Rochester Institute of Technology
RIT Scholar Works

Theses | Thesis/Dissertation Collections

5-1-2012

Scalable GPU acceleration of b-spline signal processing operations

Alexander Karantza

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Karantza, Alexander, "Scalable GPU acceleration of b-spline signal processing operations" (2012). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Scalable GPU Acceleration of B-Spline Signal Processing Operations

by

Alexander Karantza

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by

Assistant Professor Dr. Sonia Lopez Alarcon
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2012

Approved by:

Dr. Sonia Lopez Alarcon, Assistant Professor
Thesis Advisor, Department of Computer Engineering

Dr. Nathan D. Cahill, Associate Professor
Committee Member, Department of School of Mathematical Sciences

---

Dr. Andres Kwasinski, Assistant Professor
Committee Member, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Scalable GPU Acceleration of B-Spline Signal Processing Operations

I, Alexander Karantza, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Alexander Karantza

Date

Contents

Abstract . . . . **viii**

1 Thesis Overview **1**

2 Background **4**

2.1 Signal Processing with B-Splines 4

2.1.1 Direct Filter 8

2.1.2 Indirect Filter 9

2.1.3 Interpolation 9

2.1.4 Partial Derivatives 10

2.2 Applications of B-Spline Processing 11

3 Existing Solutions **13**

3.1 Unser *et al.* 14

3.2 Ruijters and Thévenaz 14

3.3 Champagnat and Le Sant 15

4 Implementation **17**

4.1 Overview of GPU Architecture and CUDA 17

4.2 Overview of Library Architecture 18

4.3 Details of Parallel Functions 20

4.3.1 Direct Transform 20

4.3.2 Indirect Transform 22

4.3.3 Interpolation 23

4.3.4 Laplacian 24
4.4 Hardware Considerations and Optimization 25

5 Results 26
5.1 Sample Datasets 26
5.2 Test Environment 27
5.3 Performance Results 28
5.3.1 Analysis and Scalability 29

6 Conclusion and Future Work 32

Bibliography 34

7 Appendix 36
7.1 Acknowledgment 36

List of Tables

5.1 Average execution time in seconds for 2D image. Missing data is due to memory limitations. 28
5.2 Speedups of CUDA over MATLAB for 2D image. 29
5.3 Average execution time in seconds for 3D volume. 29
5.4 Speedups of CUDA over MATLAB for 3D volume. 30

List of Figures

2.1 B-Spline basis functions for orders 0 through 5. [7] 6
2.2 A random signal along with the coefficients for a spline that interpolates it. The coefficients are on the same order of magnitude as the original signal with the same spacing, making storage simple. 6
2.3 Illustrating B-Splines of different orders interpolating a random 2D signal. The original discrete signal consists of 25 samples at the integer coordinates, while the continuous spline function can interpolate for all real coordinates in between. Higher order splines result in smoother interpolation, from nearest-neighbor to linear to quadratic to cubic. (The colors illustrate intensity, not separate channels.) 7
2.4 Splines of various degree and smoothing interpolating a random 1-D signal. 9
2.5 The first and second derivative of the spline representation of arctangent. 10

5.1 GigaPan of Devil's Golf Course, Death Valley, USA. A grayscale version was used for testing. 26
5.2 Slices of a three-dimensional magnetic resonance (MR) full body scan. [7] 27
5.3 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. [7] 31

Abstract

Scalable GPU Acceleration of
B-Spline Signal Processing Operations

Alexander Karantza

Supervising Professor: Dr. Sonia Lopez Alarcon

B-Splines are a useful tool in signal processing, and are widely used in the analysis of two and three-dimensional images. B-Splines provide a continuous representation of the signal, image, or volume, which is useful for interpolation, resampling, noise removal, and differentiation - all important steps in many signal processing algorithms. These splines are defined entirely by an array of coefficients that is roughly the same size as the original signal and of values in the same order of magnitude, making storage and representation trivial.

What is not trivial, however, is the quick calculation and processing of those coefficients, especially for very large data. As technology improves in fields such as medical imaging, algorithms that use B-Splines will need to process increasingly higher resolution images and voxel volumes. New implementations are needed to make use of modern parallel architectures to keep these algorithms practical.

This thesis presents a library for performing many common B-Spline operations in CUDA, the parallel programming framework for NVIDIA GPUs, and analyzes the considerations necessary when implementing a large-scale parallel version of such a well-established sequential algorithm. This library is meant to be used both by C++ programs as well as algorithms implemented in MATLAB without requiring significant changes.

Significant speedups are obtained using this library to perform various common B-Spline image processing operations (as much as 30x for some), and the scalability limitations of the GPU implementation are addressed.

Chapter 1

Thesis Overview

In many signal processing and numerical analysis algorithms, it is necessary to interpolate existing data or to find smooth curves that approximate this data. Often, operations are best defined on continuous functions but the data of interest is discretized as a result of the method of acquisition or storage. This discrete data may also contain noise that a continuous representation should not include. A popular approach to finding this continuous representation is through the use of splines.

A spline is a piecewise connection of polynomial functions. These functions connect at points called *knots* with a smoothness defined by the degree of the spline. B-Splines are a special case of spline function that has minimal support and can be computed in a numerically stable way [6]. The ability to define splines that can interpolate and smooth over discrete functions with only a few parameters has encouraged their use in many applications of signal processing, including image registration [11], contour detection [4], image reconstruction [2], superresolution [3], and optical flow [8].

These algorithms all make use of the continuous nature of B-Splines. For any B-Spline of a given degree, it is straightforward to compute the derivative as a B-Spline of a lesser degree, and to interpolate the original signal by evaluating the B-Spline function at points between the knots. To arrive at a continuous representation for a signal (which could be an image, video, voxel volume, etc) from the discrete source data is known as the cardinal spline interpolation problem. In the case of a uniform spacing, the discrete data can be transformed into the coefficients for a B-Spline of a given degree and smoothness by means of a linear operation called the *direct transform*.

The coefficient signal is the same size as the original signal, and the coefficients themselves are of the same order of magnitude as the original data [6]. From these coefficients, it is possible to compute analytic operations such as finding the laplacian of the B-Spline with additional linear operations on the coefficients. Such operations result in a coefficient signal to a new B-Spline. To evaluate these splines and arrive back at an intensity signal, the inverse of the direct transform (called the *indirect transform*) can be applied [15].

The theoretical basis and initial approaches to implementation were discussed by Unser *et al.* in [15] and [16] in 1993, and have since served as the seminal work on B-Spline signal processing. The efficient algorithms for the direct and indirect filters, as well as differentiation and smoothing, have been implemented in many applications, but these algorithms do not consider modern parallel architectures.

More recently, research has been done to implement signal processing algorithms on graphics processing units (GPUs) [13, 5]. For these algorithms that use B-Splines parallel implementations of these filtering steps (or at least a subset) have been needed. The transforms have a great deal of parallelization available, and GPU implementations show great performance benefits, but due to the relative complexity and recent availability of GPUs as a practical computing platform these implementations are often application-specific and inflexible.

Specifically, most of the research in using B-Splines is focused on their use in cubic image interpolation. Hardware support for this common operation is notably lacking, so efficient implementations are in high demand for graphics applications. While some great progress has been made in this area, the scope is limited and not directly applicable to researchers who are using B-Splines in a more abstract or compute-intensive way. These implementations rarely address large-scale data, instead focusing on conventional two-dimensional images or simple three-dimensional voxel volumes. Additionally, by focusing on a single algorithm, these implementations do not tend to expose the B-Spline coefficients in their implementation in an accessible way, which is necessary if any advanced operations are to chained together for more complicated algorithms.

The product of this thesis is a library for flexible and reliable GPU acceleration of these algorithms that would be valuable to researchers working with B-Splines in signal processing applications. This library can be used both through a MATLAB interface or directly through binary linking to take advantage of the performance gains of the GPU implementation when working in either a prototyping environment or in a more traditional program. Since researchers are hopefully working on novel algorithms and new solutions to problems in a wide variety of fields, the flexibility of the library is paramount.

In addition to measuring the performance of this new library in comparison to an existing implementation, the metric of scalability is assessed. Large datasets are becoming increasingly common, especially in the fields of medical imaging, and so the applicability of various approaches to these large images is an important consideration when developing practical tools. Aspects of the hardware and implementation approaches place limits on the size of data that it is possible to handle as well as impacting the performance of the solution.

By using the latest approaches to parallelizing these algorithms, the library consistently outperforms the existing implementation. The concerns about scalability were well-founded, and modifications to the implementation are proposed to alleviate this shortcoming. The finished library will be made available for reference and public use.

Chapter 2

Background

2.1 Signal Processing with B-Splines

Splines are used in many common image processing techniques whenever a continuous representation of a signal is desired, such as downsampling, interpolation, as well as registration and deformation. Having such a continuous representation allows for smoother interpoloation, as well as the ability to evaluate the analytic derivatives of the signal with a guarantee of continuity. Splines are piecewise polynomial functions described by a finite sequence of coefficients and parameters of degree and smoothness. B-Splines are a special kind of spline that has minimal support with coefficients and can be computed in a numerically stable manner [6]. Unser *et al.* first demonstrated the theory, implementation, and application of B-Splines to signal processing in [15, 16], which has become the seminal work and basis for many implementations.

Before discussing how B-Splines are used, it is important to have an understanding of the mathematical definition of these functions. Given a discrete signal $g(k)$, we wish to describe it as a continuous polynomial spline function $g(x)$. This continuous function can be constructed from a weighted sum of shifted B-Splines of order n:

$$g^n(x) = \sum_{k=-\infty}^{+\infty} y(k)\beta^n(x-k) \tag{2.1}$$

Some of these B-Spline basis functions are shown in Figure 2.1, and a comparison of the data $g(k)$ with the corresponding spline coefficients $y(k)$ can

be seen in Figure 2.2.

The weighting coefficients $y(k)$ are what we seek to compute, given various parameters for the B-Spline β. These B-Spline functions are essentially approximations to the sinc interpolator. The zeroth-order B-Spline β^0 is defined as a rectangular pulse of unit area: $\mu(\frac{1}{2} - |x|)$, where μ is the unit step function. When convolved over the discrete signal $y(k)$ in equation 2.1, this results in a step function that interpolates the signal with a nearest-neighbor approach. Higher order B-Splines are described as repeated convolutions against this function:

$$\beta^n = \beta^{n-1} * \beta^0 \tag{2.2}$$

With repeated convolution, the B-Spline function approaches the sinc interpolator [14]. The most common spline used in signal processing is the cubic spline (β^3), however higher orders may be useful in some applications.

Unser *et al.* additionally derive B-Spline basis functions that do not directly interpolate the signal, but rather approximate the signal by attempting to increase smoothness. Such smoothing splines are useful if the original data is corrupted by noise. The parameter λ is a nonnegative value used to denote the contribution of the smoothing constraint. Splines with $\lambda = 0$ exactly interpolate their datapoints, while splines with a higher value for λ appear smoother as illustrated in Figure 2.4.

Splines also naturally extend to higher-dimensional signals, as the convolution can be considered separable. This is especially useful since it allows the trivial application of one-dimensional spline theory to other signals such as images, video, and voxel volumes. The use of various orders of B-Spline to interpolate an arbitrary 2D signal is demonstrated in Figure 2.3.

It should be noted that a B-Spline of order n requires $n+1$ control points to completely calculate the convolution. This implies that the array of coefficients is n elements larger in each dimension than the original signal. Several approaches are technically valid to extrapolate the signal to form these extra border points in the direct filtering step. One could consider out-of-bounds data points as being equal to 0 or some other constant, but this may cause unexpected behavior near the edges for high-order splines. A common approach, and the approach used from here on, is to mirror the source data over these borders as far as is necessary to provide data against

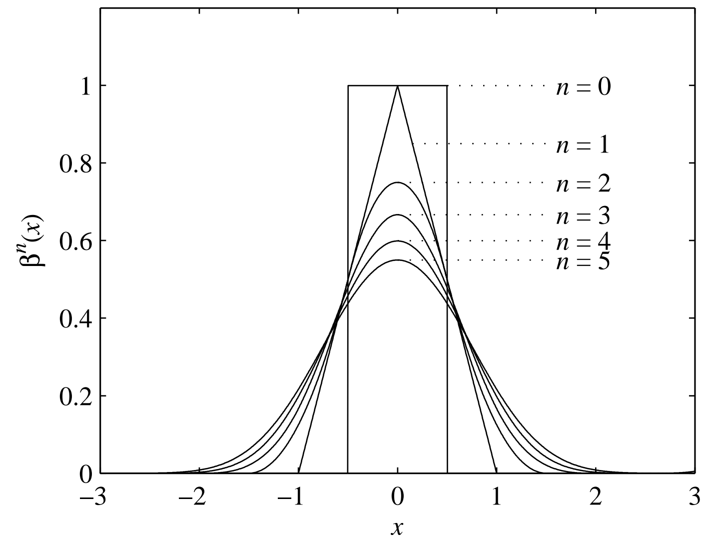

Figure 2.1: B-Spline basis functions for orders 0 through 5. [7]

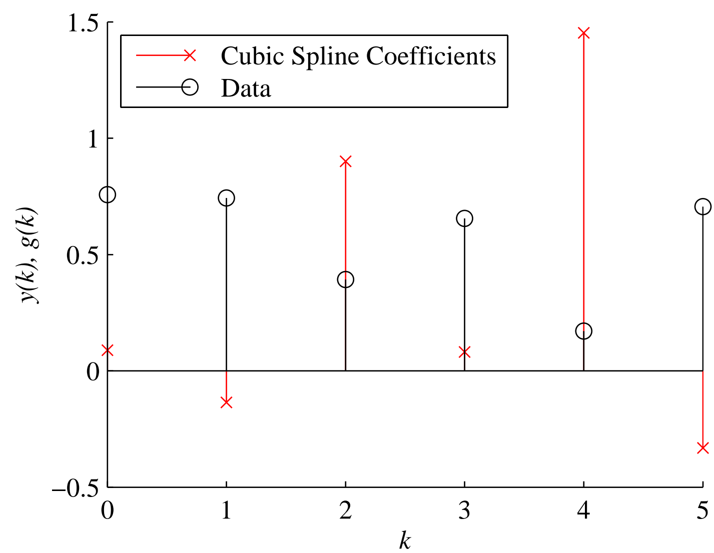

Figure 2.2: A random signal along with the coefficients for a spline that interpolates it. The coefficients are on the same order of magnitude as the original signal with the same spacing, making storage simple.

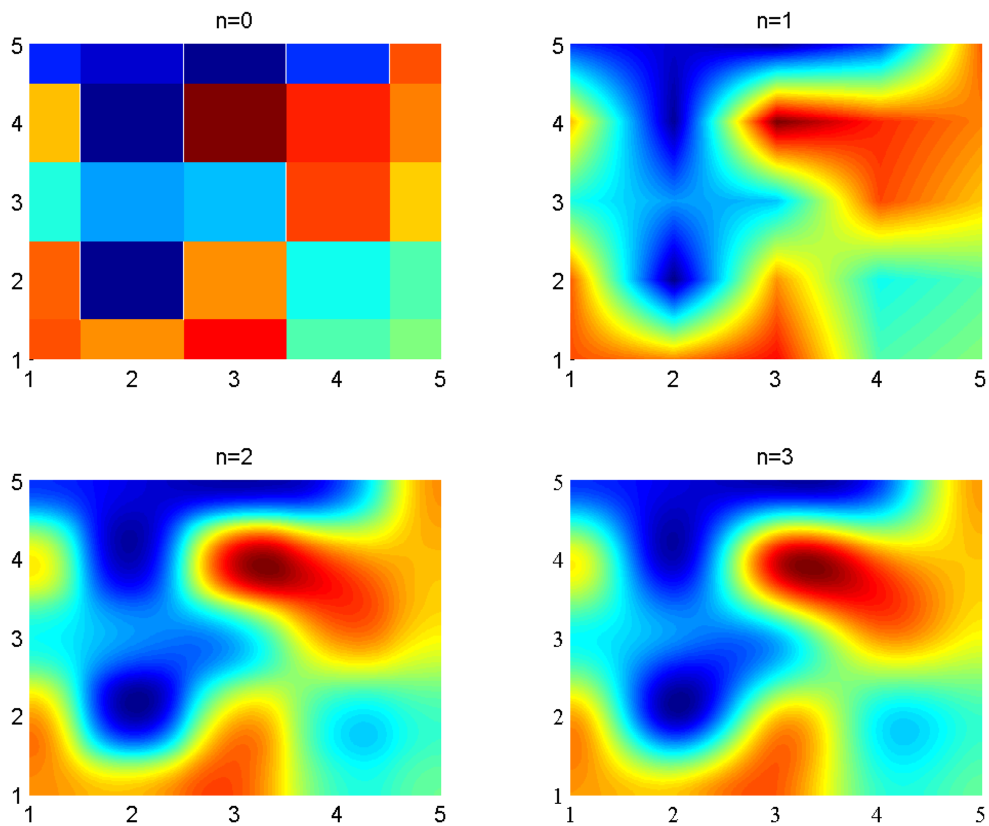

Figure 2.3: Illustrating B-Splines of different orders interpolating a random 2D signal. The original discrete signal consists of 25 samples at the integer coordinates, while the continuous spline function can interpolate for all real coordinates in between. Higher order splines result in smoother interpolation, from nearest-neighbor to linear to quadratic to cubic. (The colors illustrate intensity, not separate channels.)

which the B-Spline basis functions may be convolved to produce the correct number of coefficients[12].

2.1.1 Direct Filter

The main computational difficulty in using B-Splines for image processing is in finding the coefficients $y(k)$ given the original signal $g(k)$. The solution to this cardinal spline interpolation problem can be considered an infinite impulse response prefilter step which transforms the original signal data into a new signal with $n + 1$ additional elements in each dimension. Prefiltering ensures that the spline function interpolates the source data as desired. These filter coefficients can be found by convolving the source data against the inverse of the B-Spline function to be used [15].

$$y(k) = (\beta_1^n)^{-1} * g(k), \tag{2.3}$$

where

$$\beta_1^n(k) = \sum_{k=-\infty}^{+\infty} \delta(k)\beta^n(x-k) \tag{2.4}$$

The traditional approach to this direct filtering step involves separating the inverse of the B-Spline β_1^n into a causal and anti-causal filter in the Z domain. These filters can be written recursively in a very efficient way [16]. While the sequential approach achieves high performance by applying the two recursive filters in succession on the original signal $g(k)$, it is also possible to compute β_1^n explicitly. It has been shown that this signal exhibits rapid exponential decay, and that replacing the infinite convolution operation with a finite operation over a limited number of coefficients is enough to maintain precision in the final result [12]. This approach to implementing the direct filter as a regular separable finite convolution makes it very obviously parallel, and we can take advantage of established parallel implementations of such separable convolutions to realize this in CUDA.

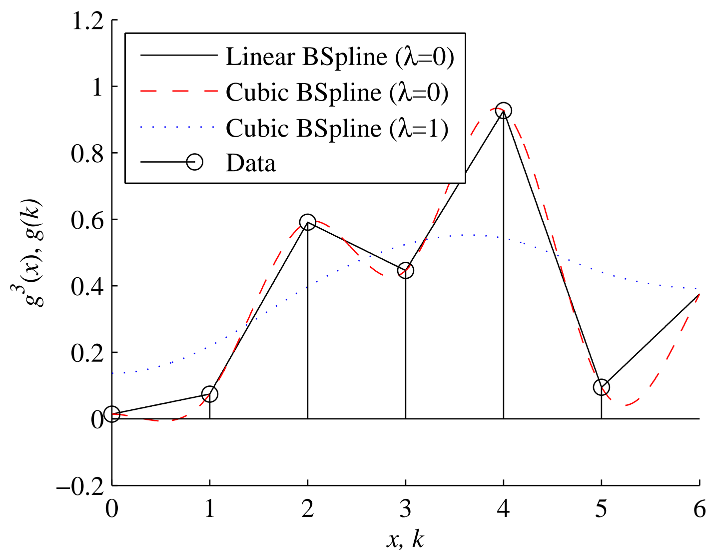

Figure 2.4: Splines of various degree and smoothing interpolating a random 1-D signal.

2.1.2 Indirect Filter

The indirect filter can be viewed as the compliment to the direct filter. The indirect filter is also a convolution that serves to evaluate the discrete B-Spline basis function at every point to recover the original signal given only coefficients. This can be represented as:

$$g(k) = \beta_1^n * y(k) \tag{2.5}$$

The B-Spline β_1^n acts as a symmetric filter with a finite impulse response with as many elements as the degree of the spline. Once again we can leverage known approaches for efficient evaluation of finite convolution to evaluate this filter and convert B-Spline representations back to intensity signals. The convolution is equivalent to evaluating the sum of the B-Spline basis functions at each knot.

2.1.3 Interpolation

Another useful property of the B-Spline representation is that, as a continuous function, it can be evaluated at any point using only the n surrounding coefficients (for a spline of order n). Evaluating the spline representation of

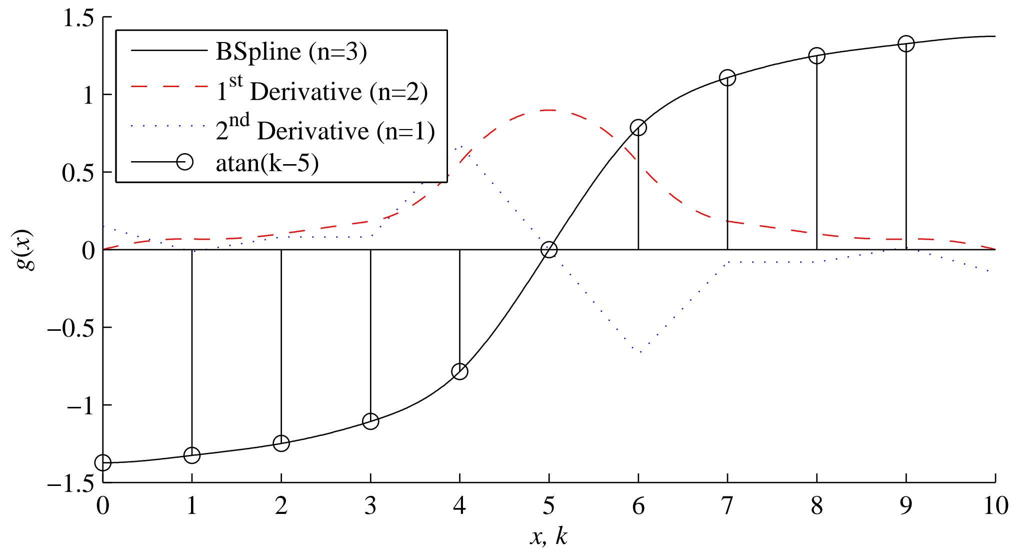

Figure 2.5: The first and second derivative of the spline representation of arctangent.

a signal rather than traditional approaches such as linear filtering (or bilinear or trilinear for higher dimensions) can produce smoother results, since the B-Spline is guaranteed to have continuous $(n-1)^{st}$ derivatives at all points. This reduction in aliasing can allow for smoother downsampling or less aliasing if the signal is to undergo subtle deformations.

This interpolation can be thought of as an extension of the indirect filter that calculates the intermediate values of β^n. Evaluating an arbitrary point in the range of the signal is accomplished by evaluating the contributions of all nearby spline coefficients as weighted by the B-Spline basis functions centered around that point. The smooth curves plotted in Figure 2.4 were generated by interpolating the splines at very small intervals, revealing their continuous shape.

2.1.4 Partial Derivatives

A task that is often performed in 2-D and 3-D image analysis is computing the Laplacian of an image - the sum of the second partial derivatives along each dimension. This is a separable operation like the direct and indirect filters, and by using the B-Spline representation, an analytic solution can be found using discrete operations, effectively performing the power rule from

calculus to determine the coefficients to the spline (of a lower degree) that represents the derivative of the original spline.

From property (2.8) in [15]:

$$\frac{\partial^2 \beta^n(x)}{\partial x^2} = \beta^{n-2}(x+1) - 2\beta^{n-2}(x) + \beta^{n-2}(x-1) \tag{2.6}$$

Since this analytic solution is a linear function of B-Splines, a signal with B-Spline coefficients $y(k)$ has a second derivative with B-Spline coefficients (for a spline of order $n-2$) of $y(k) * [1, -2, 1]$. While the laplacian is the focus of this operation, any derivative can be calculated, and any spline of order n is guaranteed to have n non-zero derivatives.

Figure 2.5 shows a cubic spline (n = 3) representation of a discrete signal (in this case samples taken from the arctangent function) and its first and second derivative splines. It can be seen from the interpolation on the second derivative plot that the spline representation is linear as expected (n = 1).

2.2 Applications of B-Spline Processing

To fully appreciate the utility of having a fast implementation of these B-Spline operations, it is worthwhile to look at some of the applications of these algorithms.

Nonrigid image registration can particularly benefit from a B-Spline representation of image and voxel signals [11]. In the paper by D. Rueckert (1999), an algorithm for nonrigid registration is presented to address the problem of comparing multiple 3D magnetic resonance imaging scans of cancer patients. Multiple MRI scans are used to track the motion of a contrast agent within the patient, however even the motion of normal respiration is enough to clutter the time differences. The registration algorithm aligns these images by computing an optimal nonrigid transformation based on the gradient vector in the 3D voxel volume. Both the gradient vector and the ultimate resampling of the data are performed using B-Splines for their continuous nature (making the analytic gradient easily computable) and their natural representation of control points (knots).

Contour outlining for biomedical imaging using cubic B-Splines was explored by P. Brigger (2000) in [4]. This work specifically takes advantage of the ability for the knots of B-Splines to be spaced irregularly. Conventional approaches to contour detection focus on minimizing the energy of a curve, but this implies calculating a number of weights for every control point on the contour based on energy regularization. The approach in this paper uses the spacing of the control points themselves to minimize the energy of the surface, resulting in fewer control points to evaluate and higher performance.

The most direct use of B-Splines is to image processing. Ruijters and Thévenaz explain in [12] that while the sinc interpolator is theoretically the best way of interpolating a sampled signal such as an image, usually only the nearest-neighbor or linear interpolation methods are implemented in hardware such as GPUs. The quality of interpolation afforded by cubic splines, however, has a profound effect on images, especially when undergoing consecutive iterations. Their paper demonstrates rotating an image in 36 steps of 10 degrees each, resampling each time using each the nearest-neighbor, linear, and cubic interpolation approach. The cubic approach has significantly fewer artifacts, and demonstrates the importance of having algorithms for rapid evaluation of higher-order splines.

Chapter 3

Existing Solutions

In this section, several recent and classic approaches to implementing B-Spline filters are considered. Ultimately no single solution was found that can perform all the B-Spline operations described above, nor that takes full advantage of modern parallel hardware in ways that can scale to very large data sets.

While these existing solutions are impressive in their speedups, and reveal some important considerations in creating a parallel implementation of the B-Spline prefilter, they fall short of being useful in practice. Rarely is the computation of spline coefficients for interpolation the end goal of research; much more likely, a researcher wishes to temporarily convert their data into the B-Spline domain to perform analytic operations such as resampling or derivation, and then return this data to the intensity domain. These previous solutions provide implementations for one small part of this workflow, but even simply having the CUDA code available does not provide a way to easily access the utility of this acceleration in an existing algorithm.

Additionally, operating on small images or voxel volumes is not quite representative of the actual use cases of these filters. In medical imaging, much larger image volumes may be produced. Conventionally resampling these images to be of a resolution that can be operated on by existing solutions may cause loss of data or aliasing that could be important for algorithms such as registration [1]. While this resampling may inevitably be necessary, it would be greatly beneficial to be able to perform it in the B-spline domain itself rather than on the raw intensity data.

These existing implementations also focus explicitly on cubic splines. While these do tend to be the most common splines in use, it would be nice

if a parallel solution allowed different orders of splines to be evaluated. Additionally, the theoretical definitions of B-Splines describe other parameters such as smoothing that influence the filters that these existing implementations do not include.

Since both of these are the subject of this thesis, it is important to point out the intent, scope, and capabilities of these other examples, both in terms of limitations and contributions.

3.1 Unser *et al.*

The first major work on this topic was done by Unser *et al.* [16]. In a companion paper to [15], they describe efficient means of implementing the various algorithms that were earlier discussed in a strictly mathematical context. They demonstrate that the direct and indirect filters, complete with the ability to smooth and downsample the signal, can be described as discrete convolutions.

They propose a class of recursive algorithms to implement the direct transform. By observing that the filter has a causal and anti-causal component, it can be written as two convolution operations that can be run forward and backwards over the signal to evaluate it in-place. This approach to calculating the spline coefficients is efficient for sequential processors, since it puts constant constraints on memory necessary in the evaluation and can be shown to avoid propagation of error, but it is not tailored to run on modern parallel hardware such as GPUs. A parallel architecture cannot directly make use of the benefits of such a recursive implementation, since each iteration relies on the computed data of the iteration prior. Indeed, there is more parallelism to be found by implementing the original non-causal convolution directly than pursuing the causal/anti-causal factoring approach.

3.2 Ruijters and Thévenaz

Work has been done to perform these operations in CUDA and other shading languages, taking advantage of NVIDIA GPUs. Ruijters and Thévenaz

propose a CUDA implementation of cubic B-Spline filtering for image processing. Their work is motivated by the fact that GPUs support nearest-neighbor and linear filtering natively, but not cubic. Their implementation follows the same approach as Unser *et al.* . It performs the one-dimensional recursive filtering on each row and column of image data, but the ultimate parallelization is limited to one thread per row due to the inherent sequential nature of the recursive filter. Since this filter must be applied once for every dimension, the parallelization available is only a function of the number of pixels perpendicular to the dimension of evaluation. (A voxel volume of m x n x p elements would require three passes, each with a best-case parallelization of n x p, m x p, and m x n.) For voxel volumes of 16Mpx, they report speedups of approximately 20x over their CPU implementation [13], which is significant, but this approach does not scale as well as could be hoped, especially if the image has uneven dimensions. One could imagine a worst-case scenario with a voxel volume that is only a few pixels tall and wide, but thousands of pixels deep. Only a handful of threads would be capable of running in parallel to evaluate the direct filter along the dimension of depth since each pixel would require the pixel just before it to have been evaluated first.

3.3 Champagnat and Le Sant

Champagnat and Le Sant have also produced a CUDA implementation of cubic spline interpolation. Their approach introduces a concept used in this thesis: they demonstrate that the infinite impulse response filter used in the direct transform decays rapidly for more distant elements. By truncating this to a smaller number of elements, the infinite response can be implemented as a finite response and evaluated using traditional convolution approaches against a reasonably small kernel without sacrificing much accuracy. They were satisfied that the use 15 coefficients in their implementation was sufficient to produce a result within the error of the floating point representation itself, and have a CUDA kernel that operates very similarly to the canonical convolution examples in the NVIDIA SDK. They report a speedup of 4.4x on a GTX 260M, with a 1Mpx image, and 10x on a Tesla C2050 for

a 4Mpx image, as compared to the GPU implementation of the infinite response [5]. While this implementation demonstrates the atainable speedups with a parallel-optimized version of the direct filter, the scope of this paper is limited to cubic interpolation, and it does not attempt to address the issue of scaling beyond the available GPU memory.

Chapter 4

Implementation

4.1 Overview of GPU Architecture and CUDA

Modern graphics processing units (GPUs) are special purpose parallel co-processors to more conventional sequential processors (CPUs). Originally designed to handle the calculations necessary for 3D computer graphics, recent hardware developments have made the processors more programmable and generalized their application. In 2007, NVIDIA released CUDA, an architecture, programming model, and language extension that allows access to the parallel architecture in NVIDIA GPUs [9].

CUDA provides an abstraction over the complex and varied architectures of individual models of GPU. All general purpose GPUs consist of arrays of streaming multiprocessors, each implementing a single-instruction multiple-data approach to execution. A CUDA program defines a single parallel function - a kernel - that is executed on these multiprocessors. Multiprocessors are grouped into blocks, and the CUDA abstraction only presents these higher-level blocks to the programmer.

A CUDA program launches a kernel over a one, two, or three dimensional grid. This grid is divided into blocks, and each block contains a number of individual threads. Blocks execute synchronously and have some shared resources (including registers and shared memory). These thread blocks are required to execute independently to allow the GPU to schedule their execution optimally.

The GPU contains a large global memory area, and this is the only memory which may be directly interacted with from the CPU host. The latency to access this global memory from a kernel is high, so there are multiple

mechanisms in place to allow these accesses to be cached. Texture memory is a kind of cache that shows the original purpose of the global memory - to hold texture imformation for 3D graphics. This cache exploits spacial locality in memory and can perform various interpolation functions in hardware. Another memory section is the shared memory, mentioned earlier. This memory is much faster than global memory, but local to blocks. It is often used as a kind of manual cache, when the needs of the kernel are known ahead of time. Many GPUs also contain L1 and L2 cache for each multiprocessor. [10]

4.2 Overview of Library Architecture

The software accelerates the operations involved in B-Spline signal processing by harnessing the parallel computing architecture of NVIDIA graphics processors. CUDA kernels that perform each operation were written, and the details of their implementation and data structures were encapsulated in the C++ class `BSArray`. The kernels are responsible for allocating and freeing GPU memory as necessary, and operate on arrays of single-precision floating point data. The detailed operation of these kernels is found in the following sections.

This `BSArray` class represents a single signal of one, two, or three dimensions, and a single B-Spline representation of that signal with parameters for degree and smoothing along each dimension. The class achieves this representation by storing single-precision floating point arrays for both the original intensity signal and the coefficient array. The coefficient array is stored either on the CPU host memory or in the GPU's global memory, depending on whether the CPU or GPU version of the direct filter was invoked to generate the coefficients. (This choice implies that individual instances of this class are committed to either CPU or GPU operations - one cannot, for instance, perform the direct filtering on the CPU and use the GPU to calculate the partial derivatives.)

The class project is compiled as a 64 bit Windows static library. This allows the library to be linked in and used with other projects easily. Static linkage also simplifies the distribution of binaries (such as MEX files or

standalone programs) that use this accelerated library since they do not need to distribute any of the library's object code separately.

This library is used by several other binaries, known as MEX files (for MATLAB EXecutable). These are dynamically linked libraries with the extension `.mex` that expose particular symbols that allow them to be called from MATLAB as functions. One MEX file is created for each MATLAB operation that is accelerated by the CUDA implementation. The MEX files are responsible for interpreting the parameters passed in from MATLAB (which includes meta-parameters such as whether to use the CPU or GPU implementation, and how many iterations to perform for average timing measurement).

A MATLAB class was written to encapsulate the MEX functions and to handle any additional interface details, as well as perform timing measurements using the MATLAB `tic` and `toc` commands, which are the standard in-situ performance profiling commands, measuring the elapsed real time using the system clock. The MATLAB class that uses the MEX files is designed to have a nearly identical API to the original class which implements the B-Spline transforms purely in Matlab. Additional optional parameters are included in the class methods to specify which implementation to use, and how many timing iterations to perform. (Multiple iterations are performed on all operations to increase the precision of the timing measurements and to average out any irregularities.)

A MATLAB script was written to test the functionality of the new library and API, and also to compare the performance of the existing pure MATLAB implementation to the new C++ library. The script loads in example images, both two and three dimensional, at many different resolutions. (The content of the images has no impact on the performance of any of these algorithms.) These example images are them transformed into their B-Spline representations using all three exposed implementations. The indirect transform is run to evaluate the B-Splines and recover an intensity image. The relative error between this recovered image and the original image, as well as between each implementation, is calculated using the root-mean-squared formula. These errors metrics are used to verify correctness of the varying implementations, and unless otherwise noted the errors found are on

the order of the machine epsilon for single precision values and considered acceptable. Once again using the B-Spline representation, the laplacian is computed by finding the magnitude of the second partial derivative (evaluated along each dimension, which reduces the order of the spline by two, from cubic to linear). This spline representation of the laplacian of the original image is then evaluated by the indirect transform to arrive at an intensity representation which is checked for errors. The duration of each function call is also measured and recorded for comparison.

4.3 Details of Parallel Functions

This section will look more closely at the details of the parallel CUDA implementation, including rationale for how the various types of memory are used, how the parallel threads are organized, and a brief analysis of how these design choices affect the performance of the system.

4.3.1 Direct Transform

The direct transform is the first operation that is applied to the source intensity signal. This input signal is initially stored in host (CPU) memory. It is copied to a 3D array on the device (GPU) using `cudaMalloc3D`, which ensures that alignment requirements are met to allow efficient access. Another 3D array is also allocated to hold the results - the coefficients for the B-Spline representation.

The direct transform itself mathematically represents an infinite impulse response filter, but as described earlier, the filter has an exponential decay, meaning that only nearby elements make a significant contribution. This allows us to represent the convolution kernel with a finite number of coefficients. Since the filter is symmetric as well, only half of the total size of the filter needs to be stored.

Each dimension in the signal may have different parameters for the B-Spline representation, and so for each axis this convolution kernel is found independently. The filter coefficients are found by running the classic recursive exponential filter on the CPU over a unit impulse signal. This quickly

generates the impulse response, cropped to the length desired. These coefficient arrays are copied to the device's constant memory for quick read-only access in all threads.

The device arrays are in global memory, but global memory is the slowest to access. Since the nature of this operation requires that each source element is read multiple times (once for every element in the convolution kernel for each dimension), some acceleration structure must be used. For this kernel, since each thread needs to access a fairly large number of elements that exhibit some spacial locality, and that many of these source elements need to be accessed from multiple blocks, the texture memory approach was chosen. Texture memory acts as a hardware accelerated cache over global memory that takes advantage of spacial locality in accesses. Since such a high fraction of elements are accessed between blocks (since the convolution filter is fairly wide), this global cache solution was chosen rather than relying on block-local caches such as shared memory, even though the shared memory may have faster individual accesses. To this end, a 1D (linear) texture reference was bound to the source device array.

Once the data has been copied to the device arrays, these arrays have been mapped to texture units, and the coefficients are stored in constant memory, the threads can be launched. Each thread ultimately performs a single inner product to compute the value of a single output element by multiplying the appropriate input array elements by the convolution kernel coefficients. The threads are organized into 1, 2, or 3 dimensional blocks (depending on the dimensionality of the signal) using a block size of 256, 16x16, or 8x8x8 elements respectively. These block sizes were chosen to attempt to maximize the occupancy of the hardware resources. Sufficient blocks are launched to cover the whole source signal. Blocks are scheduled to hardware resources by the CUDA driver.

The CUDA threads that actually do the processing are concise. Each thread calculates the coordinate of the output pixel it is assigned to based on the block and thread index. Bounds checking is performed to accommodate signals that are not a multiple of the block size. Pointers into the source and destination arrays are calculated using simple pointer arithmetic based on the thread's coordinate and the pitch of the aligned device array.

A small loop (with a `#pragma unroll` statement included to automatically unroll the loop) is used to perform the inner product. The source element is read with a call to `tex1Dfetch` to make use of the texture cache. The coordinates to fetch are based on the thread's coordinate as well as the offset along the dimension that the inner product is being evaluated. The coordinate in this dimension is passed through a function that causes out-of-bounds indices to be mirrored back into the bounds of the array, which implements the bounds checking behavior of the original MATLAB implementation. Once the inner product has been evaluated, the result is placed into the destination array.

Three variants on the CUDA kernel exist to perform the filtering on the X, Y, and Z dimensions individually, since the index calculations and bounds checking must be handled uniquely for each dimension. Between invocations of each kernel, the destination array is copied to the source array, ensuring that each kernel begins with the finished results of the previous kernel.

Once all needed kernels are run, the device array contains the B-Spline coefficients. This array is copied back to the host, but the device memory is not freed. Rather, the device pointer is stored so that future operations on this B-Spline object do not incur the needless cost of copying the coefficients back from the host to device.

4.3.2 Indirect Transform

The indirect transform is similar to the direct transform, with a few differences that affect the implementation. This transform convolves the coefficient array against a small kernel (the length is equal to the order of the spline) to produce the intensity signal - essentially evaluating the spline at each point. Two new device arrays are allocated to hold the results of this transform. One is used as the output buffer and the other as the input buffer. The input buffer is initialized with the array of spline coefficients already on the GPU by copying from the device pointer stored during the direct transform.

The convolution kernel coefficients are calculated on the host as a function of spline order and the smoothing value for each dimension and are stored in the device's constant memory. With the spline coefficients and the convolution kernel in place, the threads are launched for each dimension.

The blocks are of the same scale as before, however each thread no longer computes just one result element. Since the amount of computation done in each thread is small compared to the direct filtering case, each thread computes multiple values (chosen to be 4 to maximize performance by manual tuning). Also since the convolution kernel is smaller, it becomes beneficial to use a more local memory for caching. The indirect transform kernels implemented here use shared memory to preload each source element once (border elements may be loaded multiple times, but the small kernel size reduces this overlap). This approach is taken directly from the 3D convolution example in the CUDA SDK.

The results of the indirect transform are intensity values that do not serve as an input to any other B-Spline operations, so the device array is copied back to the host and freed.

4.3.3 Interpolation

The interpolation function is unique in its parellel implementation. The objective of this function is to evaluate the B-Spline surface at any point within the bounds of the signal, interpolating from surrounding points as necessary. This function is useful not only for evaluating specific points but also for resampling entire images non-uniformally. Normally a large number of points are evaluated simultaneously, none of which can be expected to have the same relative offset from a knot. The evaluation of the spline function requires coefficients that are a function of this offset; it can be considered a general case of the indirect filter, but where the convolution kernel for each element is unique and the elements to evaluate may not have a one-to-one correspondence with image elements at all.

It is therefore the job of the individual threads to evaluate the B-Spline basis functions at the appropriate offsets to determine the contribution of each nearby coefficient value to the final intensity value of the spline. For a

spline of degree n on a signal of tensor order d, $(n + 1)^d$ coefficients must be uniquely weighted per result value.

A vector of coordinates at which to evaluate the spline is supplied to the function. This vector, along with space to store the result values, is placed in the device's global memory. Since there is no inherent dimensionality in this representation, the threads are issued on a one-dimensional structure of blocks. Again due to the arbitrary request structure, there is little possibility for manual caching that could improve on the device's hardware cache, and so none is used. In the section on future work, it is discussed how another more structured interface to this operation may provide further opportunities for optimization without sacrificing much flexibility in use.

4.3.4 Laplacian

The Laplacian operator diverges from the original MATLAB implementation of the B-Spline's library slightly in that originally the function calculated the first partial derivative only, and two successive calls were made to find the second partial derivative of a signal. It was found that the vast majority of use cases followed this pattern, so for efficiency the second partial derivative is calculated directly, requiring only one call per dimension. The only implementation difference is the differentiation kernel used.

This differentiation kernel is applied to the B-Spline coefficients to obtain the coefficients to another B-Spline (of two degrees smaller in the dimension of differentiation, from the power rule). Since the process generates a B-Spline of a different size, and the operation should be nondestructive, a new BSArray class instance is created to hold the proper number of elements. Unlike the direct and indirect transform functions, only the output device array is allocated (and becomes the stored device coefficient array for the new BSArray object). Since this operation only alters one dimension at a time, the input array remains unchanged and can be used directly.

The CUDA kernels for this operator are very simple. Each computes an output element based on the weighted sum of three elements of the input array (implementing a second order central discrete difference operator).

Since each element in this case is only accessed three times in a very regular pattern no acceleration structure was used to speed up global memory accesses, relying only on the device's L2 and L1 cache local to each multiprocessor. (The performance was adequate with this approach however using another acceleration structure may provide further improvement, and is considered in the chapter on future work.)

4.4 Hardware Considerations and Optimization

There are several limitations that are imposed by the hardware used to run these operations. The most critical metric is the amount of available memory. The direct and indirect filter both require two additional device arrays to be allocated of the same size as the coefficient array which, for the indirect filter, is already allocated. This means that if the device has insufficient free global memory to store three copies of the single-precision coefficient array, the operation will fail. If many BSArray objects exist at the same time, they will all compete for GPU memory.

In addition to the limitations imposed by the device memory, the available host memory may also be insufficient for certain tasks, especially when run through MATLAB. MATLABmakes the source intensity data available as an array of double precision floating point values, which the implementation copies to a single-precision buffer before uploading to the device. For very large source images, the allocation of this extra array (especially if many BSArray objects are in use at once) may exceed the system's available memory.

An additional consideration on hardware limitations is the processing capability of the GPU being used. Large source images may saturate the GPU device, reducing the speedup as there are more blocks than can be run on the available multiprocessors. Depending on the resources available to a specific GPU (including architectures and models that do not yet exist), tweaking the block sizes for all the kernels may alter the performance. This is naturally an issue with all GPU accelerated programs, but worth noting to help assess the practical benefits and limitations of this solution.

Chapter 5

Results

Several tests were performed on the CUDA implementation of these B-Spline operations. Each operation was tested individually under two and three dimensional images of various sizes, and a further test of a combination of operations was run to demonstrate a practical scenario for using the library. This final scenario consists of performing the direct filter on the image, computing the magnitude of the gradient vector (the laplacian) of the image, and finally running the indirect filter on this new B-Spline.

5.1 Sample Datasets

Two images were selected for these tests: a two-dimensional GigaPan image of the Devil's Golf Course in Death Valley National Park, with a resolution of 61,028x14,941 pixels (shown in Figure 5.1), and a sample three-dimensional magnetic resonance (MR) full body scan with a resolution of

Figure 5.1: GigaPan of Devil's Golf Course, Death Valley, USA.
A grayscale version was used for testing.

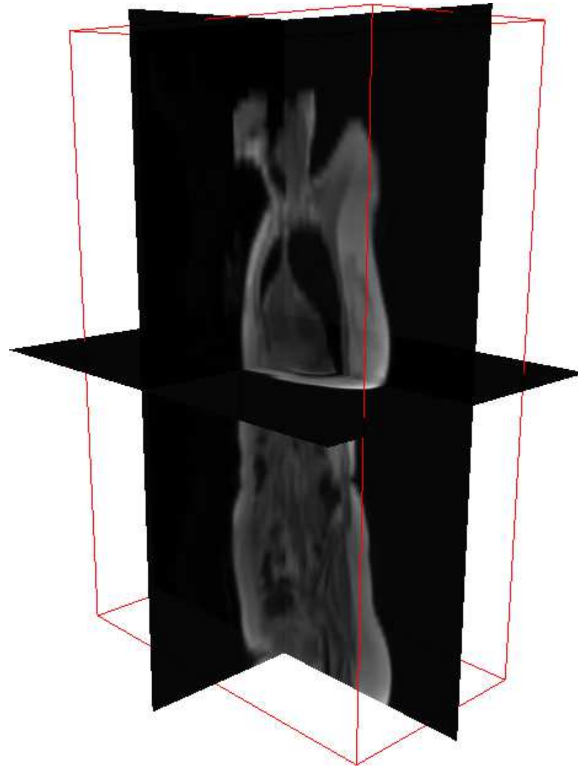

Figure 5.2: Slices of a three-dimensional magnetic resonance (MR) full body scan. [7]

400x300x151 voxels (shown in Figure 5.2). Each of these images were reduced in scale by half several times to produce images of smaller scales for testing; the Devil's Golf Course image was reduced 5 times to a final resolution of 1908x467, and the MR image was reduced three times to a resolution of 51x39x20. The two highest resolutions of the GigaPan image were too large for the system to currently handle. Explanations and solutions to this are explored in the section on future work.

It should once again be noted that none of these B-Spline algorithms are dynamic, and therefore the content of the images used in testing has no bearing on the complexity or performance of the system.

5.2 Test Environment

The tests 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.

5.3 Performance Results

Tables 5.1 and 5.3 contains the timing data collected from the direct filter, indirect filter, laplacian operation, and the combination of all three on a cubic B-Spline with $\lambda = 0$. (Altering λ would have no effect on the performance, but would merely change the result and make validation more difficult.) Performace values on the C++ implementation used for validation are included for comparison, though this C++ implementation is not intended to be optimal. The relative speedup of the CUDA implementation of each algorithm over the MATLAB implementation is given in Tables 5.2 and 5.4. The data is also summarized in the plots of Figure 5.3.

Image Size (px)	1908x467	3815x934	7629x1868	15257x3736
GPU Direct	0.0349435	0.168642	0.703172	2.97078
C++ Direct	0.0577606	0.330237	1.43281	6.00256
Matlab Direct	0.111017	0.367267	1.32345	5.08171
GPU Indirect	0.00433462	0.0124669	0.0450161	0.13825
C++ Indirect	0.0342882	0.146997	0.615947	2.79802
Matlab Indirect	0.0267627	0.0374136	0.0769091	0.248526
GPU Laplacian	0.00214226	0.00405299	0.0117717	0.0433254
C++ Laplacian	0.024445	0.119919	0.524821	2.48524
Matlab Laplacian	0.0530575	0.195527	0.708148	3.05992
GPU Interpolation	0.0179	0.0742955	0.262864	N/A
C++ Interpolation	0.234002	0.982837	4.08019	N/A
Matlab Interpolation	1.18228	4.76937	19.111	N/A
GPU Practical	0.0668724	0.275525	1.01457	4.21061
C++ Practical	0.269582	0.900768	3.40543	14.7385
Matlab Practical	0.18977	0.654231	2.42234	9.50324

Table 5.1: Average execution time in seconds for 2D image. Missing data is due to memory limitations.

Image Size (px)	1908x467	3815x934	7629x1868	15257x3736
Direct	3.177042941	2.177790823	1.882114191	1.710564229
Indirect	6.174174437	3.00103474	1.708479855	1.79765642
Laplacian	24.76706842	48.24265542	60.15681677	70.62646854
Interpolation	66.04916201	64.19460129	72.7029947	N/A
Practical	2.837792572	2.374488703	2.387553348	2.256974643

Table 5.2: Speedups of CUDA over MATLAB for 2D image.

Image Size (voxels)	51x39x20	101x76x39	201x151x76	400x300x151
GPU Direct	0.00233725	0.0136622	0.108884	0.989294
C++ Direct	0.00374299	0.0327844	0.379837	2.97501
Matlab Direct	0.070434	0.245027	1.07005	5.17591
GPU Indirect	0.00653725	0.00217969	0.00979878	0.0603285
C++ Indirect	0.0019047	0.0178024	0.226991	2.8662
Matlab Indirect	0.0239431	0.0287211	0.0384704	0.241422
GPU Laplacian	0.000595558	0.00250267	0.00503281	0.0227507
C++ Laplacian	0.000909788	0.0080598	0.0788267	1.32513
Matlab Laplacian	0.0393234	0.0304353	0.177145	1.32812
GPU Interpolation	0.00162156	0.021428	0.163852	1.6498
C++ Interpolation	0.0188506	4.03826	32.4409	259.059
Matlab Interpolation	0.501686	3.71272	28.9187	228.407
GPU Practical	0.0110168	0.0360151	0.214102	1.67243
C++ Practical	0.122215	0.206587	1.15923	11.4194
Matlab Practical	0.0919112	0.301141	1.36455	7.70801

Table 5.3: Average execution time in seconds for 3D volume.

5.3.1 Analysis and Scalability

In every test case, the performance of the CUDA implementation was better than that of the MATLAB implementation. For the laplacian filter, this speedup is quite impressive, with a speedup of over 70x in one instance.

Having these speedups is expected due to the new parallel implementation. The interesting information comes from observing the changes in speedup as the size of the image data changes. For the direct and indirect filters, the larger images are experiencing a greater speedup. The laplacian operator exhibits the opposite behavior. This result is likely an indication of the length and memory access behavior of each CUDA thread. Specifically, once the GPU has issued warps to each multiprocessor and the hardware is

Image Size (voxels)	51x39x20	101x76x39	201x151x76	400x300x151
Direct	30.13541555	17.93466645	9.827431028	5.231922967
Indirect	3.662564534	13.17669026	3.926039772	4.001790199
Laplacian	66.02782601	12.16113191	35.19803052	58.37710488
Interpolation	309.38479	173.2648871	176.4928106	138.4452661
Practical	8.342821872	8.361520584	6.373364097	4.608868533

Table 5.4: Speedups of CUDA over MATLAB for 3D volume.

performing as many operations in parallel as possible, the throughput begins to scale as a function of the behavior of individual warps. The direct and indirect filter both implement convolutions of a larger scale than the laplacian filter. Because of this they also spend more time waiting on memory accesses, leaving room for the GPU to schedule additional warps and increase throughput. The multiprocessors are more fully utilized and there is relatively less overhead. With the simpler kernel, once all the multiprocessors are in use, there is less opportunity to work on any additional blocks in parallel. In this case, making the laplacian kernel perform more work - perhaps by computing multiple results per thread - would decrease the overall number of threads and provide better utilization, and ultimately higher throughput for larger images.

The one instance where the performance of the laplacian operator is greater for smaller images is likely due to the overhead of the parallel architecture outweighing the performance benefits of parallelizing such a simple operation.

A final note on the scalability involves the limitation of being unable to process the two largest images in the dataset. The issue with these images does not actually concern the GPU implementation, but rather the host memory. Between the data needed in MATLAB and the additional memory allocated by the library to perform the upload, the host had insufficient RAM (16GB) to complete the operation. Since 16GB is still considered reasonably high-end for desktop computers, this limitation is a cause for concern. Even if the system memory were increased, the GPU memory would soon become a similarly limiting factor. In the next section on future work, a solution is explored that could help remove this strict dependancy on images completely fitting within system (or GPU) memory.

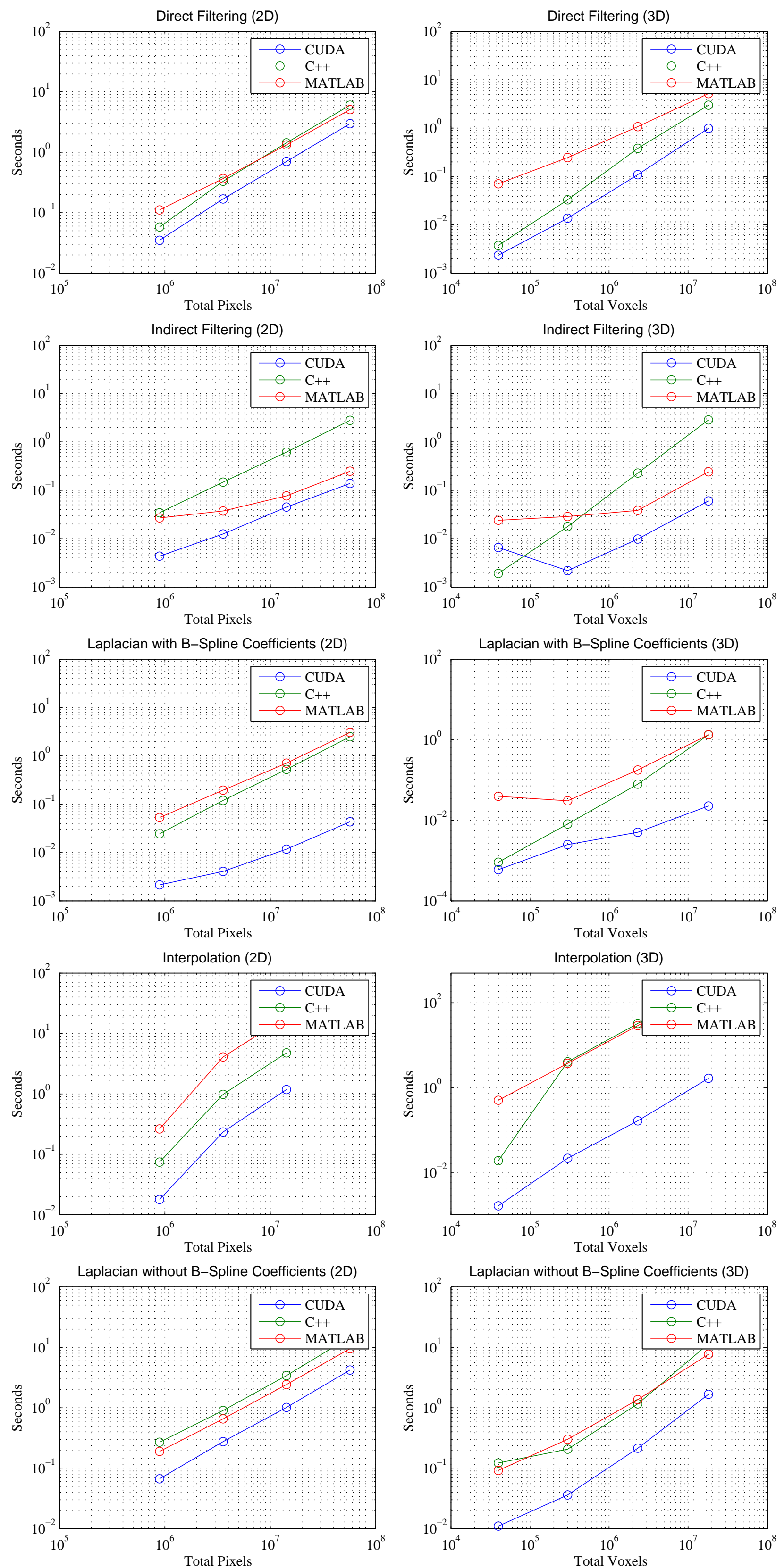

Figure 5.3: Results of timing of various B-spline operations as implemented in MATLAB, C++, and CUDA and applied to different levels of the 2D and 3D images. [7]

Chapter 6

Conclusion and Future Work

This thesis has discussed the background of B-Spline signal processing and given a brief overview of how these fundamental operations are used at the heart of many more complicated algorithms. Anywhere a continuous representation of a discrete signal is desired to allow for interpolation, differentiation, or noise filtering, B-Splines are a robust and useful tool. However the cost of accurately using these splines is high, especially when considering very large sets of data.

To this end a C++ library was written that accelerates these fundamental B-Spline operations using CUDA, the programming toolkit that exposes the highly parallel architecture of NVIDIA's graphics processors. This library can be used both directly by C++ applications or through MATLAB via a series of wrapper executables and MATLAB functions.

The library was profiled against very large two and three dimensional images and shown in every case to be an improvement over the existing MATLAB implementations of these algorithms and in most cases to scale well with image size. Limitations were found in the largest of test images that will need to be addressed in future work. Overall this project has demonstrated the feasibility of flexible and generic parallel B-Spline signal processing operations on a graphics processor.

The work that remains to bring this library to its fullest potential is mostly related to this issue of scalability. The immediate problems that were faced in the collection of performance data for this thesis could have been avoided by increasing the available system memory, for instance, but the fundamental problem of having more data than working memory will not go away so easily. There are alternatives, however, that can make use of the accelerated

spline operations to allow larger images to be used more effectively.

The first proposed improvement to the library is to support loading images directly from the C++ library itself. MATLAB is not as efficient with loaded images as will be necessary to push the envelope of supported resolutions. By loading images through C++ directly, perhaps using an open-source image library, the software could ensure that no bit depth is wasted when storing the image before uploading to the GPU, and it might take advantage of using memory-mapped files and allowing the operating system to page the data efficiently. Especially in combination with streaming from files, it may be possible to perform B-Spline decimation on smaller regions of the image sequentially. The software could read in enough of the image to fill the available device memory and then, using splines, produce a down-sampled array of coefficients for that region and repeat as necessary. This has the benefit of downsampling with the quality of splines rather than with linear approaches, and the coefficients can remain on the device to be used immediately.

Another proposed improvement which does not directly relate to scalability is to increase the range of parameters supported. Currently, splines of orders 0 through 7 are supported when the smoothing parameter λ is set to 0, and only the lowest three spline parameters when the smoothing parameter is increased. The reason for this omission is simply that these use-cases are a low priority. They should be implemented for maximum flexibility and full feature parity with the original MATLAB implementation.

Bibliography

[1] R.E. Ansorge, S.J. Sawiak, and G.B. Williams. Exceptionally fast non-linear 3d image registration using gpus. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4088 –4094, 24 2009-nov. 1 2009.

[2] M. Arigovindan, M. S uhling, C. P. Jansen, and M. Unser. Variational image reconstruction from arbitrarily space samples: a fast multiresolution spline solution. *IEEE Transactions on Image Processing*, 14(4):450–460, 2005.

[3] L. Baboulaz and P. L. Dragotti. Image super-resolution with b-spline kernels. *Proc. 7th IMA International Conference on Mathematics in Signal Processing*, December 2006.

[4] P. Brigger, J. Hoeg, and M. Unser. B-spline snakes: a flexible tool for parametric contour detection. *IEEE Transactions on Image Processing*, 9(9):1484–1496, 2000.

[5] F. Champagnat and Y. Le Sant. Efficient cubic B-spline image interpolation on a GPU. *Journal of Graphics Tools,* submitted, 2012.

[6] C. de Boor. *A Practical Guide to Splines*. Springer Verlag, 1978.

[7] A. Karantza, S. Alarcon, and N. Cahill. A comparison of sequential and gpu-accelerated implementations of B-spline signal processing operations for 2-d and 3-d images. In *3rd International Conference on Image Processing Theory, Tools and Applications (IPTA),* in press, 15-18 Oct 2012.

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

[9] NVIDIA, Inc. *NVIDIA CUDA Architecture: Introduction & Overview*, 2009.

[10] NVIDIA, Inc. *NVIDIA CUDA C Programming Guide*, 2012.

[11] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes. Nonrigid registration using free-form deformations: Application to breast mr images. *IEEE Transactions on Medical Imaging*, 18(8):712–721, 1999.

[12] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens. Efficient GPU-based texture interpolation using uniform B-splines. *Journal of Graphics, GPU, and Game Tools*, 13(4):61–69, 2008.

[13] D. Ruijters and P. Thévenaz. GPU prefilter for accurate cubic B-spline interpolation. *The Computer Journal*, 55(1):15–20, January 2012.

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

[15] M. Unser, A. Aldroubi, and M. Eden. B-spline signal processing: Part I - theory. *IEEE Transactions on Signal Processing*, 41(2):821–833, February 1993.

[16] M. Unser, A. Aldroubi, and M. Eden. B-spline signal processing: Part II - efficient design and applications. *IEEE Transactions on Signal Processing*, 41(2):834–848, February 1993.

Chapter 7

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.

7.1 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.