A Comparison of Sequential and GPU-Accelerated Implementations of B-Spline Signal Processing Operations for 2-D and 3-D Images

Alex Karantza, Sonia Lopez Alarcon and Nathan D. Cahill

1 Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY USA, 14623
2 Center for Applied and Computational Mathematics, School of Mathematical Sciences, Rochester Institute of Technology, Rochester, NY USA, 14623

Abstract—B-spline signal processing operations are widely used in the analysis of two and three-dimensional images. In this paper, we investigate and compare some of these basic operations (direct transformations, indirect transformations, and computation of partial derivatives) by (1) recursive filter based implementations in MATLAB and C++, and (2) GPU-accelerated implementations in CUDA. All operations are compared at a variety of resolution levels on a 2-D panoramic image as well as a 3-D magnetic resonance (MR) image. Results indicate significant improvements in efficiency for the CUDA implementations.

A MATLAB toolkit implementing the various B-spline signal processing tasks as well as the C++ and CUDA implementation described here is currently publicly available.

Keywords—B-spline, filtering, interpolation, GPU

I. INTRODUCTION

Two of the most common tasks in numerical analysis are the interpolation of data and the approximation of smooth curves from noisy data. Splines, or piecewise polynomial functions with a prescribed degree of smoothness, provide a great amount of flexibility for these tasks, given their local behavior, accuracy, and ease of evaluation. B-splines are splines that have minimal support given their degree, and they can be computed in a numerically stable manner [1]. Since a thorough account of the theory and implementation of B-spline based algorithms for signal processing was published in [2], [3], B-splines have been applied in various image processing and analysis algorithms, including image registration [4], contour detection [5], image reconstruction [6], superresolution [7], and motion estimation [8].

Most image processing applications that utilize B-splines require performing one or more of the following fundamental tasks: direct transformation, indirect transformation, interpolation, and evaluation of partial derivatives. Interpolation and evaluation of partial derivatives can be implemented using analytical formulas once a direct transformation is performed; direct and indirect transformations are implemented as symmetric infinite impulse response (IIR) and finite impulse response (FIR) filters, respectively. Prior to the advent of massively parallel computer architectures, the most efficient way to perform direct B-spline transformation was by recursive filtering [3].

More recently, however, various GPU-based strategies have been proposed [9], [10], [11] to significantly improve the efficiency of the direct B-spline transformation. These existing implementations focus on cubic interpolation, mostly on 2-dimensional images. This process involves performing the prefiltering step (the direct transform) and evaluating the calculated B-splines at different locations (interpolation). This cubic interpolation process is useful and can be cleanly represented in a single CUDA program [11], but due to their direct implementation they cannot be directly applied to many of the more challenging or unique applications of B-spline signal processing. Also, processing the data on the GPU introduces additional limitations on the size of the input data, which none of these existing implementations attempt to overcome.

To increase the applicability and accessibility of these GPU implementations, our approach separates the individual transforms and allows the user to execute them independently through a MATLAB interface. We will show how CUDA-based implementations of direct and indirect transformations, interpolation, and evaluation of partial derivatives can provide significant improvements in computational efficiency over CPU-based implementations in two examples of large images: a high resolution 2-D panoramic image created by a GigaPan® EPIC system [12], and a 3-D full-body magnetic resonance (MR) image.

The remainder of this paper is organized in the following manner. Section II provides the theoretical background of B-splines and the fundamental tasks of direct and indirect transformations, interpolation, and evaluation of partial derivatives. Section III describes efficient GPU-based implementations of these fundamental B-spline signal processing tasks, with special care given to their application to large images. Section IV describes a set of experiments for comparing GPU versus CPU implementations of these tasks on 2-D panoramic and 3-D MR images, and section V presents the results of these experiments. Finally, Section VI lists conclusions and ideas for future work.
II. B-SPLINE SIGNAL PROCESSING

Using the notation of [2], [3], a B-spline $\beta^n(x)$ of order $n$ is defined by:

$$\beta^n(x) := \sum_{j=0}^{n+1} \frac{(-1)^j}{n!} \binom{n+1}{j} \left(x + \frac{n+1}{2} - j\right)^n \mu \left(x + \frac{n+1}{2} - j\right),$$

where

$$\mu(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}.$$  \hspace{1cm} (1)

As shown in Fig. 1, each $\beta^n(x)$ is a piecewise polynomial of order $n$ that is nonzero on $\left(-\frac{n+1}{2}, \frac{n+1}{2}\right)$ and has $(n-1)$ continuous derivatives.

Discrete B-splines are defined by sampling continuous B-splines; i.e., $b^n_m(k) := \beta^n(k/m)$, where $m$ is an integer expansion factor. (Shifted versions of discrete B-splines are also useful to define. We refer the interested reader to [2], [3]). Discrete B-splines can be used to define a one-to-one correspondence between a discrete signal $\{y(k)\}$ and its sequence $\{g(k)\}$ of B-spline coefficients of order $n$.

If it is possible to efficiently transform from $\{g(k)\}$ to $\{y(k)\}$ and back, other signal processing operations such as interpolation, decimation, and computation of partial derivatives can be performed using analytical properties of B-splines. In 2-dimensional or 3-dimensional image processing applications, this is especially useful because these transformations and computations can be performed in a separable manner.

A. Direct Transformation

Given a discrete signal $\{g(k)\}$, its corresponding sequence $\{y(k)\}$ of B-spline coefficients can be computed by a linear filtering operation known as the direct B-spline transformation:

$$y(k) = (b^n_1)^{-1} * g(k),$$  \hspace{1cm} (3)

where $*$ indicates convolution and $(b^n_1)^{-1}$ is the impulse response (the solution to (3) when $y(k) = \delta(k)$ and $g(k) = b^n_1(k)$). As described in [3], $(b^n_1)^{-1}$ is a symmetric filter having infinite impulse response (IIR), allowing the direct transformation to be implemented efficiently in a CPU by recursive filtering in the $z$ transform domain.

B. Indirect Transformation

The original discrete signal $\{g(k)\}$ can be reconstructed from its sequence $\{y(k)\}$ of B-spline coefficients by the indirect B-spline transformation:

$$g(k) = b^n_1 * y(k),$$  \hspace{1cm} (4)

where the discrete B-spline $b^n_1$ is a symmetric filter having finite impulse response (FIR).

C. Partial Derivatives

One task that is regularly performed in 2-D and 3-D image analysis is computing the Laplacian of the image, or the sum of the second partial derivatives of the image with respect to each independent variable. This task can be done separately along every row, column (and slice) of an image if its B-spline coefficients are known.

From property (2.8) in [2], we find that:

$$\frac{\partial^2 \beta^n(x)}{\partial x^2} = \beta^{n-2}(x+1) - 2\beta^{n-2}(x) + \beta^{n-2}(x-1).$$  \hspace{1cm} (5)

This implies that if a discrete signal $\{g(k)\}$ has B-spline coefficients $\{y(k)\}$, the B-spline coefficients corresponding to the second derivative of the signal can be obtained by convolution of $\{y(k)\}$ with the finite difference operator $[1 \ -2 \ 1]$. (Note that the resulting B-spline coefficients are for a B-spline of order $n = 2$.)

Hence, if the B-spline coefficients of a signal are not already known, the second derivative of a discrete signal $\{g(k)\}$ can be computed by:

$$g^{(2)}(k) = b^{n-2}_1 * [1 \ -2 \ 1] * (b^n_1)^{-1} * g(k).$$  \hspace{1cm} (6)

III. GPU IMPLEMENTATION

In this section, we describe how each type of B-spline signal processing operation is efficiently implemented in GPU. The implementations on the GPU are paired with standard C++ implementations and MATLAB implementations for comparison and validation. The C++ and CUDA implementations are exposed through MEX (MATLAB Executable) functions and wrapped in a MATLAB class that exposes a comparable interface to the pure MATLAB version. This makes the accelerated algorithms more accessible and easy to use.

A. Direct Transformation

The direct transform is a symmetric infinite impulse response filter responsible for transforming a discrete signal of intensity values into coefficients for a B-spline of a given
degree. The impulse response of the filter is an inverse exponential, and decays very rapidly as it gets farther from zero. This allows the filter to be well approximated by a convolution with a finite signal. It has been shown by Champagnat and Le Sant [9] that very few coefficients are necessary to retain a very accurate approximation. Our implementation chooses to retain 15 coefficients as a balance between accuracy and performance, though smaller kernels may still be adequate depending on the needs of the application.

A set of CUDA kernels were written to implement 1, 2, and 3D convolution, assuming a separable and symmetric kernel. These are all properties of the filter that allow for a more efficient program. This kernel is calculated on the CPU using predetermined coefficients for B-splines of degrees 1 through 7 by performing the efficient two-pass filtering step introduced in [3] on an impulse signal. These 8 values representing half of the symmetric kernel are then uploaded to the GPU’s constant memory. The image or volume to be transformed is uploaded into the GPU’s global memory, and the kernels are run to transform the data along the necessary dimensions. The GPU’s read-only texture memory was used to improve the performance of these kernels, since each pixel (voxel) will be read 15 times during each convolution. Each thread computes a single output pixel.

Once the data has been transformed into coefficients, the array is retained in the GPU’s global memory. The GPU implementations of other algorithms may then use this data in place rather than being required to re-upload the coefficients. The data is only copied back down when specifically requested.

B. Indirect Transformation

The indirect transform is the counterpart to the direct transform; this operation transforms an array of B-spline coefficients into intensity data. Like the direct transform, it is represented by a finite convolution. The kernel used can be thought of as the coefficients to the polynomial needed to evaluate the B-spline at a given location.

This convolution is implemented in CUDA using constant memory once again to store the kernel, and shared memory to accelerate the reads from global memory. Shared memory is much faster to access than global memory, but is local to a single block of threads. The individual threads copy data from global memory to populate a local shared array with all the values needed to perform the convolution of the given size. Once the threads have completed their copy, they perform the convolution as normal and save the result back to global memory. Each thread computes at most 8 output pixels, and reads in multiple pixels in a striped fashion to help avoid bank conflicts [13]. The indirect transform copies the resulting array back to host memory afterwards.

C. Partial Derivatives

Finding partial derivatives (or the Laplacian) is once again implemented as a simple finite impulse response filter. The B-spline coefficients (already in GPU memory) are convolved against a simple differentiation kernel ([1, −1] for the 1st derivative, [−1, 2, −1] for the 2nd) along a chosen dimension. Since each pixel is only read 2 or 3 times (for the first and second derivative, respectively) using a construct such as the texture cache or shared memory would have a less pronounced effect on the performance, and so these kernels read from global memory directly for simplicity.

In practical situations when the B-spline coefficients for a signal/image may not have been computed in advance, the process of computing partial derivatives must include a direct transformation to determine B-spline coefficients, convolution to perform the differentiation, and an indirect transformation to compute the new signal/image. Figure 2 illustrates the architecture used to perform this type of task. The original image signal, along with the parameters for the B-spline filtering, are set in MATLAB. The direct filter, partial derivative filters, and indirect filter are then run (either entirely in CUDA or C++) to produce the desired result.

IV. EXPERIMENTS

In this section, we perform a series of experiments to assess the performance of GPU implementations of B-spline signal processing operations on two different types of images: (1) a 2-D panoramic image, and (2) a 3-D magnetic resonance (MR) full body scan.

The 2-D panoramic image is a 14914 × 61028 pixel image comprising a 360° view of the Devil’s Golf Course in Death Valley, California, USA. It was captured and stitched with a GigaPan® EPIC system [12], and is shown in Fig. 5. For the purposes of this paper, the image was converted from RGB to grayscale. The 3-D MR full body scan is a 400 × 300 × 151 voxel volumetric image. Orthogonal slices through the volumetric images are illustrated in Fig. 3.

Both the panoramic image and the MR image volume were downsampled to multiple smaller resolutions to provide
Fig. 3. Slices of a three-dimensional magnetic resonance (MR) full body scan.

additional data points. For the panoramic image, the resolution levels were as follows: L0 (61028 × 14941), L1 (30514 × 7471), L2 (15257 × 3736), L3 (7629 × 1868), L4 (3815 × 934), and L5 (1908 × 467). For the MR image, the resolution levels were: L0 (400 × 300 × 151), L1 (201 × 151 × 76), L2 (101 × 76 × 39), and L3 (51 × 39 × 20). The time taken by each algorithm scales by the size of the image and the number of dimensions; the content of the images does not affect the performance.

All experiments were performed on a Windows 7 (64 bit) computer with an Intel Core i7-2600 CPU at 3.40GHz and 16GB of RAM. The GPU used was an NVIDIA Tesla C2070 with 4GB of global memory at 1494 MHz and a core clock of 1.147 GHz.

A. Direct Filter

The direct filter calculates the B-spline coefficients of the input for a given order of B-spline. For these tests, a cubic (third-order) spline was used, since this is a very common choice for image processing applications. The MATLAB and C++ code use the two-pass infinite impulse response (IIR) filter, while the CUDA implementation uses the finite impulse response (FIR) approximation. MATLAB processes in double precision, while the C++ and CUDA implementations work in single precision. The standard error between the MATLAB and each compiled implementation is calculated to verify that the results are equivalent.

B. Indirect Filter

The indirect filter is used to evaluate the B-Spline coefficients at each pixel and recover the original image. The coefficients calculated during the test of the direct filter are used here once again to evaluate the MATLAB, C++, and CUDA implementations of indirect filtering.

C. Laplacian with Known B-Spline Coefficients

Calculating the Laplacian is a common operation in many image processing algorithms and an excellent test of using B-spline coefficients to calculate the partial derivatives of a signal. Like the indirect filter, this is implemented as a finite convolution in all platforms. The C++ and CUDA implementations use the 3-element kernel to compute the second partial derivative in one pass, while the MATLAB implementation uses two passes of a 2-element kernel implementing the first derivative. Due to this discrepancy, the results are offset by two pixels. Before verifying the implementations, the MATLAB results are shifted appropriately.

D. Laplacian with Unknown B-Spline Coefficients

A final test of the implementations was done that combines all of these operations into a single example that would better reflect the real-world use of such a library. The direct filter is run on the original image data, the second partial derivative along each dimension is calculated from the resulting B-spline coefficients, and then the indirect filter is run on those results and summed to find the Laplacian of the original image. This whole process is timed from beginning to end, and so includes all the overhead of GPU memory operations when they occur and the overhead of copying the data to and from MATLAB for both the C++ and CUDA implementations.

V. RESULTS

Figure 4 shows the timing results of direct filtering, indirect filtering, Laplacian computation with known B-spline coefficients, and Laplacian computation with unknown B-spline coefficients as implemented in MATLAB, C++, and CUDA. The circles in each plot represent the resolution level at which the images were processed. For the 2-D GigaPan image, these points represent levels L5, L4, L3, and L2; for the 3-D MR image, these points represent levels L3, L2, L1, and L0. (The levels L1 and L0 of the GigaPan image required too much memory for our implementations to run successfully.)

The direct filter as implemented in C++ is nearly as fast as the MATLAB version for 3D volumes, and even slower than MATLAB on the 2D samples (worst case of 0.84 relative performance, best of 18.80), but the CUDA version was significantly faster for all test cases (worst case of 1.71, best case of 30.13 times faster).

The C++ implementation of indirect filtering is clearly inferior to the MATLAB implementation (which uses the builtin function convn). However once again, in all cases the CUDA implementation performs better than the MATLAB version with a worst-case speedup of 1.70 and a best case of 13.17.

For computing the Laplacian when B-spline coefficients are known, the MATLAB and C++ implementations are similar in terms of performance, with C++ only narrowly improving on MATLAB’s convolution. The CUDA implementation is at worst 12.16 faster than the MATLAB version, and on the
Fig. 4. Results of timing of various B-spline operations as implemented in MATLAB, C++, and CUDA and applied to different levels of the 2D and 3D images.
largest image achieved a 70.62 relative performance. This is likely due to the very few reads from memory and the simple coefficients known at compile-time.

For the more practical scenario of computing a Laplacian given the original image data (which requires performing direct and indirect transformations), the C++ implementation of this practical scenario is much slower than the MATLAB version. This can be explained by the poor performance of the C++ function for indirect filtering (which is called two or three times in this test, depending on the dimensionality of the data). The CUDA implementation is once again consistently faster than all other implementations with a best-case relative performance of 8.36 (and worst-case of 2.25), demonstrating that this library could indeed provide a practical benefit to users working with these algorithms.

VI. CONCLUSION

In this paper, we provided a comparison of various B-spline signal processing operations as implemented sequentially (in MATLAB and C++) versus in parallel (in CUDA). The comparison focused on the tasks of direct and indirect transformations and the computation of the Laplacian of two and three-dimensional images. Results show the superiority of the CUDA implementation in all cases.

APPENDIX

The MATLAB class bsarray, implementing various B-spline signal processing operations, is available for download at MATLAB Central (http://www.mathworks.com/matlabcentral/), under File ID #19632. The CUDA implementation is available for download from the website of the RIT High-Performance Hardware & Software Solutions Research Group (http://hiper.ce.rit.edu/).

ACKNOWLEDGMENT

The GigaPan® image was provided by Brandon May of the Center for Imaging Science at Rochester Institute of Technology and was captured with the support of National Science Foundation grant #0909588.

REFERENCES
