Fast Spline Interpolation using GPU Acceleration

Sebastian Kazmarek Præsius[1], Jørgen Arendt Jensen[1]

[1]Center for Fast Ultrasound Imaging, DTU Health Tech.
Technical University of Denmark, DK-2800 Lyngby, Denmark

Abstract*—Super-resolution imaging uses spatiotemporal filtering to separate moving blood from tissue, but the motion from, e.g., heartbeats and breathing reduces the effectiveness of these filters. Therefore, a motion correction is often performed to make the tissue stationary at a sub-pixel level through interpolation. However, the high-quality interpolation method cubic spline lacks an implementation for graphics processing units (GPUs), making it slow and impractical. The hypothesis is that motion correction can be performed at a 5000 Hz frame rate with GPU-accelerated spline interpolation, enabling this step in super-resolution imaging pipelines to run in real time. Spline interpolation was derived in 1D and then generalized to 2D. The system of equations for the necessary derivatives for spline interpolation was shown to be reducible to a convolution for large inputs within numerical precision, which could be parallelized on GPU. In-vivo data from a Sprague-Dawley rat kidney was acquired with a 10 MHz, 168-element GE L8-18iD linear array probe and a Verasonics Vantage 256 scanner. Motion was estimated using a transverse oscillation autocorrelation motion estimator, and then 2D motion correction was performed by resampling already-beamformed images with new pixel positions. An NVIDIA GeForce RTX 4090 GPU and Intel Xeon W-3223 CPU were used for processing, and the output was compared with that of MATLAB's interp2. Motion correction was performed at the rate of 5013 images per second using the proposed GPU implementation, which was 668 times faster than MATLAB's 2D spline interpolation, which lacks GPU support. Comparing the output to that of interp2 showed only insignificant numerical differences of less than -75 dB. Thus, GPU-accelerated motion correction for 2D imaging can be performed in real time using spline interpolation to benefit super-resolution and power Doppler imaging. The source code is made freely available online.

I. Introduction

Super-resolution ultrasound imaging or ultrasound localization microscopy have become popular methods for acquiring high-resolution images of vasculature from ultrasound [1]–[4]. Traditionally, these techniques required an injection of contrast agents (microbubbles), which were sparse, to allow individual peaks to be detected and localized with sub-pixel accuracy [5]. Recent methods have attained super-resolution without the use of contrast agents [6]–[9], bringing down the acquisition times.

Even with short acquisitions, the motion from breathing and heartbeats is a problem. It reduces the sensitivity to slow blood flow, and it can result in a loss of resolution from the lack of alignment during imaging [10]. Motion correction can be used to remedy this problem by aligning the tissue between frames through measured motion fields [10]–[13], making the images appear stationary (except for out-of-plane motion).

Interpolation must be used to account for sub-pixel motion, but the interpolation method "spline" of the interp2 MATLAB function currently lacks support for graphics processing units (GPU), making the processing slow for this high-quality interpolation method. The hypothesis is that the motion correction can be performed at a 5 kHz rate with a GPU implementation, enabling this step in processing to run in real time. The main contribution is the GPU implementation of spline interpolation for images, available at https://github.com/sebftw/interp2gpu.

II. Theory

This section describes the theory behind Spline Interpolation and derives how the spline is constructed. The theory from [14] is summarized here. A spline is made of polynomial segments, $q_i(x)$, each covering a subset of space as

$$S(x) = \begin{cases} q_1(x) & x_1 \leq x < x_2 \\ q_2(x) & x_2 \leq x < x_3 \\ & \vdots \\ q_{n-1}(x) & x_{n-1} \leq x \leq x_n, \end{cases} \tag{1}$$

where $x_1 < x_2 < \ldots < x_n$. For the spline to be interpolating, it must match the n recorded data points $(x_i,\, y_i)$ exactly [15]

$$S(x_i) = y_i \text{ for } i = 1 \ldots n. \tag{2}$$

An example satisfying (2) is "Nearest Neighbor Interpolation," where each segment matches the nearest data point. It is further required that the spline is continuous

$$S \in \mathcal{C} \Rightarrow q_i(x_i) = q_{i-1}(x_i) \text{ for } i = 2 \ldots n-1 \tag{3}$$

Methods satisfying (3) include, e.g., "Linear Interpolation" and "Lagrange Interpolation [16]." The spline must also be smooth, meaning that its derivative is continuous

$$S \in \mathcal{C}^1 \Rightarrow q_i'(x_i) = q_{i-1}'(x_i) \text{ for } i = 2 \ldots n-1 \tag{4}$$

An example satisfying (4) is the method "cubic" in `interp1` in MATLAB. A cubic polynomial has four degrees of freedom, and thus many solutions are satisfying eqs. (2) to (4). However, by additionally requiring 2nd-order smoothness

$$S \in \mathcal{C}^2 \Rightarrow q_i''(x_i) = q_{i-1}''(x_i) \text{ for } i = 2 \ldots n-1, \tag{5}$$

there are only two "unknowns" left, as there were $4(n-1)$ degrees of freedom and $n+3(n-2)$ constraints. For uniqueness, MATLAB's `interp1` uses the *not-a-knot* end conditions

$$q_2'''(x_2) = q_1'''(x_2) \text{ and } q_{n-1}'''(x_{n-1}) = q_{n-2}'''(x_{n-1}). \tag{6}$$

There must be at least four points: $n \geq 4$, since otherwise (6) is only one constraint. This derivation shows that all the methods

included in the MATLAB function `interp1` are splines. Still, the name "Spline Interpolation" is reserved for the cubic spline satisfying eqs. (2) to (6), which features 2nd-order smoothness.

A. Cubic Hermite Spline in 1D

This section derives the cubic Hermite form, which guarantees that the spline is smooth. Each cubic polynomial segment $q(x)$ has four degrees of freedom, as demonstrated by writing it in the *monomial form* with four arbitrary coefficients $\boldsymbol{a} \in \mathbb{R}^4$

$$q(x) = \sum_{i=1}^{4} a_i x^{i-1} = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} \boldsymbol{a} = \boldsymbol{p}(x)\boldsymbol{a}. \tag{7}$$

Given four data points (x_i, y_i) satisfying $x_1 < x_2 < x_3 < x_4$, there exists a unique cubic polynomial that interpolates them

$$\begin{aligned} \boldsymbol{y} &= \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} q(x_1) \\ q(x_2) \\ q(x_3) \\ q(x_4) \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}(x_1)\boldsymbol{a} \\ \boldsymbol{p}(x_2)\boldsymbol{a} \\ \boldsymbol{p}(x_3)\boldsymbol{a} \\ \boldsymbol{p}(x_4)\boldsymbol{a} \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}(x_1) \\ \boldsymbol{p}(x_2) \\ \boldsymbol{p}(x_3) \\ \boldsymbol{p}(x_4) \end{bmatrix} \boldsymbol{a} \\ &= \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix} \boldsymbol{a} = \boldsymbol{p}(\boldsymbol{x})\boldsymbol{a} \Leftrightarrow \boldsymbol{a} = \boldsymbol{p}(\boldsymbol{x})^{-1}\boldsymbol{y}, \end{aligned} \tag{8}$$

where $\boldsymbol{p}(\boldsymbol{x}) \in \mathbb{R}^{4\times 4}$ is a Vandermonde matrix, which is invertible as the x's are unique [17]. Cubic Lagrange Interpolation refers to the spline in which each segment is in the form

$$q(x|\boldsymbol{y}) = \boldsymbol{p}(x)\boldsymbol{p}(\boldsymbol{x})^{-1}\boldsymbol{y}, \tag{9}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ contain the four data points nearest x. However, this does not result in a smooth spline. The *cubic Hermite form* may be used instead to enforce smoothness, as defined in (4).

The cubic Hermite form can be derived from the monomial form in (7) by taking the derivative

$$q'(s) = \boldsymbol{p}'(s)\boldsymbol{a} = \begin{bmatrix} 0 & 1 & 2s & 3s^2 \end{bmatrix} \boldsymbol{a} = \boldsymbol{p}(s)\mathbf{D}\boldsymbol{a}, \tag{10}$$

where $x_1 < x < x_2$ is the point to be evaluated and $s = \frac{x-x_1}{x_2-x_1}$ is its fractional position. Differentiation is a linear operation, which conveniently turns into a multiplication by the matrix

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{11}$$

Given two data points and derivatives $(x_i,\, y_i,\, y_i')$, there exists a unique cubic polynomial in Hermite form matching the data

$$\begin{aligned} \begin{bmatrix} y_1 \\ y_2 \\ y_1' \\ y_2' \end{bmatrix} &= \begin{bmatrix} q(0) \\ q(1) \\ q'(0) \\ q'(1) \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}(0)\boldsymbol{a} \\ \boldsymbol{p}(1)\boldsymbol{a} \\ \boldsymbol{p}(0)\mathbf{D}\boldsymbol{a} \\ \boldsymbol{p}(1)\mathbf{D}\boldsymbol{a} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \boldsymbol{a} = \mathbf{M}^{-1}\boldsymbol{a} \\ \Leftrightarrow \boldsymbol{a} &= \mathbf{M} \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{y}' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_1' \\ y_2' \end{bmatrix}. \end{aligned} \tag{12}$$

This is analogous to (8), and the cubic Hermite form becomes

$$q(x|\boldsymbol{y}, \boldsymbol{y}') = \boldsymbol{p}\left(\frac{x-x_1}{x_2-x_1}\right) \mathbf{M} \begin{bmatrix} y_1 \\ y_2 \\ (x_2-x_1)y_1' \\ (x_2-x_1)y_2' \end{bmatrix} \tag{13}$$

Due to the previous change of variables $s = \frac{x-x_1}{x_2-x_1}$, the slopes $\boldsymbol{y}'$ must be scaled by $(x_2 - x_1)$ with this definition. However, *unit spacing will be assumed* (without loss of generality) going forward to simplify the notation. Using this form allows $q(x)$ to be evaluated using the two values and derivatives nearest x. The smooth interpolation methods "cubic," "makima," "pchip," and "spline" of MATLAB's `interp1` effectively correspond to different choices for the derivative vector $\boldsymbol{y}' \in \mathbb{R}^n$.

B. Spline Construction

This section describes how the spline's derivatives are computed. These n unknowns are determined by the n constraints given from (5) and (6). The $S \in \mathcal{C}^2$ criteria from (5) becomes

$$\begin{aligned} q_i''(1) = q_{i+1}''(0) &\Leftrightarrow \boldsymbol{p}(1)\mathbf{D}^2\mathbf{M} \begin{bmatrix} y_i \\ y_{i+1} \\ y_i' \\ y_{i+1}' \end{bmatrix} = \boldsymbol{p}(0)\mathbf{D}^2\mathbf{M} \begin{bmatrix} y_{i+1} \\ y_{i+2} \\ y_{i+1}' \\ y_{i+2}' \end{bmatrix} \\ &\Leftrightarrow 2y_i' + 8y_{i+1}' + 2y_{i+2}' = 6y_{i+2} - 6y_i \\ &\Leftrightarrow \begin{bmatrix} 1 & 4 & 1 \end{bmatrix} \begin{bmatrix} y_i' \\ y_{i+1}' \\ y_{i+2}' \end{bmatrix} = \begin{bmatrix} -3 & 0 & 3 \end{bmatrix} \begin{bmatrix} y_i \\ y_{i+1} \\ y_{i+2} \end{bmatrix}. \end{aligned} \tag{14}$$

This is a linear system of equations, and the end-conditions in (6) give the initial and final set of equations. E.g., in one end:

$$\begin{aligned} & q_2'''(0) = q_1'''(1) \\ \Leftrightarrow\ & 12y_1 - 12y_2 + 6y_1' + 6y_2' = 12y_2 - 12y_3 + 6y_2' + 6y_3' \\ \Leftrightarrow\ & 12y_1 - 12y_2 + 6y_1' + 6y_2' = 12y_2 - 18y_1 + 6y_3 - 18y_2' - 6y_1' \\ \Leftrightarrow\ & \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} -\frac{5}{2} & 2 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}, \end{aligned} \tag{15}$$

where the substitution $y_3' = 3(y_3 - y_1) - 4y_2' - y_1'$ from (14) was used to make the full linear system tridiagonal. For $n = 4$ data points, the solution for the derivatives $\boldsymbol{y}' \in \mathbb{R}^n$ becomes

$$\boldsymbol{y}' = \mathbf{C}^{-1}\mathbf{P}\boldsymbol{y} \tag{16}$$

$$\text{with } \mathbf{C} = \begin{bmatrix} 1 & 2 & & \\ 1 & 4 & 1 & \\ & 1 & 4 & 1 \\ & & 2 & 1 \end{bmatrix}, \mathbf{P} = \begin{bmatrix} -2.5 & 2 & 0.5 & \\ -3 & 0 & 3 & \\ & -3 & 0 & 3 \\ & -0.5 & -2 & 2.5 \end{bmatrix}.$$

If there are more than four data points, the only change needed is to insert more rows of $\begin{bmatrix} 1 & 4 & 1 \end{bmatrix}$ diagonally in $\mathbf{C}$, and more rows of $\begin{bmatrix} -3 & 0 & 3 \end{bmatrix}$ diagonally in $\mathbf{P}$. The solution $\mathbf{S} = \mathbf{C}^{-1}\mathbf{P}$ can be pre-computed, enabling a fast computation as $\boldsymbol{y}' = \mathbf{S}\boldsymbol{y}$ on the GPU. However, a potentially faster solution is given in the next section.

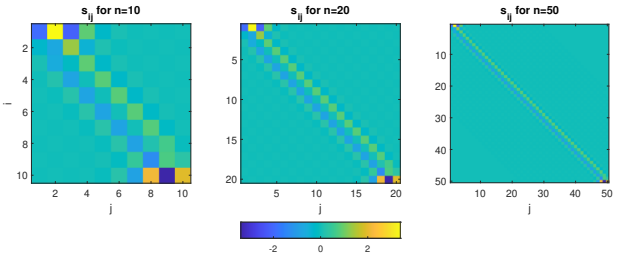

Fig. 1. The matrix $\mathbf{S} \in \mathbb{R}^{n\times n}$ for different n. Figure from [14].

C. Spline Derivatives by Convolution

The middle row of the matrix $\mathbf{S} = \mathbf{C}^{-1}\mathbf{P} \in \mathbb{R}^{n\times n}$ appears to converge for $n \to \infty$ to some rapidly decaying function, as illustrated in Fig. 1. It resembles the matrix representation of convolution and the corresponding kernel for $n \to \infty$ is given by (see Section 2.4 in [15] for a proof)

$$s_i = \frac{\sqrt{3}}{2}\phi^{|i+1|} - \frac{\sqrt{3}}{2}\phi^{|i-1|} \text{ for } i = -k \dots k, \tag{17}$$

where $\phi = -2+\sqrt{3}$ is a root of $1+4x+1x^2$. Since $|\phi| \approx 0.25$, the kernel can be truncated to a finite size: $\boldsymbol{s} \in \mathbb{R}^{2k+1}$, without a significant loss of accuracy. For instance, s_{80} is exactly zero within 32-bit floating-point precision, making $79 \cdot 2 + 1 = 159$ the largest filter size for single-precision data. It is thus faster for some n to compute the derivatives as $\boldsymbol{y}' = \boldsymbol{y} * \boldsymbol{s}$ because the number of operations required per output sample is finite, whereas for $\boldsymbol{y}' = \mathbf{S}\boldsymbol{y}$, the operations required grows with n.

The spline construction from a convolution shows even large splines can be constructed efficiently. The end-elements of $\boldsymbol{y}'$ may still be computed using the finite subset of $\mathbf{S}$ where it does not resemble a convolution to attain both speed and accuracy.

D. Generalizing Spline Interpolation to 2D

For the 2D spline, the inputs are entire images $\mathbf{Y} \in \mathbb{R}^{n\times m}$. The idea is completely analogous to the 1D case. Each segment covers a rectangular subset of 2D space, and since the spline must be smooth, it can be specified in a bicubic Hermite form

$$q(s,t|\mathbf{V}) = \boldsymbol{p}(s)\mathbf{M}\mathbf{V}\mathbf{M}^\top\boldsymbol{p}(t)^\top, \tag{18}$$

where $s = \frac{x-x_{ij}}{x_{ij}-x_{i+1,j+1}}$ and $t = \frac{z-z_{ij}}{z_{ij}-z_{i+1,j+1}}$ are the fractional positions of the pixel, and $\boldsymbol{p}$ and $\mathbf{M}$ are as in (12). The matrix $\mathbf{V}$ supplies the values and derivatives for this piece, such that

$$\begin{bmatrix} q(0,0) & q(0,1) & q'_t(0,0) & q'_t(0,1) \\ q(1,0) & q(1,1) & q'_t(1,0) & q'_t(1,1) \\ q'_s(0,0) & q'_s(0,1) & q''_{st}(0,0) & q''_{st}(0,1) \\ q'_s(1,0) & q'_s(1,1) & q''_{st}(1,0) & q''_{st}(1,1) \end{bmatrix} = \mathbf{V} \tag{19}$$

Thus, all that is needed to evaluate the 2D spline are the axial, lateral, and mixed-direction derivatives. These are found as

$$\mathbf{Y}'_s = \mathbf{S}\mathbf{Y} \tag{20a}$$
$$\mathbf{Y}'_t = \mathbf{Y}\mathbf{S}^\top \tag{20b}$$
$$\mathbf{Y}''_{st} = \mathbf{S}\mathbf{Y}\mathbf{S}^\top, \tag{20c}$$

with $\mathbf{S}$ from (16). This construction can be shown to have 2nd-order smoothness, but the proof (see [14]) is rather tedious.

Spline interpolation can be generalized to higher dimensions as well; however, the memory usage and number of operations to construct the spline grows exponentially. Only $2n$ memories were required to store the 1D values $\boldsymbol{y}$ and derivatives $\boldsymbol{y}'$, but for the 2D spline, $4nm$ memories are required.

This concludes the theory, which showed how using a cubic Hermite form for the spline's segments guarantees smoothness. Any point on the spline can then be evaluated from the nearest values and derivatives, and these derivatives may be computed through matrix multiplications or, in some cases, convolutions.

III. Method

This section describes how the ultrasound data was acquired and processed. An in-vivo dataset provided realistic inputs for the spline interpolation to measure its performance and quality.

A. Data Acquisition

The acquisition setup was similar to that of [8], [9]. In-vivo data from a Sprague-Dawley rat kidney was acquired using a 10 MHz GE L8-18iD probe connected to a Verasonics Vantage 256 scanner. A sequence of 12 defocused emissions was used for each high-resolution image. The emission rate was 5 kHz, resulting in a $5000/12 = 416$ Hz frame rate.

B. Beamforming and Motion Estimation

The beamforming and motion estimation was similar to that of [8], [9]. The RF data from each emission was transformed into a "low-resolution image" (LRI) using a GPU beamformer [18], and each frame was computed as the sum of twelve LRIs

$$\text{HRI}(x,z) = \sum_{i=1}^{12} \text{LRI}_i(x,z), \tag{21}$$

where HRI is the high-resolution image. A transverse oscillation auto-correlation motion estimator [19] was used to find the motion between consecutive HRIs, and the resulting vector fields were accumulated to get the total displacement over time

$$\boldsymbol{d}_t(x,z) = \sum_{i=1}^{t} \text{MotionEst}_i(x,z), \tag{22}$$

where MotionEst_t is the estimated motion field between frame t and $t-1$, corresponding to the displacement at time $t-0.5$. The final motion-corrected imaging was then

$$\text{HRI}^*(x,z) = \sum_{i=1}^{12} \text{LRI}_i(x - d_t^x(x,z), z - d_t^z(x,z)). \tag{23}$$

The displacements must be found at time $t = i/12 + 0.5$ since the time between LRIs is one-twelfth of a frame. A 1D spline interpolation was used to find $\boldsymbol{d}_t(x,z) = [d_t^x(x,z),\ d_t^z(x,z)]$ at these twelve points in time. The result of initial processing was thus twelve LRIs and twelve sets of positions to evaluate for each frame, as illustrated in the MATLAB code below.

TABLE I
OVERVIEW OF THE HARDWARE/SOFTWARE SETUP

CPU	Intel Xeon W-3223
GPU	NVIDIA GeForce RTX 4090
Software	CUDA 12.4, MATLAB R2022b, Ubuntu 20.04.1

```
HRI = 0;
for i = 1:12
    HRI = HRI + interp2(LRI(:,:,i), ...
        x_position(:,:,i), ...
        z_position(:,:,i), 'spline', 0);
end
```

Listing 1. The MATLAB implementation of motion correction.

The images and positions were of size $N_z \times N_x = 936 \times 655$.

C. Measurement of Performance

The rate for motion correction was measured by wrapping the code in Listing 1 in a function and measuring its execution time using the `timeit` function. An experiment was repeated 15 times, where 10 HRIs were motion-corrected each time, and the processing rate in terms of LRIs/s was then calculated as

$$R = \frac{1}{15}\sum_{i=1}^{15}\frac{10 \cdot 12}{T_i}, \tag{24}$$

where T_i was the 15 execution times reported by `timeit`. To measure the proposed implementation, the call to `interp2` in Listing 1 was replaced with a call to the GPU-accelerated spline interpolation "`interp2gpu`", and the processing time was measured with the `gputimeit` function. The execution environment for these measurements is summarized in Table I.

D. Measurement of Correctness

The rounding errors of the GPU spline implementation will differ from those of MATLAB's interp2 because it depends on the order of additions and multiplications, which is difficult to replicate without the source code. The relative error is defined

$$\varepsilon_{\text{dB}} = 20 \cdot \log_{10}\left(\frac{|y - y_{ref}|}{|y_{ref}|}\right). \tag{25}$$

To claim that the implementation is "correct," it is required that its errors match interp2 in distribution. The values from interp2 were computed in double precision to provide a reference y_{ref}.

E. Implementation Details

To evaluate the spline, its derivatives must first be computed. The images were complex-valued, but $\mathbf{S}$ in (20) is real, and it was found to be faster to then compute the real and imaginary parts separately. For example, for the real part: $\Re\mathbf{Y}'_s = \mathbf{S}\Re\mathbf{Y}$. The reason is likely because the linear algebra library cuBLAS does not support real-times-complex matrix products, but the real-times-real product $\Re\mathbf{Y}'_s = \mathbf{S}\Re\mathbf{Y}$ is supported in cuBLAS.

When computing the spline derivatives using a convolution (Section II-C), the filter $\boldsymbol{s}$ was truncated to size $2 \cdot 15 + 1 = 31$ for single precision inputs, as this appeared to be sufficient.

TABLE II
PERFORMANCE RESULTS FOR INTERPOLATION METHODS.

	Method	Processing Rate [Hz]
MATLAB's interp2	Linear	18406
	Cubic	20358
	Spline	**7.5**
Proposed GPU implementation	**Spline**	**5013**
	Spline (Conv.-based)	6337

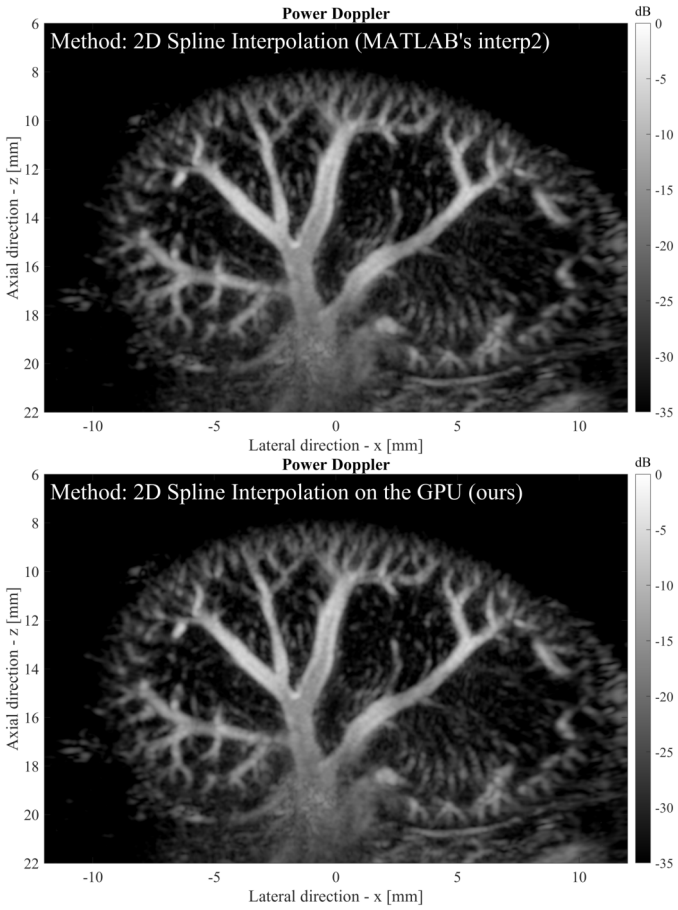

Fig. 2. The results after using each implementation for motion correction. An SVD-based clutter filtering was used to remove the then stationary tissue signal [20], and the resulting blood speckle images were summed incoherently to produce the Power Doppler images shown.

IV. RESULTS AND DISCUSSION

The measured motion-correction rates are shown in Table II, where the proposed implementation was 668 times faster than MATLAB's interp2 spline, which lacked GPU support. Thus, real-time motion correction was achieved at a rate of 5013 Hz using GPU-accelerated spline interpolation. The convolution-based spline (see Section II-C) was even faster at 6337 LRI/s.

As accuracy is crucial, the output from the proposed GPU implementation was compared to that of MATLAB's interp2. It can be seen from Figure 2 that the results from using each

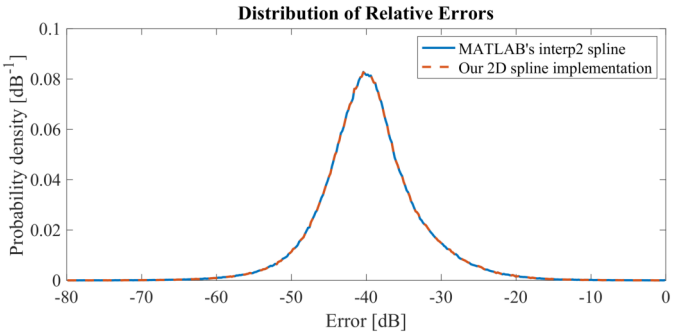

Fig. 3. The distribution of relative errors (pixel-wise) computed as a histogram with 401 bins. It shows both implementations have the same error distribution.

implementation are virtually identical. Further, Figure 3 shows that both implementations had the same distribution of errors. Lastly, the greatest difference $|y_{proposed} - y_{interp2}|/|y_{ref}|$ was equal to -78.5 dB, making it insignificant for most purposes.

The implementation made motion correction, which was the slowest part of the SURE pipeline [8], run in real-time [21].

A. Further Optimized Construction

A lot of time is spent constructing the spline as time must be spent splitting the data $\mathbf{Y}$ into its real and imaginary parts and then carrying out the products in (20) for the parts separately. This can potentially be performed efficiently using low-level cuBLAS calls. For instance, the gemmStridedBatched matrix multiply function allows one to specify the access stride, where a stride of 2 in $\mathbf{Y} \in \mathbb{C}^{n \times m}$ means every second element will be skipped. Since complex values are stored as real pairs, one can re-interpret the input as $\mathbf{Y} \in \mathbb{R}^{2n \times m}$, such that an access stride of two then corresponds to selecting either $\Re\mathbf{Y}$ or $\Im\mathbf{Y}$. Thus, it is possible to multiply $\mathbf{S}$ with the real/imaginary parts of $\mathbf{Y}$ without splitting $\mathbf{Y}$ into two arrays, using cuBLAS calls.

V. Conclusion

The proposed spline interpolation enabled motion correction to be performed in real-time at the rate of 5013 Hz, which was 668 times faster than MATLAB's interp2 spline interpolation. Moreover, the accuracy of the proposed implementation was similar to MATLAB's interp2, with minor differences in output likely due to differences in floating-point rounding errors. The code can be found at https://github.com/sebftw/interp2gpu.

References

[1] O. Couture, B. Besson, G. Montaldo, M. Fink, and M. Tanter, "Microbubble ultrasound super-localization imaging (MUSLI)," in *Proc. IEEE Ultrason. Symp.*, 2011, pp. 1285–1287.

[2] O. M. Viessmann, R. J. Eckersley, K. Christensen-Jeffries, M. X. Tang, and C. Dunsby, "Acoustic super-resolution with ultrasound and microbubbles," *Phys. Med. Biol.*, vol. 58, pp. 6447–6458, 2013.

[3] K. Christensen-Jeffries, R. J. Browning, M. Tang, C. Dunsby, and R. J. Eckersley, "In vivo acoustic super-resolution and super-resolved velocity mapping using microbubbles," *IEEE Trans. Med. Imag.*, vol. 34, no. 2, pp. 433–440, 2015.

[4] C. Errico, J. Pierre, S. Pezet, Y. Desailly, Z. Lenkei, O. Couture, and M. Tanter, "Ultrafast ultrasound localization microscopy for deep super-resolution vascular imaging," *Nature*, vol. 527, pp. 499–502, 2015.

[5] K. Christensen-Jeffries, O. Couture, P. A. Dayton, Y. C. Eldar, K. Hynynen, F. Kiessling, M. O'Reilly, G. F. Pinton, G. Schmitz, M. Tang *et al.*, "Super-resolution ultrasound imaging," *Ultrasound Med. Biol.*, vol. 46, no. 4, pp. 865–891, 2020.

[6] A. Bar-Zion, O. Solomon, C. Rabut, D. Maresca, Y. C. Eldar, and M. G. Shapiro, "Doppler slicing for ultrasound super-resolution without contrast agents," *bioRxiv preprint*, pp. 1–10, 2021.

[7] Q. You, M. R. Lowerison, Y. Shin, X. Chen, N. V. C. Sekaran, Z. Dong, D. A. Llano, M. A. Anastasio, and P. Song, "Contrast-free super-resolution power doppler (cs-pd) based on deep neural networks," *IEEE Trans. Ultrason. Ferroelec. Freq. Contr.*, 2023.

[8] J. A. Jensen, M. A. Naji, S. K. Præsius, I. Taghavi, M. Schou, L. N. Hansen, S. B. Andersen, C. M. Sørensen, M. B. Nielsen, C. Gundlach, H. M. Kjer, A. Dahl, B. G. Tomov, M. Ommen, and E. V. Thomsen, "Super-resolution ultrasound imaging using the erythrocytes—Part I: Density images," *IEEE Trans. Ultrason. Ferroelec. Freq. Contr.*, vol. 71, no. 8, pp. 925–944, 2024.

[9] M. A. Naji, I. Taghavi, M. Schou, S. K. Præsius, L. N. Hansen, S. B. Andersen, S. B. Søgaard, N. Pandru, M. B. Nielsen, H. M. Kjer, B. G. Tomov, A. B. Dahl, C. M. Sørensen, and J. A. Jensen, "Super-resolution ultrasound imaging using the erythrocytes—Part II: Velocity images," *IEEE Trans. Ultrason. Ferroelec. Freq. Contr.*, vol. 71, no. 8, pp. 945–959, 2024.

[10] V. Hingot, C. Errico, M. Tanter, and O. Couture, "Subwavelength motion-correction for ultrafast ultrasound localization microscopy," *Ultrasonics*, vol. 77, pp. 17–21, 2017.

[11] K. B. Hansen, C. A. Villagomez-Hoyos, J. Brasen, K. Diamantis, V. Sboros, C. M. Sørensen, and J. A. Jensen, "Robust microbubble tracking for super resolution imaging in ultrasound," in *Proc. IEEE Ultrason. Symp.*, 2016, pp. 1–4.

[12] S. Harput, K. Christensen-Jeffries, J. Brown, Y. Li, K. J. Williams, A. H. Davies, R. J. Eckersley, C. Dunsby, and M. Tang, "Two-stage motion correction for super-resolution ultrasound imaging in human lower limb," *IEEE Trans. Ultrason. Ferroelec. Freq. Contr.*, vol. 65, no. 5, pp. 803–814, 2018.

[13] I. Taghavi, S. B. Andersen, C. A. V. Hoyos, M. B. Nielsen, C. M. Sørensen, and J. A. Jensen, "In vivo motion correction in super resolution imaging of rat kidneys," *IEEE Trans. Ultrason. Ferroelec. Freq. Contr.*, vol. 68, no. 10, pp. 3082–3093, 2021.

[14] S. K. Præsius, "Real-time SURE image processing," Master's thesis, The Technical University of Denmark, 2022.

[15] J. Ahlberg, E. Nilson, and J. Walsh, Eds., *The Theory of Splines and Their Applications*, ser. Mathematics in Science and Engineering. Elsevier, 1967, vol. 38.

[16] E. Waring, "Problems concerning interpolations," *Philos. Trans. R. Soc.*, vol. 69, pp. 59–67, 1779.

[17] N. Macon and A. Spitzbart, "Inverses of vandermonde matrices," *The American Mathematical Monthly*, vol. 65, no. 2, pp. 95–100, 1958.

[18] M. B. Stuart, P. M. Jensen, J. T. R. Olsen, A. B. Kristensen, M. Schou, B. Dammann, H. H. B. Sørensen, and J. A. Jensen, "Real-time volumetric synthetic aperture software beamforming of row-column probe data," *IEEE Trans. Ultrason. Ferroelec. Freq. Contr.*, vol. 68, no. 8, pp. 2608–2618, 2021.

[19] C. Kasai, K. Namekawa, A. Koyano, and R. Omoto, "Real-Time Two-Dimensional Blood Flow Imaging using an Autocorrelation Technique," *IEEE Trans. Son. Ultrason.*, vol. 32, no. 3, pp. 458–463, 1985.

[20] C. Demene, T. Deffieux, M. Pernot, B.-F. Osmanski, V. Biran, J.-L. Gennisson, L.-A. Sieu, A. Bergel, S. Franqui, J.-M. Correas, I. Cohen, O. Baud, and M. Tanter, "Spatiotemporal clutter filtering of ultrafast ultrasound data highly increases Doppler and fUltrasound sensitivity," *IEEE Trans. Med. Imag.*, vol. 34, no. 11, pp. 2271–2285, 2015.

[21] S. K. Præsius, M. B. Stuart, M. Schou, B. Dammann, H. H. B. Sørensen, and J. A. Jensen, "Real-time super-resolution ultrasound imaging using GPU acceleration," in *Proc. IEEE Ultrason. Symp.*, 2022, pp. 1–4.