision. The two main optimization tricks to achieve this consist in: (1) transposing the B-spline coefficients before the prefiltering of the rows, and (2) dividing columns into subregions in order to use more threads. We compare the impact of using single or double precision on the accuracy and conclude on the best precision to use. Also, we compare the performance using float or double precision on all supported orders for several accelerators. We show that to get an accuracy gain with double precision, all computation and storage must use double precision, but that using double precision nearly doubles the computation time.

This article is organized as follows: Section 2 is a short introduction to B-spline interpolation. Section 3 introduces briefly GPU architectures, gives details on our implementation, and compares it to other methods. In Section 4 we compare speed and accuracy for several orders, choices of floating point precision and for different accelerators.

2 Introduction to B-spline Interpolation

In this section we introduce the reader to B-spline interpolation of uniformly spaced data points (such as image pixels). We consider the efficient and stable approach of Unser et al., proposed in [16], that is based on linear filtering and works for any order. This is a two-step interpolation method whose first step, namely the prefiltering step, can be decomposed into a cascade of exponential filters, themselves separated into two complementary causal and anticausal components. Two slightly different prefiltering algorithms, whose precision is proven to be controlled thanks to a rigorous boundary handling, were introduced in [4] for 1D and 2D finite signals. More precisely, in both versions the control of the error is guaranteed by the use of adequate truncation indices. Our proposed GPU implementation, described in Section 3, is based on this method.

Section 2.1 introduces the method of [16] for the B-spline interpolation of one-dimensional infinite signals. Section 2.2 proposes a summary of the B-spline interpolation method of [4] for finite one-dimensional signals. The extension to higher dimensions, and in particular to dimension 2, is discussed in Section 2.3. More details, and in particular, proofs, can be found in [4].

2.1 Case of an Infinite Signal

Let n be a non-negative integer. First, we define the B-spline interpolate of order n of an infinite one-dimensional signal.

Definition 1. *The normalized B-spline function of order n, noted $\beta^{(n)}$, is defined recursively by*

$$\beta^{(0)}(x) = \begin{cases} 1, & -\frac{1}{2} < x < \frac{1}{2} \\ \frac{1}{2}, & x = \pm\frac{1}{2} \\ 0, & \textit{otherwise} \end{cases} \qquad \textit{and for } n \geq 0, \quad \beta^{(n+1)} = \beta^{(n)} * \beta^{(0)}, \tag{2}$$

where the symbol $*$ denotes the convolution operator.

Definition 2 (B-spline interpolation)**.** *The B-spline interpolate of order n of a discrete signal $f \in \ell^\infty(\mathbb{Z})$ is the function $\varphi^{(n)}$ defined for $x \in \mathbb{R}$ by*

$$\varphi^{(n)}(x) = \sum_{i \in \mathbb{Z}} c_i \beta^{(n)}(x - i), \tag{3}$$

where the B-spline coefficients $c = (c_i)_{i \in \mathbb{Z}}$ are uniquely characterized by the interpolating condition

$$\varphi^{(n)}(k) = f_k, \quad \forall k \in \mathbb{Z}. \tag{4}$$

Given a bounded signal f and a real x, computing the right hand side of (3) requires two evaluations: the signal $c = (c_i)_{i\in\mathbb{Z}}$ and the $\beta^{(n)}(x-i)$. This explains the two steps involved in the computation:

- Step 1 (prefiltering or direct B-spline transform) computes the B-spline coefficients c. The method that we consider for the prefiltering step is presented below.

- Step 2 (indirect B-spline transform [16]) reconstructs the signal values. Given the Dirac comb of B-spline coefficients $\mathbf{c} = \sum_{i\in\mathbb{Z}} c_i\delta_i$ the value of $\varphi^{(n)}(x)$ in (3) is computed at any location $x \in \mathbb{R}$ as a convolution of $\mathbf{c}$ with the finite signal $\beta^{(n)}(x-.)$. Details about this step can be found in [4].

Prefiltering as a linear filtering. Except in the simplest cases, $n = 0$ or $n = 1$, in which case $\beta^{(n)}(k-i) = \delta_i(k)$ and so $c_i = f_i$, the determination of the coefficients c_i from f so as to satisfy (4) is not straightforward. In the following we summarize how the prefiltering step is rewritten in [16] as a linear filtering process. Proofs and details can be found in [4].

As introduced in [16], the prefiltering step boils down to the discrete convolution

$$c = (b^{(n)})^{-1} * f, \tag{5}$$

where the filter $(b^{(n)})^{-1}$ denotes the inverse of the filter $b^{(n)}$ for the discrete convolution operator. This filter can be decomposed into a cascade of exponential filters, themselves separated into two complementary causal and anticausal components.

Decomposition of $(b^{(n)})^{-1}$ into exponential filters. The exponential filters involved are parametrized by negative numbers called the poles of the B-spline interpolation. For order n, the $\tilde{n} = \lfloor \frac{n}{2} \rfloor$ poles $(z_i)_{1\le i\le \tilde{n}}$ are defined and computed as described in [4], and are arranged so that

$$-1 < z_1 < z_2 < \cdots < z_{\tilde{n}} < 0. \tag{6}$$

Let $-1 < \alpha < 0$, α playing the role of one z_i. Denote by $k^{(\alpha)} \in \mathbb{R}^{\mathbb{Z}}$ the causal filter and by $l^{(\alpha)} \in \mathbb{R}^{\mathbb{Z}}$ the anticausal filter defined for $i \in \mathbb{Z}$ by

$$k_i^{(\alpha)} = \begin{cases} 0 & i < 0 \\ \alpha^i & i \ge 0 \end{cases} \quad \text{and} \quad l_i^{(\alpha)} = \begin{cases} 0 & i > 0 \\ \alpha^{-i} & i \le 0. \end{cases} \tag{7}$$

Set $h^{(\alpha)} = -\alpha l^{(\alpha)} * k^{(\alpha)}$. The filter $h^{(\alpha)}$ is called exponential filter as it verifies for $j \in \mathbb{Z}$,

$$h_j^{(\alpha)} = \frac{\alpha}{\alpha^2 - 1}\alpha^{|j|}. \tag{8}$$

It can be shown that the filter $(b^{(n)})^{-1}$ can be decomposed as

$$(b^{(n)})^{-1} = \gamma^{(n)} h^{(z_{\tilde{n}})} * \cdots * h^{(z_1)}, \tag{9}$$

where the normalization constant $\gamma^{(n)}$ is defined by

$$\gamma^{(n)} = \begin{cases} 2^n n! & n \text{ even} \\ n! & n \text{ odd}. \end{cases} \tag{10}$$

Prefiltering algorithm. This provides a new expression for the B-spline coefficients c,

$$c = \gamma^{(n)} h^{(z_{\tilde{n}})} * \cdots * h^{(z_1)} * f. \tag{11}$$

To simplify, define recursively the signal $c^{(i)} \in \mathbb{R}^{\mathbb{Z}}$ for $i \in \{0, \ldots, \tilde{n}\}$ by

$$c^{(0)} = f \quad \text{and for } i \geq 1, \quad c^{(i)} = h^{(z_i)} * c^{(i-1)}. \tag{12}$$

We have $c = \gamma^{(n)} c^{(\tilde{n})}$. Thus the computation of the prefiltering step can be decomposed into $\tilde{n}$ successive filtering steps with exponential filters that can themselves be separated into two complementary causal and anticausal components. This leads to a theoretical prefiltering algorithm that cannot be used in practice because it requires an infinite input signal. Turning this algorithm into a practical one, that is, applicable to a finite signal, is the subject of Section 2.2.

2.2 Case of a Finite Signal

In practice the signal $\underline{f}$ to be interpolated is finite and discrete, i.e., $\underline{f} = (\underline{f}_i)_{0 \leq i \leq K-1}$ for a given positive integer K. An extension of $\underline{f}$ outside $\{0, \ldots, K-1\}$ is required. B-spline interpolation theory can then be applied to the extended signal $f \in \ell^\infty(\mathbb{Z})$. To simplify the notation, in the following no distinction will be made between the signal $\underline{f}$ and its extension f when there is no ambiguity.

Extension on a finite domain. Let $x \in [0, K-1]$. As presented in the following, to compute $\varphi^{(n)}(x)$ with a relative precision ε it is sufficient to extend the signal to a finite domain $\{-L^{(n,\varepsilon)}, \ldots, K-1+L^{(n,\varepsilon)}\}$ where $L^{(n,\varepsilon)}$ is a positive integer that only depends on the B-spline order n and the desired precision ε. The precision is relative to the values of the signal. A relative precision of ε means that the error committed is less than $\varepsilon \sup_{k \in \mathbb{Z}} |f_k| = \varepsilon \|f\|_\infty$. In practice the images are large enough so that $L^{(n,\varepsilon)} < K$ and it is possible to express the extension as a boundary condition around 0 and $K-1$ (otherwise the extension is obtained by iterating the boundary condition). The most classical boundary condition choices are summarized in Table 2.

Extension	Signal *abcde*
Constant	*aaa*|*abcde*|*eee*
Half-symmetric	*cba*|*abcde*|*edc*
Whole-symmetric	*dcb*|*abcde*|*dcb*
Periodic	*cde*|*abcde*|*abc*

Table 2: Classical boundary extensions of the signal *abcde* by $L = 3$ values.

Prefiltering computation using the extension. For computing the B-spline coefficients using the prefiltering decomposition given in (11), only the first exponential filter $h^{(z_1)}$ is applied directly to $f = c^{(0)}$. Therefore, for $i \geq 1$ the intermediate filtered signals $c^{(i)}$ are known a priori only where they are computed. Considering this, the two following approaches were proposed in [4] in order to perform the prefiltering.

- **Approach 1: extended domain.** The intermediate filtered signals $c^{(i)}$ are computed in a larger domain than $\{-\tilde{n}, \ldots, K-1+\tilde{n}\}$. This works with any extension.
- **Approach 2: exact domain.** The extension, expressed as a boundary condition, is chosen so that it is "transmitted" after the application of each exponential filter $h^{(z_i)}$. The intermediate filtered signals $c^{(i)}$ (and the B-spline coefficients c) verify the same boundary condition as the input f and only need to be computed in $\{0, \ldots, K-1\}$.

Among the four classical boundary extensions presented in Table 2, the periodic, half-symmetric and whole-symmetric boundary conditions are transmitted after the application of an exponential filter. For the periodic extension the filtered signal by any filter always remains periodic. For the half-symmetric and whole-symmetric extensions it is a consequence of the symmetry of the exponential filters. However, this property is not satisfied for the constant extension, so that Algorithm 2 is not applicable in this case. The second approach is more efficient as it requires fewer computations and is the considered approach in the following.

Section 2.2.1 details the general method for computing an exponential filter application on a finite domain. Then, Section 2.2.2 describes the algorithm for computing the B-spline coefficients of a finite signal with a given precision when the boundary condition is transmitted. Finally, Section 2.2.3 presents the simple algorithm for performing the indirect B-spline transform, i.e., for evaluating the interpolated values.

2.2.1 Application of the Exponential Filters

Let $s \in \ell^\infty(\mathbb{Z})$ be an infinite discrete signal and let $-1 < \alpha < 0$. The application of the exponential filter $h^{(\alpha)} = -\alpha l^{(\alpha)} * k^{(\alpha)}$ to the signal s is computed in the domain of interest $\{0, \dots, K-1\}$ as follows. To simplify the notation we set $s^{(\alpha)} = k^{(\alpha)} * s$ so that

$$h^{(\alpha)} * s = -\alpha l^{(\alpha)} * s^{(\alpha)}. \tag{13}$$

Causal filtering. Given the initialization

$$s_0^{(\alpha)} = \left(k^{(\alpha)} * s\right)_0 = \sum_{i=0}^{+\infty} \alpha^i s_{-i}, \tag{14}$$

the application of the causal filter $k^{(\alpha)}$ to s can be computed recursively from $i = 1$ to $i = K-1$ according to the recursion formula

$$s_i^{(\alpha)} = s_i + \alpha s_{i-1}^{(\alpha)}. \tag{15}$$

Anticausal filtering initialization. As shown in [4, p. 11] the initialization of the renormalized anticausal filtering $-\alpha l^{(\alpha)} * s^{(\alpha)} = h^{(\alpha)} * s$ is given by

$$\left(h^{(\alpha)} * s\right)_{K-1} = \frac{\alpha}{\alpha^2 - 1}\left(s_{K-1}^{(\alpha)} + \left(l^{(\alpha)} * s\right)_{K-1} - s_{K-1}\right), \tag{16}$$

where

$$\left(l^{(\alpha)} * s\right)_{K-1} = \sum_{i=0}^{+\infty} \alpha^i s_{K-1+i}. \tag{17}$$

Anticausal filtering. The renormalized anticausal filtering is computed recursively from $i = K-2$ to $i = 0$ according to the following formula,

$$\left(h^{(\alpha)} * s\right)_i = \alpha\left(\left(h^{(\alpha)} * s\right)_{i+1} - s_i^{(\alpha)}\right). \tag{18}$$

Approximation of the initialization values. The two infinite sums in (14) and (17) cannot be computed numerically. Let N be a non-negative integer. The initialization values are approximated by truncating the sums at index N so that

$$\left(k^{(\alpha)} * s\right)_0 \simeq \sum_{i=0}^{N} \alpha^i s_{-i}, \tag{19}$$

and

$$\left(l^{(\alpha)} * s\right)_{K-1} \simeq \sum_{i=0}^{N} \alpha^i s_{K-1+i}. \tag{20}$$

Algorithm. The general method for computing the application of the exponential filter $h^{(\alpha)}$ to a discrete signal in a finite domain is summarized in Algorithm 1. Note that it is sufficient to know the signal in the domain $\{-N, \ldots, K-1+N\}$. It consists of $6(N+1)+4(K-1)$ operations.

Algorithm 1: Application of the exponential filter $h^{(\alpha)}$ to a discrete signal

Input : A pole $-1 < \alpha < 0$, a truncation index N and a discrete signal s (whose values are known in $\{-N, \ldots, K-1+N\}$)

Output: The filtered signal $h^{(\alpha)} * s$ at indices $\{0, \ldots, K-1\}$

```
Compute s_0^(α) using (19)
for i = 1 to K − 1 do
    Compute s_i^(α) using (15)
end
Compute (l^(α) ∗ s)_{K−1} using (20)
Compute (h^(α) ∗ s)_{K−1} using (16)
for i = K − 2 to 0 do
    Compute (h^(α) ∗ s)_i using (18)
end
```

2.2.2 Prefiltering of a Finite Signal

Let $\varepsilon > 0$. In [4] two algorithms for computing the B-spline coefficients of a finite signal with precision ε were proposed. In this section we present the version where the input signal is extended with a boundary condition that is assumed to be transmitted after the application of the exponential filters.

Truncation indices. Define $(\mu_j)_{1 \le j \le \tilde{n}}$ by

$$\begin{cases} \mu_1 = 0 \\ \mu_k = \left(1 + \frac{1}{\log|z_k| \sum_{i=1}^{k-1} \frac{1}{\log|z_i|}}\right)^{-1}, \quad 2 \le k \le \tilde{n}. \end{cases} \tag{21}$$

Define for $1 \le i \le \tilde{n}$ the truncation index

$$N^{(i,\varepsilon)} = \left\lfloor \frac{\log\left(\varepsilon \rho^{(n)} (1-z_i)(1-\mu_i) \prod_{j=i+1}^{\tilde{n}} \mu_j\right)}{\log|z_i|} \right\rfloor + 1, \tag{22}$$

where

$$\rho^{(n)} = \left(\prod_{j=1}^{\tilde{n}} \frac{1+z_j}{1-z_j}\right)^2. \tag{23}$$

Algorithm. As the boundary condition is transmitted after the application of the exponential filters, the $c^{(i)}$ for $0 \leq i \leq \tilde{n}$ share the same boundary condition. Then, $c^{(i)}$ can be computed at any index from the values of $c^{(i-1)}$ in $\{0, \ldots, K-1\}$ using Algorithm 1 (with truncation index $N^{(i,\varepsilon)}$). Indeed, we notice that the initialization values in (19) and (20) only depend on the values of $c^{(i-1)}$ in $\{0, \ldots, K-1\}$. The prefiltering algorithm with a transmitted boundary condition is described in Algorithm 2. The choice for the truncation indices $N^{(i,\varepsilon)}$ guarantees a precision ε for $n \leq 16$, as stated in the next theorem [4].

Theorem 1. *Let $\varepsilon > 0$ and f be a finite signal of length at least 4. Assume that $n \leq 16$ and that f is extended with a boundary condition that is transmitted after the application of the exponential filters.*

Then, the computation of the B-spline coefficients of f using Algorithm 2 with the truncation indices $(N^{(i,\varepsilon)})_{1 \leq i \leq \tilde{n}}$ has a precision of ε, i.e., the error committed is less than $\varepsilon \|f\|_\infty$.

Algorithm 2: Prefiltering algorithm (with a transmitted boundary condition)

Input : A finite discrete signal f of length K, a boundary condition that is transmitted, a B-spline interpolation order $n \leq 16$ and a precision ε (the poles and the normalization constant are assumed to be known)

Output: The B-spline coefficients c of order n at indices $\{0, \ldots, K-1\}$ with precision ε

Precomputations

1 **for** $1 \leq i \leq \tilde{n}$ **do**
2 | Compute the truncation index $N^{(i,\varepsilon)}$ using (22)
3 **end**

Prefiltering of the signal:

4 Set $c_k^{(0)} = f_k$ for $k \in \{0, \ldots, K-1\}$
5 **for** $i = 1$ **to** $\tilde{n}$ **do**
6 | Compute $c_k^{(i-1)}$ for $k \in \{-N^{(i,\varepsilon)}, \ldots, -1\} \cup \{K, \ldots, K-1+N^{(i,\varepsilon)}\}$ using the boundary condition
7 | Compute $c_k^{(i)} = \left(h^{(z_i)} * c^{(i-1)}\right)_k$ for $k \in \{0, \ldots, K-1\}$ using Algorithm 1 with truncation index $N^{(i,\varepsilon)}$
8 **end**
9 Renormalize $c = \gamma^{(n)} c^{(\tilde{n})}$ using (10) *The normalization can be moved to Algorithm 4*

Particular cases. For the three boundary conditions that are transmitted, the anticausal initialization value in (16) can be computed without using (20). In practice, when one of these three boundary conditions is used in Algorithm 2, a slightly different version of Algorithm 1 is called in Line 6. The boundary extension is added to the input list and the computation of $\left(h^{(\alpha)} * s\right)_{K-1}$ (see Line 5 and Line 6) is done using the corresponding formula in Table 3.

2.2.3 Indirect B-spline Transform: Computation of the Interpolated Value

The indirect B-spline transform reconstructs the signal values from the B-spline representation. Given the B-spline coefficients c in $\{-\tilde{n}, \ldots, K-1+\tilde{n}\}$, the interpolated value $\varphi^{(n)}(x)$ can be computed with precision ε, using (3), as the convolution of $\sum_{i=-\tilde{n}}^{K-1+\tilde{n}} c_i \delta_i$ with the compactly supported function $\beta^{(n)}$. The computation of the indirect B-spline transform at location $x \in [0, K-1]$ is presented in Algorithm 3.

Extension	$\left(h^{(\alpha)} * s\right)_{K-1}$
Half-symmetric	$\frac{\alpha}{\alpha-1}\left(k^{(\alpha)} * s\right)_{K-1}$
Whole-symmetric	$\frac{\alpha}{\alpha^2-1}\left(\left(k^{(\alpha)} * s\right)_{K-1} + \alpha\left(k^{(\alpha)} * s\right)_{K-2}\right)$
Periodic	$-\alpha\left(s^{(\alpha)}_{K-1} + \alpha\sum_{i=0}^{N-1} s^{(\alpha)}_i \alpha^i\right)$

Table 3: anticausal initialization value $\left(h^{(\alpha)} * s\right)_{K-1}$ for particular boundary conditions. N denotes the truncation index. See [4] for details.

Algorithm 3: Indirect B-spline transform

Input : A finite discrete signal f of length K, a B-spline interpolation order n, the corresponding B-spline coefficients c in $\{-\tilde{n}, \ldots, K-1+\tilde{n}\}$ and $x \in [0, K-1]$

Output: The interpolated value $\varphi^{(n)}(x)$

1 $x_0 \leftarrow \lceil x - (n+1)/2 \rceil$
2 Initialize $\varphi^{(n)}(x) \leftarrow 0$
3 **for** $k = 0$ **to** $\max(n, 1)$ **do**
4 | Update $\varphi^{(n)}(x) \leftarrow \varphi^{(n)}(x) + c_k \beta^{(n)}(x - (x_0 + k))$
5 **end**

Assume the B-spline coefficients are computed with precision ε using Algorithm 2. Then, $\varphi^{(n)}(x)$ is also computed with precision ε because $\sum_{k\in\mathbb{Z}} \beta^{(n)}(x-k) = 1$. We recall that in Algorithm 2 the B-spline coefficients are known in $\{-\tilde{n}, \ldots, -1\} \cup \{K, \ldots, K-1+\tilde{n}\}$ thanks to the boundary condition.

2.3 Extension to Higher Dimensions

The one-dimensional B-spline interpolation theory (Section 2.1) and practical algorithms (Section 2.2) are easily extended to higher dimensions by using tensor-product basis functions. The multi-dimensional prefiltering is performed by applying the one-dimensional prefiltering successively along each dimension. The indirect B-spline transform is efficiently computed as the convolution with a separable and compactly supported function.

Note d the dimension. The Normalized B-spline function of order n is defined in dimension d as follows.

Definition 3. *The Normalized B-spline function of order n and dimension d, noted $\beta^{(n,d)}$, is defined for $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$ by*

$$\beta^{(n,d)}(x) = \prod_{j=1}^{d} \beta^{(n)}(x_j). \tag{24}$$

Then, the B-spline interpolate of order n of a discrete signal $f \in \ell^\infty(\mathbb{Z}^d)$ can be naturally defined as follows.

Definition 4. *The B-spline interpolate of order n of a discrete signal $f \in \ell^\infty(\mathbb{Z}^d)$ is the function $\varphi^{(n)} : \mathbb{R}^d \mapsto \mathbb{R}$ defined for $x \in \mathbb{R}^d$ by*

$$\varphi^{(n)}(x) = \sum_{i\in\mathbb{Z}^d} c_i \beta^{(n,d)}(x-i), \tag{25}$$

where the B-spline coefficients $c = (c_i)_{i\in\mathbb{Z}^d}$ are uniquely defined by the interpolation condition

$$\varphi^{(n)}(k) = f_k, \quad \forall k \in \mathbb{Z}^d. \tag{26}$$

The B-spline interpolation in dimension d is also a two-step interpolation method. The B-spline coefficients c can be computed by filtering f successively along each dimension by $\left(b^{(n)}\right)^{-1}$. In other words, the multi-dimensional prefiltering is decomposed in successive one-dimensional prefilterings along each dimension.

2.3.1 Algorithms in 2D

A particular and interesting case is given by $d = 2$ where the finite discrete signals to be interpolated are images.

Separable prefiltering. The B-spline coefficients are obtained by applying successively the one-dimensional prefiltering on the columns and on the rows. More precisely, let $g \in \mathbb{R}^{\mathbb{Z}^2}$ and $(i, j) \in \mathbb{Z}^2$. We denote $C^j(g) \in \mathbb{R}^{\mathbb{Z}}$ the j-th column of g so that $C^j(g)_i = g_{i,j}$. Similarly, we denote $R^i(g) \in \mathbb{R}^{\mathbb{Z}}$ the i-th row of g so that $R^i(g)_j = g_{i,j}$. Define $c_{\text{col}}(f) \in \mathbb{R}^{\mathbb{Z}^2}$, the one-dimensional prefiltering of the columns of f, by their columns

$$C^j(c_{\text{col}}(f)) = C^j(f) * (b^{(n)})^{-1}. \tag{27}$$

Then, the B-spline coefficients c are given by the one-dimensional prefiltering of the lines of c_{col} i.e.

$$R^i(c) = R^i(c_{\text{col}}(f)) * (b^{(n)})^{-1}. \tag{28}$$

Prefiltering of a finite image. In practice images are finite and an arbitrary extension has to be chosen. Assume that the boundary condition is separable and is transmitted after the application of the exponential filters. Let f be an image of size $K \times L$. In order to compute interpolated values in $[0, K-1] \times [0, L-1]$ the B-spline coefficients c of f have to be computed in $\{0, \dots, K-1\} \times \{0, \dots, L-1\}$. According to (27) and (28) the prefiltering can be done by applying Algorithm 2 successively on the columns and on the rows. Denote $\varepsilon' = \frac{\varepsilon\rho^{(n)}}{2}$. Using the truncation indices $N^{(i,\varepsilon')}$ for both dimensions guarantees a precision ε for $n \leq 16$, as stated in the next theorem.

Theorem 2. *Assume $n \leq 16$. Let $\varepsilon > 0$ and f be a finite image of size at least 4 along each dimension. Assume that the boundary condition is separable and is transmitted after the application of the exponential filters. Denote $\varepsilon' = \frac{\varepsilon\rho^{(n)}}{2}$.*

Then, the computation of the B-spline coefficients of f by applying Algorithm 2 successively on the columns and on the rows with truncation indices $(N^{(i,\varepsilon')})_{1\leq i\leq \tilde{n}}$ (as in Algorithm 5) have a precision of ε.

Computation of interpolated values. The two-dimensional indirect B-spline transform is efficiently performed, as described in Algorithm 4, as a convolution with the separable and compactly supported function $\beta^{(n,2)}$. We recall that the B-spline coefficients are known outside of $\{0, \dots, K-1\} \times \{0, \dots, L-1\}$ thanks to the boundary condition. Finally, the two-dimensional B-spline interpolation of an image is presented in Algorithm 5. It also has precision ε.

Algorithm 4: Two-dimensional indirect B-spline transform

Input : An image f of size $K \times L$, a B-spline interpolation order n, the corresponding B-spline coefficients c in $\{-\tilde{n}, \ldots, K-1+\tilde{n}\} \times \{-\tilde{n}, \ldots, L-1+\tilde{n}\}$ and a location $(x, y) \in [0, K-1] \times [0, L-1]$

Output: The interpolated value $\varphi^{(n)}(x, y)$

```
x_0 ← ⌈x − (n + 1)/2⌉
y_0 ← ⌈y − (n + 1)/2⌉
for k = 0 to max(1, n) do
    Tabulate xBuf[k] ← β^(n)(x − (x_0 + k))
    Tabulate yBuf[k] ← β^(n)(y − (y_0 + k))
end
Initialize φ^(n)(x, y) ← 0
for l = 0 to max(1, n) do
    Initialize s ← 0
    for k = 0 to max(1, n) do
        Update s ← s + c_{x_0+k, y_0+l} xBuf[k]
    end
    Update φ^(n)(x, y) ← φ^(n)(x, y) + s yBuf[l]
end
```

Algorithm 5: Two-dimensional B-spline interpolation

Input : An image f of size $K \times L$, a boundary condition that is transmitted, a B-spline interpolation order $n \leq 16$, a precision ε and a list of pixel locations $(x_j, y_j)_{1 \leq j \leq J} \in ([0, K-1] \times [0, L-1])^J$

Output: The interpolated values $(\varphi^{(n)}(x_j, y_j))_{j \in J}$ (with precision ε)

```
Compute ε' = ερ^(n)/2.
Compute with precision ε' the one-dimensional prefiltering of the columns of f, noted c_col,
  using Algorithm 2
Compute with precision ε' the one-dimensional prefiltering of the rows of c_col, noted c, using
  Algorithm 2
for j = 1 to J do
    Compute, from the B-spline coefficient c, the interpolated value φ^(n)(x_j, y_j) using
      Algorithm 4
end
```

3 The Proposed GPU Implementation

In this section, we detail our GPU implementation of the B-spline interpolation method presented in Section 2. Our implementation is focused on 2D images. It supports B-spline interpolation from orders 0 to 11, as well as the border conditions presented in Table 2 (except for the constant extension).

3.1 Motivations

Most image processing algorithms can be implemented on GPUs, which are highly parallel computing devices. In 2018, a midrange GPU was able to perform 6 Tera FLOPS, while a CPU was able to perform about 0.04 Tera FLOPS per core when using the extended CPU instruction set. For real-time

computing, many image processing programs propose to run entirely on GPUs.

To get optimal performance, transfers between the GPU and the CPU must be avoided and thus the whole image processing pipeline must be implemented on GPUs. Interpolation is an important image processing technique for many algorithms. GPUs have native operators for nearest neighbor interpolation and bilinear interpolation, but many algorithms benefit from a more precise interpolation algorithm.

Open Computing Language (OpenCL) is a programming language close to C that enables to target GPUs, CPUs and FPGAs. Special tuning is required for best performance for a given target, but clearly one advantage of OpenCL is that a given code can run on many platforms. While in this paper we specifically target GPUs, our code will be able to run on CPUs as well (we have not tested our code on FPGAs).

3.2 A Very Short Introduction to GPU Architectures

In order to understand some of the concerns in the following implementation, a short introduction to GPUs is required. Current GPU architectures work on a SIMT model (Single Instruction - Multiple Threads). Essentially threads are grouped together in a "warp" (also named "wavefront") and will always be executing the same instructions in lock-steps. GPUs run many of these groups of threads in parallel and share their computational and memory GPU resources.

Essentially, GPU algorithms are either limited by the available computational resources or by the available bandwidth. GPUs run more groups of threads than what the hardware can handle in order to maximize both resources: some groups will be sleeping, waiting typically for memory commands to finish, while others use the compute units. To maximize computational use, there must always be some threads doing computations, while to maximize bandwidth use, there must always be some threads requesting memory contents. For 2018 GPU architectures, at peak bandwidth (excluding cache) and peak computing, reading a float in memory can cost as much as doing 100 floating point operations. Some GPUs can have more than 100K simultaneous threads (though only a small portion would actually use the computational resources at the same time).

In the case of B-spline interpolation, not many operations are required per memory access, thus the algorithm will be mainly bandwidth limited. Advanced techniques can be used to maximize bandwidth usage. As we mentioned previously, while groups of threads are waiting on memory operations, which can have high latency if the data are not in cache, other groups of threads can be executed. This enables to reduce the impact of the latency of these memory operations, but in some cases, it is better in addition to interleave computation between the data loading and the use of the data. In addition, for optimal bandwidth usage, the threads inside a warp/wavefront must read consecutive elements in memory, ideally aligned to fit in as few cache lines as possible. Essentially, if one thread needs data inside a cache line, the whole line is loaded (which typically is 64B for the lowest cache, but is higher if loading from the GPU dedicated memory). Thus the cost is minimized if all threads need the same cache lines. More details can be found in technical documents such as [8, 2, 10].

3.3 Prefiltering Step

The prefiltering step is heavily bandwidth limited as only a few operations per loaded item are needed. Having an efficient memory access pattern is thus essential. As for each causal or anticausal pass, every image element needs to be read and written once, one could expect a good implementation to be as fast as a simple image copy, ignoring cache effects. We consider the prefiltering computation introduced in Section 2.2.2 as it doesn't require to add border lines. As in [4], the renormalization by $(\gamma^{(n)})^2$ of the B-spline coefficients is moved to the indirect B-spline transform step (see Section 3.4).

Naïve algorithm. A naïve GPU implementation of the prefiltering step consists in handling each row or column with a different thread. More precisely, the causal (or anticausal) filtering of each line is handled by a different thread during the application of the exponential filters. The corresponding algorithm is presented in Algorithm 6. This implementation has an ideal memory access pattern for the prefiltering of the columns but raises two main issues:

- For the prefiltering of the rows, each thread accesses simultaneously elements of different rows (and thus different cache lines), which is a terribly suboptimal memory access pattern.
- The size of the image being unknown at compilation, the *for* loops cannot be unrolled efficiently, thus preventing efficient memory latency reduction.

Algorithm 6: Naïve prefiltering

```
input  : I the input image and its dimensions, the spline order and the corresponding poles, a
         boundary condition, a precision ε
output: O the (unnormalized) prefiltered image (with precision ε)
1  for all poles α do
                                                          Causal filtering, Columns
2      for all columns j - One thread per j do
3          Specific handling of the top border                          See Section 2.2.2
4          for row i, from top to bottom do
5              O[i, j] ← I[i, j] + α · O[i − 1, j]      Note O[i − 1, j] doesn't need to be reloaded
6          end
7      end
                                                          Anticausal filtering, Columns
8      for all columns j - One thread per j do
9          Specific handling of the bottom border                       See Section 2.2.2
10         for row i, from bottom to top do
11             O[i, j] ← α · (O[i + 1, j] − O[i, j])
12         end
13     end
                                                          Causal filtering, Rows
14     for all rows j - One thread per j do
15         Specific handling of the left border                         See Section 2.2.2
16         for column i, from left to right do
17             O[j, i] ← O[j, i] + α · O[j, i − 1]
18         end
19     end
                                                          Anticausal filtering, Rows
20     for all rows j - One thread per j do
21         Specific handling of the right border                        See Section 2.2.2
22         for column i, from right to left do
23             O[j, i] ← α · (O[j, i + 1] − O[j, i])
24         end
25     end
26 end
```

The proposed algorithm. Our algorithm does the following in order to speed up the computations:

- Improved memory pattern: Our implementation uses transposition so that the inefficient prefiltering of the rows is performed as an efficient prefiltering of columns. We use the efficient transpose operation from OpenCV [3]. Its cost is close to a buffer copy. Note that for a given order, column prefiltering operations can be grouped together and done consecutively. The same property holds for line prefiltering, thus only two transpositions are needed.

- Improved latency reduction: For most images and GPUs, the number of available threads is larger than the width or the height of the image. To use more threads, and thus better benefit from hardware latency reduction, the image is divided into subregions and each group of threads is only tasked one subregion.

  Each thread, instead of computing the causal (or anticausal) filtering for an entire column, computes it only for a subset of size K (which value is specified below). When the column subsection doesn't start at the border, the causal initialization is computed from the preceding N elements using (19). Similarly, when the column subsection doesn't end at the border, the anticausal initialization is computed from the following N elements using (20). By choosing for N the same value as the current truncation index, i.e., $N = N^{(i,\varepsilon')}$, we ensure that the division is acceptable in terms of precision (Theorem 2). Our algorithm therefore does have a relative precision of ε.

  Note that, because of the subdivision, many elements are read twice per causal (or anticausal) filtering. More computations are required, but our algorithm is more efficient than the naïve one, since the redundancies are handled by different threads. We found a subsection length of $K = 256$ to be a good performance compromise. Moreover to improve latency reduction further, we manually separated the *for* loops into a part of fixed size, executed several times, and the remaining iterations. The part of fixed size can be efficiently unrolled and thus computation and memory accesses can be interleaved by the compiler.

The proposed algorithm is summarized in Algorithm 7. The assignment of threads for both the naïve and optimized algorithm are illustrated on Figure 1. Note that it is possible to optimize further, for orders above 4, by merging causal passes together, as well as anticausal passes. Indeed the causal passes can be made sequentially, followed by all the anticausal passes, modulo a different handling of image borders. The merged passes would have the same bandwidth need as one pass, which would give a significant speedup given that the prefiltering is bandwidth limited. We do not use this optimization as, except for the periodic extension, the computations for the causal and anticausal initializations do not hold and errors are introduced near the image boundary.

Comparison with other methods. Other methods have been proposed to implement efficient B-spline filtering on GPU.

- In [11], the authors implement optimized cubic B-spline filtering for 1D, 2D and 3D dimensional data. But the speed of the filtering varies depending on the filtered dimension. For a 1024×1024 image on a NVIDIA GeForce 9800 GTX, the prefiltering takes at best 2ms for the y dimension, while they get at best 21 ms for the x one.

  Considering the transposition takes about the same time as a pass of our prefiltering, for the cubic B-spline we take twice the time for the rows than for the columns. This ratio reduces with the order of the B-spline.

Algorithm 7: Optimized prefiltering

input : I the input image and its dimensions, the spline order and the corresponding poles, a boundary condition, a precision ε

output: O the (unnormalized) prefiltered image (with precision ε)

1 First time we read O from I. Use a buffer O_2 to have different source and destination for each step.

2 **for** *all poles* α **do**

Causal filtering, Columns

3 **for** *all columns* j *- Several threads per* j **do**

4 **if** *Image border* **then**

5 Specific handling of the top border *See Section 2.2.2*

6 **else**

7 Initialize first item of the treated region of O_2 by accessing rows above from O using (19) with the corresponding truncation index

8 **end**

9 **for** *row* i*, from top of the region to bottom of the region* **do**

10 $O_2[i,j] \leftarrow O[i,j] + \alpha \cdot O_2[i-1,j]$

11 **end**

The above loop is written with manual partial unrolling

12 **end**

Anticausal filtering, Columns

13 **for** *all columns* j *- Several threads per* j **do**

14 **if** *Image border* **then**

15 Specific handling of the bottom border *See Section 2.2.2*

16 **else**

17 Initialize first item of the treated region of O by accessing rows below from O_2 using (20) with the corresponding truncation index

18 **end**

19 **for** *row* i*, from bottom of the region to top of the region* **do**

20 $O[i,j] \leftarrow \alpha \cdot (O[i+1,j] - O_2[i,j])$

21 **end**

The above loop is written with manual partial unrolling

22 **end**

23 **end**

24 Transpose O

25 Repeat from Line 2 to Line 23

26 Transpose O

- In [5], the authors improve over [11] for the case of 2D images, but still restricting to cubic B-spline. The authors observe the precision of the prefiltering is respected by writing the passes as a 2D 15×15 convolution, which they implement as two separable convolutions. We expect their approach to perform better than our approach, as the transposition is not required for good performance, and because in one pass of 1D convolution, both a causal and an anticausal pass is implemented, thus reducing the required memory bandwidth. This approach, however, is not scalable to higher orders, or for high precision computation (double), as the size of the convolution increases significantly. We note, however, if using periodic or half-symmetric boundary conditions, that one could implement the convolution as a multiplication in the

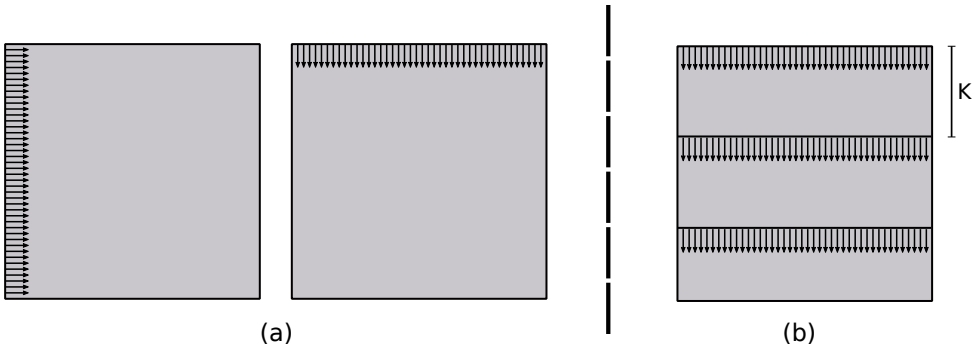

Figure 1: Illustration of naïve prefiltering (a) and our optimized prefiltering (b). On the left (a: naïve prefiltering), prefiltering is implemented by assigning a thread per row for the pass on the rows, and a thread per column for the pass on the columns. Each arrow represents a thread, while the gray area represents the image. On the right (b: optimized prefiltering), both are implemented as a pass on the columns (thanks to a transposition). To better use the available threads, the work region is separated in pieces (of height K) which are assigned to different threads.

Fourier domain or in the DCT domain respectively.

- In [13], the authors describe a GPU implementation of the B-spline interpolation for `elastix`, a module of `ITK`. While their implementation may not be as optimized as the ones of [11] and [5], orders up to 5 are supported, for 1D, 2D and 3D images.

3.4 Indirect B-spline Transform: Computation of Interpolated Values

For the computation of interpolated values we used a simple local GPU convolution filter, and used the separability of the B-spline kernel to reduce the number of computations. This corresponds to Algorithm 8.

Similarly to [4], we normalize in Line 33 the B-spline coefficients by $(\gamma^{(n)}/n!)^2$ instead of $(\gamma^{(n)})^2$ by simplifying the $n!$ factor in the evaluation of the kernel values in Line 4 and Line 5. We recall that

$$\frac{\gamma^{(n)}}{n!} = \begin{cases} 2^n & n \text{ even} \\ 1 & n \text{ odd.} \end{cases} \tag{29}$$

The input unnormalized coefficients are only known in $\{0, \dots, K-1\} \times \{0, \dots, L-1\}$ thus boundary handling may be required - using the same boundary condition as used for the prefiltering pass. To save unnecessary computations when all the coefficients for a given pixel location are inside the image, we use two different paths for computing the convolution. The fast path from Line 9 to Line 15 does not perform boundary handling while the slow path from Line 17 to Line 31 performs boundary handling for every coefficient.

In the provided code (available at the IPOL web page of this article[2]) a geometric transformation $\varphi : \mathbb{R}^2 \to \mathbb{R}^2$ is applied to the input image I. For an input image of size $K \times L$, the interpolated values are computed at locations $\varphi^{-1}(k, l)$ for $(k, l) \in \{0, \dots, K-1\} \times \{0, \dots, L-1\}$.

Depending on the application, global shift, homography, zoom, flow warping or others, the memory access pattern could be optimized, and in some cases the kernel coefficients precomputed. Moreover the last prefiltering transposition could be integrated to the interpolation (by inverting the read indices). Our code doesn't implement these optimizations.

[2] https://doi.org/10.5201/ipol.2019.257

Algorithm 8: Optimized two-dimensional indirect B-spline transform

Input : An image f of size $K \times L$, a B-spline interpolation order n, the corresponding unnormalized B-spline coefficients c in $\{0, \ldots, K-1\} \times \{0, \ldots, L-1\}$, a boundary condition and a location $(x, y) \in \mathbb{R}^2$

Output: The interpolated value $\varphi^{(n)}(x, y)$

```
1  x_0 ← ⌈x − (n + 1)/2⌉
2  y_0 ← ⌈y − (n + 1)/2⌉
3  for k = 0 to max(1, n) do
4      Tabulate xBuf[k] ← n! β^(n)(x − (x_0 + k))
5      Tabulate yBuf[k] ← n! β^(n)(y − (y_0 + k))
6  end
7  Initialize φ^(n)(x, y) ← 0
8  if (x_0, y_0) ∈ {0, ..., K − 1 − max(1, n)} × {0, ..., L − 1 − max(1, n)} then
                                                        Fast path without boundary handling
9      for l = 0 to max(1, n) do
10         Initialize s ← 0
11         for k = 0 to max(1, n) do
12             Update s ← s + c_{x_0+k, y_0+l} xBuf[k]
13         end
14         Update φ^(n)(x, y) ← φ^(n)(x, y) + s yBuf[l]
15     end
16 else
                                                        Slow path with boundary handling
17     for l = 0 to max(1, n) do
18         Initialize s ← 0
19         Set l' = y_0 + l
20         if l' ∉ {0, ..., L − 1} then
21             Update l' with its equivalent position in {0, ..., L − 1} using the boundary
                 condition
22         end
23         for k = 0 to max(1, n) do
24             Set k' = x_0 + k
25             if k' ∉ {0, ..., K − 1} then
26                 Update k' with its equivalent position in {0, ..., K − 1} using the boundary
                     condition
27             end
28             Update s ← s + c_{k', l'} xBuf[k]
29         end
30         Update φ^(n)(x, y) ← φ^(n)(x, y) + s yBuf[l]
31     end
32 end
33 Update φ^(n)(x, y) ← φ^(n)(x, y) (γ^(n)/n!)^2 using (29)                    Renormalization
```

3.5 Online Demo

This article is accompanied by an online demo[3] where the user can upload an image and apply to it a shift using B-spline interpolation with a selection of boundary conditions.

4 Experiments

All results (precision and running time) of this section are computed on a reference 4608×3456 image (shown on Figure 2), a shift of $(0.5, 0.5)$ and with half-symmetric border condition. The reference image contains flat areas, regions with different levels of textures and sharp edges.

Figure 2: Reference image of size 4608×3456 used for the experiments of Section 4. It contains flat areas, regions with different levels of textures and sharp edges. In all the experiments this image is shifted by $(0.5, 0.5)$ with half-symmetric border condition.

4.1 Float Versus Double Precision

In this section we would like to answer the following question: Should float precision or double precision be used for B-spline interpolation?

Tables 4 and 5 show experimentally that up to a target precision ε of $1e^{-6}$, floats are enough to guarantee the target precision, as the measured maximum relative error is below the target for the tested image. However, for smaller ε, all storage and computations must use double precision. The table shows that if any computation or storage uses float precision, the gain of using double precision for the other computations and storage is lost. The limit of double precision is reached around ε of $1e^{-13}$.

[3] https://doi.org/10.5201/ipol.2019.257

Configuration\ ε	1e-1	1e-2	1e-3	1e-4	1e-5	1e-6	1e-8	1e-12	1e-16
all float	2.24e-03	1.61e-04	1.28e-05	9.53e-07	4.51e-07	4.00e-07	4.00e-07	4.00e-07	4.00e-07
float prefilter computation	2.24e-03	1.61e-04	1.28e-05	9.30e-07	3.62e-07	3.65e-07	3.65e-07	3.65e-07	3.65e-07
float prefilter storage	2.24e-03	1.61e-04	1.27e-05	8.57e-07	2.50e-07	1.25e-07	1.25e-07	1.25e-07	1.25e-07
float interpolation kernel	2.24e-03	1.61e-04	1.26e-05	8.67e-07	3.30e-07	2.43e-07	2.43e-07	2.43e-07	2.43e-07
save result as float	2.24e-03	1.61e-04	1.27e-05	8.67e-07	2.40e-07	6.10e-08	5.68e-08	5.68e-08	5.68e-08
all double	2.24e-03	1.61e-04	1.27e-05	8.59e-07	2.45e-07	1.78e-08	9.52e-11	3.10e-14	6.34e-16

Table 4: Maximum relative error using the result obtained with the CPU code of [4] as reference (with target ε of $1e^{-20}$). Several target maximum errors ε are tested. The interpolation order 3 was used. Several precision modes are compared. On the first row, all computations and storage use single precision, while for the subsequent rows, double precision is used, except for the specified part of the algorithm which uses float.

Configuration\ ε	1e-1	1e-2	1e-3	1e-4	1e-5	1e-6	1e-8	1e-12	1e-16
all float	1.01e-05	7.00e-06	6.21e-06	6.21e-06	6.21e-06	6.21e-06	6.21e-06	6.21e-06	6.21e-06
float prefilter computation	9.99e-06	6.78e-06	6.01e-06	6.01e-06	6.01e-06	6.01e-06	6.01e-06	6.01e-06	6.01e-06
float prefilter storage	6.38e-06	6.39e-06	6.39e-06	6.39e-06	6.39e-06	6.39e-06	6.39e-06	6.39e-06	6.39e-06
float interpolation kernel	4.92e-06	2.44e-06	2.20e-06	2.30e-06	2.30e-06	2.30e-06	2.30e-06	2.30e-06	2.30e-06
save result as float	4.69e-06	6.20e-07	8.65e-08	5.83e-08	5.60e-08	5.59e-08	5.59e-08	5.59e-08	5.59e-08
all double	4.70e-06	5.94e-07	5.04e-08	6.05e-09	4.75e-10	6.74e-11	4.69e-13	1.42e-14	1.42e-14

Table 5: Maximum relative error using the result obtained with the CPU code of [4] as reference (with target ε of $1e^{-20}$). Several target maximum errors ε are tested. The interpolation order 11 was used. Several precision modes are compared. On the first row, all computations and storage use single precision, while for the subsequent rows, double precision is used, except for the specified part of the algorithm which uses float.

Tables 6 and 7, 8 show the running time for float and double precisions on several accelerators. The prefiltering and transposition running time is approximately doubled when using double precision on all accelerators, which is explained by the doubling of the bandwidth needs. Interpolation is more computationally heavy, and cache re-use is higher, and thus the performance ratio differs depending on whether computation or bandwidth is the limitation for the given accelerator. The computation of the sampling positions also incurs a fixed cost which isn't affected by the choice of float or double precision.

If considering double precision, one has to be aware that the difference between orders is much higher than the difference between float and double precision. Thus if high precision interpolation is required, increasing the order may be a better choice than using double precision.

4.2 Execution Time

Execution time is of prime importance in many applications, thus in this section we discuss the impact of the B-spline order on the running time. Table 8 shows the float and double precision running times for all supported orders on the compared accelerators (Table 1), while Tables 6 and 7 show where time is spent for orders 3 and 11. The first accelerator was a 4 core CPU, while the other accelerators were GPUs with increasing performance levels.

From these tables, one can deduce that there is a significant runtime advantage to using low orders, such as order 3. While the prefiltering cost increases linearly with the number of causal and anticausal passes (1 full 2D pass is added every two orders), the cost of the interpolation increases very significantly. While the prefiltering cost (if we integrate the transposition cost) is predominant over the interpolation cost at order 3, the reverse is true for most configurations tested for order 11. The impact of the interpolation order differs depending on the accelerator, and for example accelerator 1 has surprisingly bad performance for order 11 when using double precision. It illustrates the challenge

of having a single code run well on all accelerators.

As for orders 0 and 1, while supported in our code, they are not optimized to use the direct hardware support for nearest neighbor and bilinear interpolation. We recommend using hardware support if possible.

Accelerator	Configuration\ Pass	Prefilters	Tranposes	Interpolation	Total
0	all float	468.88	239.80	640.20	1348.88
	all double	781.91	274.82	724.10	1780.82
1	all float	64.19	30.53	69.73	164.44
	all double	123.60	60.48	147.43	331.52
2	all float	8.05	4.23	4.43	16.71
	all double	16.65	8.71	20.39	45.75
3	all float	2.98	1.63	1.38	5.99
	all double	5.75	2.83	1.85	10.44

Table 6: Running time of B-spline interpolation order 3 on the compared accelerators (in ms). The figures were obtained by averaging 100 runs on each accelerator in profiling mode.

Accelerator	Configuration\ Pass	Prefilters	Tranposes	Interpolation	Total
0	all float	2379.62	242.38	4248.79	6870.79
	all double	4121.44	267.04	4281.65	8670.13
1	all float	325.46	30.53	641.27	997.26
	all double	628.51	60.42	4787.90	5476.83
2	all float	40.85	4.24	79.19	124.27
	all double	84.36	8.69	269.09	362.15
3	all float	15.01	1.62	20.40	37.04
	all double	28.92	2.84	20.95	52.71

Table 7: Running time of B-spline interpolation order 11 on the compared accelerators (in ms). The figures were obtained by averaging 100 runs on each accelerator in profiling mode.

Accelerator	Configuration\ order	0	1	2	3	4	5	6	7	8	9	10	11
0	all float	603.34	397.33	1344.35	1332.13	2071.00	2186.50	3050.61	3281.27	4365.03	5032.56	6414.90	6875.39
	all double	637.68	463.04	1743.42	1824.25	2917.09	3068.17	4323.99	4532.20	5863.52	6681.37	8313.24	8618.64
1	all float	73.95	28.18	183.91	164.58	304.25	302.48	457.88	467.53	641.50	674.41	864.81	998.24
	all double	102.26	52.42	320.59	331.86	553.28	619.25	863.72	980.61	1316.83	1435.78	1820.08	5485.54
2	all float	2.94	2.25	16.85	16.76	28.62	29.68	45.50	45.73	76.15	80.79	112.71	124.74
	all double	6.59	5.62	40.18	45.89	74.61	91.05	143.01	161.62	221.77	250.35	329.64	361.19
3	all float	1.37	1.37	5.96	5.99	9.10	9.91	14.23	15.93	20.72	24.53	32.91	37.03
	all double	1.38	1.38	9.90	10.44	16.90	18.26	25.73	27.21	37.31	43.65	48.98	52.71

Table 8: Running time of B-spline interpolation for all supported orders on the compared accelerators (in ms). The figures were obtained by averaging 100 runs on each accelerator in profiling mode.

5 Conclusion

In this work, we presented an optimized GPU implementation of the image B-spline interpolation method of [4]. The two main optimizations consists in: (1) transposing the B-spline coefficients

before the prefiltering of the rows, and (2) dividing columns into subregions in order to use more threads.

In an experimental section we compared the use of float or double precision for the different steps and found that using double precision increased accuracy only if all the steps and storage use double precision. Float precision met the maximum error target for targets up to 1e-5, and lower errors required double precision. We compared the speed for both precisions and for orders ranging from 0 to 11 on several accelerators (a 4 core CPU, and 3 GPUs). Using GPUs enabled significantly better performance. For example on two of the compared GPUs, an interpolation of order 11 for the 16Mpx reference image took less than one second for both precision modes. Using double precision was generally about twice slower for the compared GPUs than using float precision, due to the doubling of the bandwidth needs.

While our implementation contains generic optimizations, applications with particular needs could get further speed boosts by implementing specific optimizations, several of which we mentioned in this article.

Acknowledgements

The authors would like to thank Prof. Jean-Michel Morel for his support, suggestions, and many fruitful discussions.

Work partly financed by Office of Naval research grant N00014-17-1-2552, DGA Astrid project "filmer la Terre" nº ANR-17-ASTR-0013-01, MENRT and Fondation Mathématique Jacques Hadamard.

Image Credits

Provided by the authors

References

[1] A. Aldroubi, M. Unser, and M. Eden, *Cardinal Spline Filters: Stability and Convergence to the Ideal Sinc Interpolator*, Signal Processing, 28 (1992), pp. 127–138. http://dx.doi.org/10.1016/0165-1684(92)90030-Z.

[2] AMD, *AMD APP SDK OpenCL^{TM} Optimization Guide*, (2015).

[3] G. Bradski, *The OpenCV Library*, Dr. Dobb's Journal of Software Tools, (2000).

[4] T. Briand and P. Monasse, *Theory and Practice of Image B-spline Interpolation*, Image Processing On Line, 8 (2018), pp. 99–141. https://doi.org/10.5201/ipol.2018.221.

[5] F. Champagnat and Y. Le Sant, *Efficient cubic B-spline image interpolation on a GPU*, Journal of Graphics Tools, 16 (2012), pp. 218–232.

[6] P. Getreuer, *Linear Methods for Image Interpolation*, Image Processing On Line, 1 (2011). http://dx.doi.org/10.5201/ipol.2011.g_lmii.

[7] H. Hou and H. Andrews, *Cubic Splines for Image Interpolation and Digital Filtering*, IEEE Transactions on Acoustics, Speech and Signal Processing, 26 (1978), pp. 508–517. http://dx.doi.org/10.1109/TASSP.1978.1163154.

[8] S. Junkins, *The compute architecture of Intel® processor graphics gen9*, (2015).

[9] M. Liou, *Spline fit made easy*, IEEE Transactions on Computers, 100 (1976), pp. 522–527. http://dx.doi.org/10.1109/TC.1976.1674640.

[10] NVIDIA, *NVIDIA OpenCL Best Practices Guide*, (2009).

[11] D. Ruijters and P. Thévenaz, *GPU prefilter for accurate cubic B-spline interpolation*, The Computer Journal, 55 (2012), pp. 15–20.

[12] I.J. Schoenberg, *Contributions to the Problem of Approximation of Equidistant Data by Analytic Functions*, in I.J Schoenberg Selected Papers, Springer, 1988, pp. 3–57. http://dx.doi.org/10.1007/978-1-4899-0433-1_1.

[13] D. Shamonin and M. Staring, *An OpenCL implementation of the Gaussian pyramid and the resampler*, (2012).

[14] P. Thévenaz, T. Blu, and M. Unser, *Interpolation Revisited [Medical Images Application]*, IEEE Transactions on Medical Imaging, 19 (2000), pp. 739–758. http://dx.doi.org/10.1109/42.875199.

[15] M. Unser, *Splines: A Perfect Fit for Signal and Image Processing*, IEEE Signal Processing Magazine, 16 (1999), pp. 22–38. http://dx.doi.org/10.1109/79.799930.

[16] M. Unser, A. Aldroubi, and M. Eden, *Fast B-spline Transforms for Continuous Image Representation and Interpolation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 13 (1991), pp. 277–285. http://dx.doi.org/10.1109/34.75515.

[17] ——, *B-spline Signal Processing. I. Theory*, IEEE Transactions on Signal Processing, 41 (1993), pp. 821–833. http://dx.doi.org/10.1109/78.193220.

[18] ——, *B-spline Signal Processing. II. Efficiency Design and Applications*, IEEE Transactions on Signal Processing, 41 (1993), pp. 834–848. http://dx.doi.org/10.1109/78.193221.