Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration

Orestis Zachariadis[a,*], Andrea Teatini[b,c], Nitin Satpute[a], Juan Gómez-Luna[d], Onur Mutlu[d], Ole Jakob Elle[b,c], Joaquín Olivares[a]

[a]*Department of Electronics and Computer Engineering, Universidad de Cordoba, Córdoba, Spain*
[b]*The Intervention Centre, Oslo University Hospital - Rikshospitalet, Oslo, Norway*
[c]*Department of Informatics, University of Oslo, Oslo, Norway*
[d]*Department of Computer Science, ETH Zurich, Zurich, Switzerland*

Abstract

Background and Objective. B-spline interpolation (BSI) is a popular technique in the context of medical imaging due to its adaptability and robustness in 3D object modeling. A field that utilizes BSI is Image Guided Surgery (IGS). IGS provides navigation using medical images, which can be segmented and reconstructed into 3D models, often through BSI. Image registration tasks also use BSI to transform medical imaging data collected before the surgery and intra-operative data collected during the surgery into a common coordinate space. However, such IGS tasks are computationally demanding, especially when applied to 3D medical images, due to the complexity and amount of data involved. Therefore, optimization of IGS algorithms is greatly desirable, for example, to perform image registration tasks intra-operatively and to enable real-time applications. A traditional CPU does not have sufficient computing power to achieve these goals and, thus, it is preferable to rely on GPUs. In this paper, we introduce a novel GPU implementation of BSI to accelerate the calculation of the deformation field in non-rigid image registration algorithms.

Methods. Our BSI implementation on GPUs minimizes the data that needs to be moved between memory and processing cores during loading of the input grid, and leverages the large on-chip GPU register file for reuse of input values. Moreover, we reformulate our method as trilinear interpolations to reduce computational complexity and increase accuracy. To provide pre-clinical validation of our method and demonstrate its benefits in medical applications, we integrate our improved BSI into a registration workflow for compensation of liver deformation (caused by pneumoperitoneum, i.e., inflation of the abdomen) and evaluate its performance.

Results. Our approach improves the performance of BSI by an average of 6.5× and interpolation accuracy by 2× compared to three state-of-the-art GPU implementations. Through pre-clinical validation, we demonstrate that our optimized interpolation accelerates a non-rigid image registration algorithm, which is based on the Free Form Deformation (FFD) method, by up to 34%.

Conclusion. Our study shows that we can achieve significant performance and accuracy gains with our novel parallelization scheme that makes effective use of the GPU resources. We show that our method improves the performance of real medical imaging registration applications used in practice today.

Keywords:
Medical Image Registration, Medical Image Processing, Parallel Computing, GPU, B-splines

1. Introduction

Image Guided Surgery (IGS) aims to provide surgeons with navigation capabilities to perform safer surgeries through better visualization [1]. IGS is created by combining medical images, such as Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) [2], with surgical instrument tracking technologies [3]. However, the accuracy of image guided surgery is often undermined by organ deformations, especially in soft tissue surgeries. These deformations are difficult to account for due to their non-linear behaviour. Non-rigid registration is a technique that has been developed to reproduce and model such non-linear deformations [4].

Non-rigid registration through Free Form Deformation (FFD) [5], based on cubic B-spline interpolation (BSI) [5, 6], is a state-of-the-art technique for non-rigid registration. FFD works by manipulating a grid of control points. The shape of a 3D object (e.g., an organ) underlying the control points can be changed by using a smooth and C^2 continuous transform (i.e., continuous up to second order derivatives). FFD uses BSI in the calculation of the deformation field.

BSI is one of the most computationally demanding parts of FFD [7]. Graphics Processing Units (GPUs) can help achieve the real-time requirements of IGS, namely FFD, as they offer massive computational performance in comparison

[*] Corresponding author
Email addresses: `orestis.zachariadis@uco.es` (Orestis Zachariadis), `andre_tea@outlook.com` (Andrea Teatini)

to Central Processing Units (CPUs). GPUs deploy thousands of execution threads, which operate on large batches of data. GPUs provide higher throughput and power-efficiency than CPUs on multithreaded workloads [8]. The performance of medical imaging applications benefits significantly from GPUs [9, 10, 11, 12, 13, 14, 15].

For these reasons, several authors have used GPUs for BSI [6, 16, 17, 18, 19]. Sigg et al. [16] and Ruijters et al. [17] achieve a substantial reduction in the number of input samples by representing the weighted sums as trilinear interpolations. More recently, Ellingwood et al. [6] and Du et al. [18] use GPU implementations of BSI to improve the performance of image registration. They improve input sample loading by aligning the control grid with the voxel grid of the volume [6, 18, 19]. However, all these works suffer from the intensive *data movement* of a large number of input samples between the memory and the GPU, which is the main performance bottleneck of BSI implementations on a GPU [16].

Our goal in this work is to accelerate BSI on GPUs by alleviating the data movement bottleneck with optimization techniques that enable a more efficient use of the on-chip memory resources. To this end, we propose a GPU implementation of BSI with three key optimizations: a) a new workload partitioning scheme for GPU execution threads that reduces the number of memory accesses, b) a register-tiling approach that keeps input data close to the execution units, and c) the replacement of weighted summation with linear interpolations, which reduces the computational load and increases the accuracy.

In order to show how our approach affects the performance and accuracy of image registration in a realistic scenario, we integrate our technique (publicly available[1]) to the FFD registration of NiftyReg [7]. NiftyReg is a lightweight medical image registration library. Recent works [20, 21] use NiftyReg as a reference for registration.

We complete our study with a pre-clinical evaluation of our method. We use FFD with our GPU-accelerated BSI on 1) CT scans of patient-specific liver phantom, and 2) MRI scans of a porcine liver model to compensate for a non-rigid soft tissue deformation caused by pneumoperitoneum. Pneumoperitoneum is a surgical procedure to inflate the patient's abdomen, which is necessary for any abdominal laparoscopic surgery. Pneumoperitoneum, however, deforms the shape of the organs [22, 23]. To account for this deformation, we capture new images during the surgery (intra-operative) and use non-rigid image registration to match them with images before pneumoperitoneum (pre-operative images). We compute non-rigid image registration for pneumoperitoneum with state-of-art implementations and with our BSI implementation. Using our implementation results in a performance increase with the same accuracy as using the state-of-the-art implementations.

2. Background

In this section, we first introduce the foundations of B-spline interpolation. Since our GPU implementation of BSI is specific to 3D medical images (CT, MRI, or US volumes), formulations and analysis focus on the 3D case. Second, we review two state-of-the-art implementations of BSI on GPUs.

2.1. B-spline interpolation theory

We introduce B-spline interpolation for 3D images, i.e., the domain of the image volume is in the x, y, z coordinate space. As Equation 1 shows [5, 6], the BSI transformation of FFD for each voxel (i.e., each interpolated point of FFD) with coordinates x, y, z is $T(x, y, z)$. The BSI transformation is a function of control points $\phi_{i,j,k}$, which are arranged into a grid of dimensions $n_x \times n_y \times n_z$. The control point grid is uniformly spaced, with δ_x, δ_y, and δ_z being the spacing (in voxels) in the three dimensions.

$$T(x, y, z) = \sum_{l=0}^{3} \sum_{m=0}^{3} \sum_{n=0}^{3} B_l(u) B_m(v) B_n(w) \phi_{i+l,j+m,k+n} \qquad (1)$$

where $i = \lfloor x/\delta_x \rfloor - 1, j = \lfloor y/\delta_y \rfloor - 1, k = \lfloor z/\delta_z \rfloor - 1, u = x/\delta_x - \lfloor x/\delta_x \rfloor, v = y/\delta_y - \lfloor y/\delta_y \rfloor, w = z/\delta_z - \lfloor z/\delta_z \rfloor$, B are the scalar B-spline coefficients [17] and ϕ are the control points. Each voxel is affected by four control points in each dimension. Thus, in a 3D space, 4 × 4 × 4 control points, forming a cube (see Figure 1), affect the inner *tile* of voxels. In general, in N-dimensional images, 4^N control points affect each voxel.

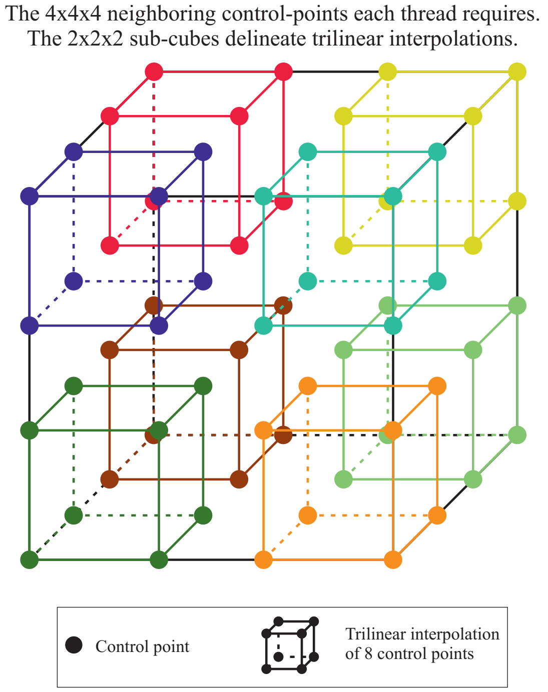

The 4x4x4 neighboring control-points each thread requires.
The 2x2x2 sub-cubes delineate trilinear interpolations.
● Control point
Trilinear interpolation of 8 control points

Figure 1: The cube of 4 × 4 × 4 control points that affect a voxel/tile in a 3D control point grid. Smaller cubes depict the grouping in trilinear interpolations.

2.1.1. Tiles

Tiles are logical groups of voxels that share common properties. Based on Equation 1, we define tiles of $\delta_x \times \delta_y \times \delta_z$ dimensions. Figure 2 illustrates a tile in a 2D example. We make

[1] https://github.com/oresths/niftyreg_bsi

two observations: 1) the *same* control points, i.e., the ones surrounding the tile, affect all voxels inside the tile, and 2) control points of neighboring tiles overlap.

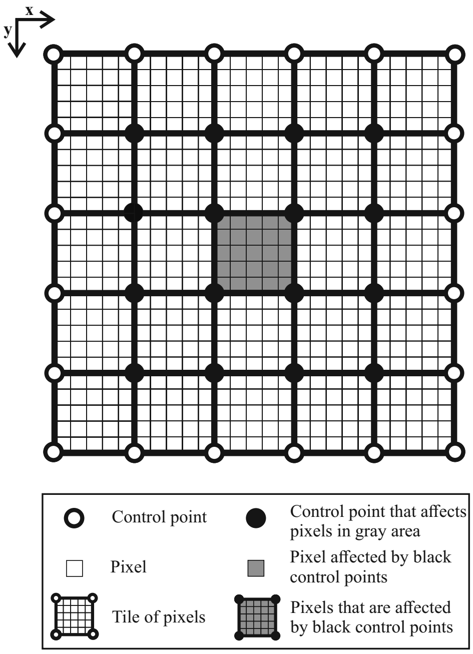

x
y
Control point
Control point that affects pixels in gray area
Pixel
Pixel affected by black control points
Tile of pixels
Pixels that are affected by black control points

Figure 2: A 2D space divided into tiles.

From the implementation perspective, partitioning a volume into tiles is a way of exploiting data reuse (i.e., reuse of control points) in on-chip memories, when calculating the interpolated voxels. Thus, *tiling* saves memory traffic between off-chip and on-chip memories.

2.2. State-of-the-art GPU implementations of BSI

This section introduces the two state-of-the-art BSI methods and their respective GPU implementations, which we use as comparison points for our work.

Texture Hardware (TH). Ruijters et al. [17, 24] provide a texture hardware method for BSI. They base their method on the observation that the weighted additions of Equation 1 can be replaced by a linear interpolation [16, 17]. Linear interpolations are well-suited for the GPU texture unit, that features a hardware interpolation unit. The hardware interpolation unit calculates the interpolation directly and it does not require separate accesses to off-chip *global* memory of the GPU to load the input control points. Hardware interpolation is fast but it has two main drawbacks. First, it has only 8 bits of accuracy [8], which limits the resolution of the interpolation. Second, the values that the hardware interpolation unit fetches from the off-chip memory are a function of the absolute position of each voxel. Therefore, TH cannot utilize custom caching schemes to aggregate data transfers for neighboring voxels (Appendix A). Texture Hardware BSI is included in an easy-to-use library by Ruijters et al. [24] and is used in recent works [25, 26].

Thread per Voxel (TV). This method assigns one thread per image element, e.g., per voxel in the case of 3D images.

Ellingwood et al. [6] present a GPU implementation of this method that applies tiling (Section 2.1.1). They assign one or more *thread blocks* to each tile, with one thread for each voxel of the tile. Tiling enables the reuse of control points, which are the same for the whole tile, by keeping them in the fast on-chip *shared* memory.

NiftyReg [7], a lightweight open-source medical image registration library, also uses the thread per voxel method. NiftyReg contains optimized implementations of BSI for both CPUs and GPUs. It is open-source and well-maintained, with competitive performance against other state-of-the-art implementations [20, 21]. The GPU implementation uses a simple, straightforward TV method, which does not take advantage of tiling. The CPU implementation, however, exploits tiling by applying multi-core and vectorization optimizations.

3. Optimizing B-spline interpolation

This section presents our GPU implementation of BSI, which follows a different approach to the state-of-the-art implementations (i.e., TH and TV). In our approach, we assign *one thread per tile* of voxels, as we explain in Sections 3.1 to 3.3. In Section 3.5, we introduce our implementations for CPU, which follow the GPU approach partially.

3.1. Overview of our GPU implementation of BSI

Our GPU implementation of BSI is based on two key ideas.

First, an entire tile of voxels is assigned to a single GPU thread (*Thread per Tile*, TT), in contrast to the one-thread, one-voxel approach. This TT assignment takes advantage of tiling in both on-chip *cache* memory and registers: 1) tiling in cache memory minimizes the reads from off-chip memory, by maximizing the overlap of input control points, and 2) tiling in registers minimizes the accesses to cache memory, by reusing the input control points for many voxels.

Second, we replace the weighted sum of the basic formula of BSI with trilinear interpolations, in a similar way as TH does. We calculate these trilinear interpolations using Fused Multiply-Add (FMA) instructions, which the GPU instruction set contains [8]. FMA increases both accuracy and speed in regard to regular multiplication and addition instructions.

We give an in-depth description of our optimizations in the next sections.

3.2. Thread per Tile (TT)

In this section, we describe the optimization techniques that we deploy in our TT approach to BSI. We show how the input loading and register optimizations reduce memory accesses.

3.2.1. Input loading optimization

The main idea is to reduce loads from global memory by taking advantage of the overlap of tiles assigned to neighboring threads. Figure 3 compares the TV approach with tiling (left), explained in Section 2.2, to our TT approach (right).

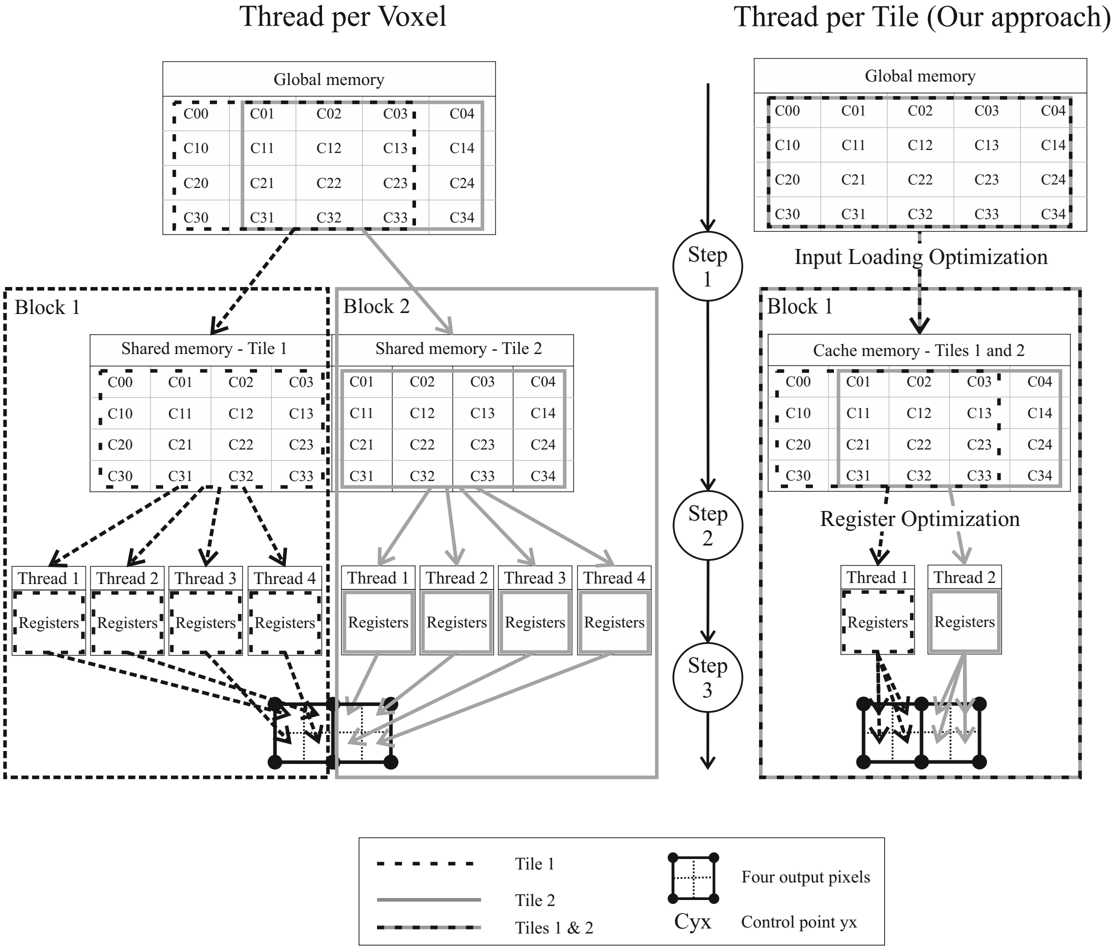


Figure 3: Comparison of input loading and register optimization for Thread per Voxel with tiling (left) and Thread per Tile (right) for two neighboring tiles.

In TV, each block of threads works on a unique tile of voxels. Thus, each block requires 4^N input control points (Section 2.1.1). Therefore, for each tile, we need to move 4^N control points from global memory to shared memory. *Step 1* in Figure 3 (left) illustrates the required data movement from global memory to shared memory for a 2D example. In this example, we have two tiles and each tile is assigned to one block. The two tiles imply the movement of $4 \times 4 + 4 \times 4$ control points from global memory to shared memory.

In TT, we assign one thread per tile to take advantage of overlapping neighboring tiles. *Step 1* of Figure 3 (right) illustrates the reduction in data movement to cache memory, with the overlap in the x-direction. Two tiles require only $4{\times}5$ control points. In 3D medical images, the reduction in data movement is more noticeable, because there is overlap in the three directions. As a result, our approach reduces the data movement from global memory dramatically. TT requires about 12× and about 187× (for $5 \times 5 \times 5$ tiles) fewer memory transfers in comparison to TV and TH (Appendix A).

3.2.2. Register optimization

The second optimization technique that we apply to TT is based on two main ideas: 1) we load the control points for all voxels of the tile from cache memory only once, and 2) we keep the loaded control points in registers, which are the fastest on-chip memory, until thread execution finishes.

In TV, threads belonging to the same block work on individual voxels of the same tile. For every voxel belonging to the tile, the corresponding threads need to access exactly the same control points as all other threads of the block. *Step 2* of Figure 3 (left) illustrates the required data movement from shared memory to registers for a 2D example. In this example, each pixel is assigned to one thread, and for every four pixels the corresponding four threads need to read (from shared memory to registers) sixteen control points each (i.e., 4×16 reads for every four pixels).

In TT, the one-thread, one-tile assignment minimizes the data movement between cache memory and registers. For all voxels belonging to the tile, the corresponding thread needs to access from cache memory a unique set of control points that is different from the set accessed by any other thread of the block (there is overlap, though). By utilizing register tiling, the thread keeps the control points in registers, which are faster than cache memory [27], to process every voxel in the tile. *Step 2* of Figure 3 (right) illustrates the reduction in data movement. For every four pixels, the corresponding thread needs to read only sixteen control points (i.e., 1×16 reads for every four pixels).

3.3. Thread per Tile with Linear Interpolations (TTLI)

We extend TT by reformulating the triple sum of Equation (1) to *trilinear* interpolations. The basic idea is that a linear interpolation can replace an addition of two weighted addends. We can extend this to three dimensions, where we combine eight addends into a trilinear interpolation [16].

We calculate a trilinear interpolation as a combination of seven linear interpolations (in our implementation, we do not use the hardware interpolation unit as this would prevent us from increasing input data locality and output data accuracy (Section 2.2)). The linear interpolations are beneficial to the performance of our approach because the compiler maps linear interpolations to FMA instructions. FMA instructions are preferable for two reasons. First, FMA is more accurate because it executes multiplication and addition in the same step, with a single rounding. Second, FMA is faster because it executes both multiplication and addition with a single instruction [28].

Figure 1 illustrates the 4 × 4 × 4 neighborhood of control points that affect a tile of voxels. Each one of the 2 × 2 × 2 colored sub-cubes of control points corresponds to one trilinear interpolation. For each voxel in the tile, the respective thread calculates each one of the eight trilinear interpolations. The arithmetic operations that are needed for each trilinear interpolation (i.e., colored sub-cube) are independent, thus enabling Instruction Level Parallelism (ILP) [27].

3.4. Implementation details of TT and TTLI

Register tiling, which we employ in our approach, requires a careful management of the registers. We explain some of our implementation decisions in the following paragraphs.

Register allocation. The deformation field of a 3D image requires 64 control points and each control point comprises three values, one for each of the three coordinates (x, y, z). Therefore, we need $3 \cdot 64 = 192$ registers for the control points only. The control point grid is aligned to the voxel grid and uniformly spaced, therefore we store the scalar B-spline coefficients in Look-Up-Tables (LUTs). TT requires 235 registers in total, whereas TTLI requires 255 registers.

Thread block configuration. The amount of required registers limits the maximum active threads per Symmetric Multiprocessor (SM) to 256 [8]. We arrange threads to blocks of 4 × 4 × 4 threads. We select this arrangement because a cube is the geometrical structure that maximizes overlap and consequently minimizes memory transfers (i.e., minimizes Equation A.4 in Appendix A).

Performance at low occupancy. Shared and cache memories are slower than registers, therefore TT keeps the control points in registers permanently. We arrange input data in such a way that there are no spills (although in TTLI we have to store a few control points into shared memory). Due to the large amount of registers our approach requires, the occupancy of the GPU falls to 12.5% for CUDA Compute Capabilities (CC) before 7.x and to 25% for newer CC [8]. Despite the low occupancy, we can maximize resource utilization by using ILP and avoiding the use of cache memories. Our approach uses a register-only approach to increase the performance substantially [27].

3.5. Application of our approach to CPUs

We can apply our TTLI approach to the CPU implementation of BSI. Table 1 summarizes the main differences with the GPU implementations. Some optimizations are not fully applicable to the CPU implementation, because they are tailored to the GPU architecture. GPUs allow for more fine-grained parallelism in comparison to CPU, which makes GPUs more efficient with small 3D groups of tiles with regards to cache and register management. We develop two parallel implementations of BSI on CPUs, which take advantage of the several cores and the SIMD units (SSE/AVX) that CPUs have [29, 30]. SIMD units pack many single values, which we call *elements*, in a special register, called a *vector*, thus applying a form of register tiling.

Table 1: Differences between GPU and CPU implementations (✓ means that an optimization technique is used in the CPU implementation).

Optimization	VT	VV
Input overlap	Only in x-direction	Only in x-direction
Register tiling	Partially	✓
Linear interpolation	✓	✓

Vector per Tile (VT). In this method, we parallelize by using SIMD vectors to simultaneously process many voxels of a tile. Each thread processes δ_x voxels simultaneously. We iterate through the y,z-dimensions of the tile, δ_x voxels at a time. The drawback of this method is that a SIMD vector is not fully utilized if δ_x, a user configurable parameter, is not a multiple of the SIMD vector length.

Vector per Voxel (VV). In this method, we parallelize by using SIMD vectors to simultaneously process each of the trilinear interpolations a single voxel requires. This means that, using the SIMD unit, each thread processes simultaneously all colored sub-cubes a voxel requires (Figure 1). Conveniently, the SIMD vector length is equal to the number of sub-cubes.

4. Pre-clinical dataset acquisition

In order to test our implementations of BSI in a pre-clinical application scenario, we perform a pre-clinical study where we use FFD. We create a dataset (publicly available) [31] which consists of two sets of subjects and imaging modalities: 1) a patient-specific liver phantom [32] with DynaCT scanning, and 2) a porcine model with MRI scanning, to validate the registration process in-vivo. Table 2 lists the characteristics of the collected dataset.

In this section, we describe the dataset in detail. We present evaluation results in Sections 6 and 7.

Table 2: Image characteristics.

Registration pair	Resolution	Voxel count (millions)	Voxel Spacing
Phantom1	512×228×385	44.94	0.49×0.49×0.49
Phantom2	294×130×208	7.95	0.90×0.90×0.90
Phantom3	294×130×208	7.95	0.90×0.90×0.90
Porcine1	303×167×212	10.73	0.94×0.94×1.00
Porcine2	267×169×237	10.70	0.94×0.94×1.00

Patient-specific phantom of liver. The patient-specific liver phantom presents a total of five tumors and a blood vessel tree. The liver phantom used in our experiments was produced by the ARTORG centre and Cascination® [32] and has been used by Teatini et al. for registration studies [33]. We performed three intra-operative CT scans (Artis Zeego, Siemens®) (DynaCT) of the liver phantom in the OR. For each scan, we apply non-rigid deformations to the phantom, which we try to correct through FFD (*Phantom 1, Phantom 2, Phantom 3*). An example of the liver phantom scans is visible in Figure 4a and Figure 4b.

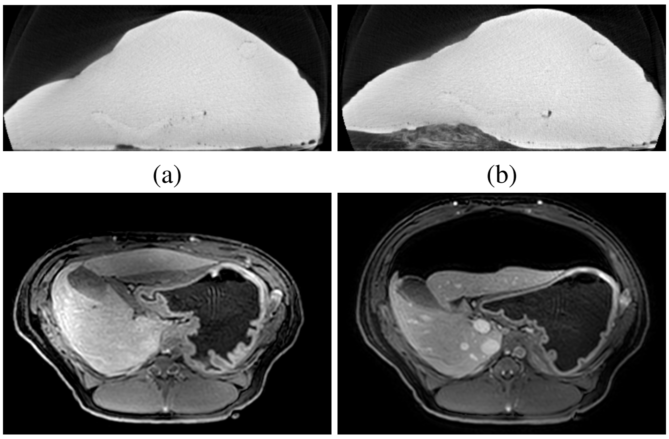
Porcine model. We performed a porcine study to acquire pre-operative (without pneumoperitoneum) and intra-operative (post pneumoperitoneum) MRI scans. These were used to study the deformation that the liver undergoes due to pneumoperitoneum alone. We performed this study at Oslo University Hospital through the use of a 3T Siemens MRI scanner, model Ingenia Philips ® [34]. We performed pneumoperitoneum at 14 mmHg. Both MRI scans were performed with injection of contrast, as done in patients, to improve imaging of the liver parenchyma and blood vessels (Flow rate 5.0 and Volume 11.0, based on the weight of the animal at 55kg). The MRI scans are thin sliced (1.5 mm in *Porcine 1* and 1 mm in *Porcine 2*) enhanced-T1 high-resolution isotropic volume examination (e-THRIVE) scans. The deformation of the liver due to pneumoperitoneum is visible in the differences between images (c) and (d) in Figure 4 and further explored in [35].

5. B-spline interpolation evaluation

In this section, we evaluate our BSI implementations on GPUs and CPUs in terms of performance and accuracy, and compare them to state-of-the-art implementations.

5.1. Evaluation methodology

Configuration. In our evaluation, we use one CPU and two GPUs. The CPU is a quad-core Intel i7-7700HQ@2.8 GHz with HyperThreading. We use gcc v5.4 compiler. To show the performance and stability among different GPU generations, we use two GPUs of different generations: 1) NVIDIA GeForce GTX 1050 (with Pascal architecture [8]), and 2) NVIDIA GeForce RTX 2070 (with Turing architecture [36]). We use CUDA SDK v9.2 for the first GPU and v10.1 for the second GPU. We use CUDA event API to acquire the timing results.



(a) (b)

(c) (d)

Figure 4: Medical images used for pre-clinical evaluation of our optimized image registration through FFD. (a) and (b) show two DynaCT scans of the liver phantom, and (c) and (d) are MRI scans of the porcine model, respectively without (c) and with pneumoperitoneum applied (d).

Comparison baseline. We compare our approaches to the state-of-the-art BSI implementations (Section 2.2). For TH, we use the library from Ruijters et al. [24]. For TV, we create an implementation that is based on the recent literature [6, 7, 19]. This implementation of TV uses tiling and is tuned for the GPUs we use. We refer to this implementation as *TV-tiling*. We also compare to the optimized GPU implementation of the NiftyReg library [7], which does *not* use tiling, as GPU reference, and the optimized CPU implementation of NiftyReg [7] as CPU reference. We refer to the NiftyReg implementations as *NiftyReg (TV)*.

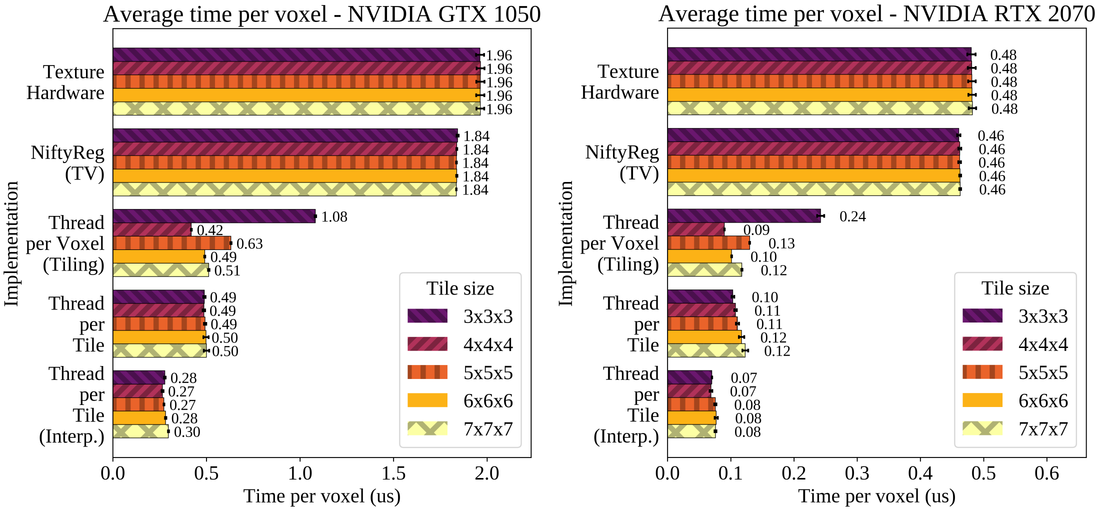
Dataset and metrics. We measure the timing information of BSI while applying registration on our dataset. We use two metrics to measure the performance: 1) *time per voxel* is the execution time necessary to interpolate a single voxel, and 2) *speedup* is the performance improvement over NiftyReg (TV).

Parameters. We select five different tile sizes to evaluate the behavior of the algorithms under different parameters, namely $3\times3\times3$, $4\times4\times4$, $5\times5\times5$, $6\times6\times6$, $7\times7\times7$. We select these tile sizes because they are centered around $5 \times 5 \times 5$, which is the default tile size for non-rigid registration in NiftyReg.

5.2. GPU performance

Figures 5a and 5b show the average *time per voxel* for TH, NiftyReg (TV), TV-tiling, TT, and TTLI on the GTX 1050 and the RTX 2070 GPUs, respectively.

We make three main observations. First, TTLI is the fastest implementation in all cases. Second, the time per voxel is almost independent of the tile size for all implementations except TV-tiling, for which the thread block size changes with the tile size. The reasons are three. 1) Bigger tiles leave more threads inactive at the borders of the image. 2) Bigger tiles decrease the coalescence of GPU memory accesses. In our approach, a single thread stores an entire tile in the output (Figure 3, Step 3). 3)



(a)

(b)

Figure 5: Average time per voxel of the five registration pairs for various tile sizes on GTX 1050 GPU (a) and RTX 2070 GPU (b). Error bars depict the standard deviation of time per voxel.

If the number of SMs does not divide the amount of blocks exactly, some SMs may remain idle (tail effect). In conclusion, the performance of our approach in regards to different tile sizes, is a balance between the acceleration that the reduction of data movement offers and the deceleration that border effects and memory uncoalescence cause. Third, for all implementations the coefficient of variation (error bars show the standard deviation across the images of our dataset) is less than 3% which reflects that the image contents do not affect the performance. The reason is that BSI is regular, i.e., it operates on all voxels uniformly.

Figures 6a and 6b show the average *speedup* over NiftyReg(TV) for TH, TV-tiling, TT, and TTLI on the GTX 1050 and the RTX 2070 GPUs, respectively.

We make two observations. First, our TTLI approach is 6.5× (up to 7×) faster than NiftyReg(TV), on average. TTLI outperforms the second fastest (TT) by an average of 1.77× on GTX 1050 and 1.5× on RTX 2070. Second, TTLI shows similar speedups over NiftyReg(TV) on both Pascal architecture (GTX 1050) and Turing architecture (RTX 2070) GPUs, which demonstrates that our optimizations are widely applicable and performance-portable.

5.2.1. *Analysis of performance limitations*

This section describes the limitations that define the performance of our approach.

TT does not provide significant speedup over TV-tiling. The reason is that our TT approach reduces data movement significantly, which makes TT compute-bound. We observe with the NVIDIA's Visual Profiler [37] that the compute utilization of TT is at about 90% of the peak. Since the amount of computation in TT is not reduced with respect to TV-tiling, the potential improvement is limited.

Reformulating the summation of TT to trilinear interpolations (Section 3.3) reduces the computational complexity of Equation (1) to half (Appendix B) and increases the usage of FMA instructions. TTLI is 50% - 80% faster than TT. After removing the computational intensity problem, TTLI is no longer compute-bound. The main bottleneck is the uncoalescence of the output (Figure 3, Step 3). In our experiments, fixing the uncoalescence proved more computationally costly than the uncoalescence itself.

Thread divergence, caused by the inactive threads at the borders of the image, reduces the computation throughput for both TT and TTLI.

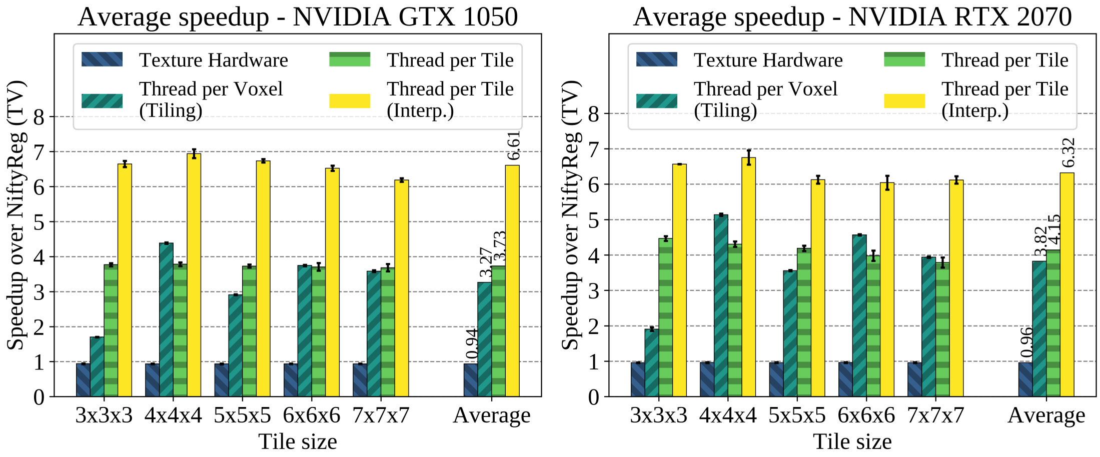
With $5\times5\times5$ tile, TTLI achieves 670 GFLOP/s and 62 GB/s on the GTX 1050 [2]. The empirical limits [38] of the GTX 1050 are 2091 GFLOP/s and 95 GB/s. We observe that TTLI is close to the bandwidth limit, but not so close to the computation limit.

5.3. *CPU performance*

We apply our approach to BSI to our CPU implementations (Section 3.5). Figures 7a and 7b show respectively time per voxel and speedup results of our CPU approaches for different tile sizes.

We make four observations. First, our CPU implementations (VT and VV) outperform the baseline NiftyReg (TV) by an average of 4.12× and 3.30×, respectively. Second, for all implementations, larger tiles result in lower time per voxel, as they can take more advantage of the CPU cache hierarchy. This effect is more pronounced in VT, which achieves a speedup of almost 5× for the largest tiles. Third, the speedup of VT increases as the tile size increases because bigger tiles fill more *slots* of the SIMD vectors. VT is the fastest option when more

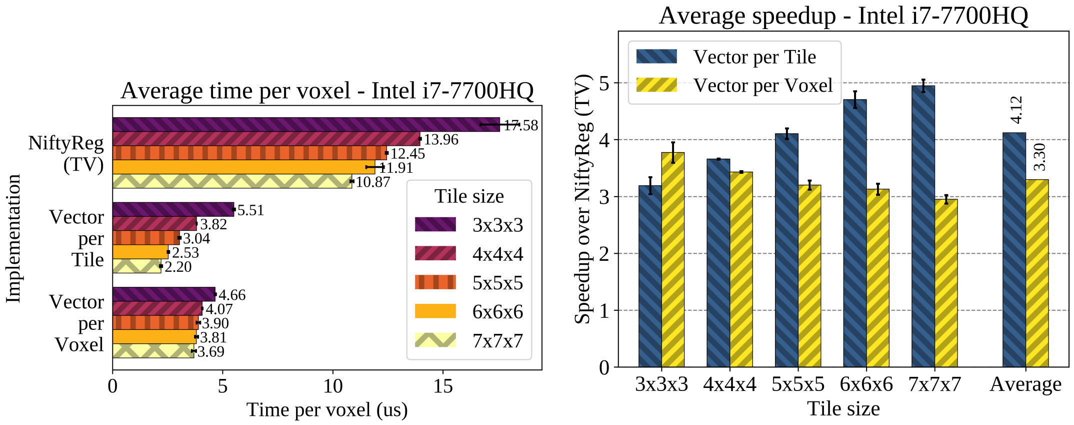
[2] NVIDIA profiler (version 2019.4.0) does not provide metrics for counting FLOPs on the RTX 2070.



(a)

(b)

Figure 6: Average speedup over NiftyReg(TV) for the five registration pairs with different tile sizes on the GTX 1050 GPU (a) and the RTX 2070 GPU (b). Error bars depict the standard deviation of the speedup.



(a)

(b)

Figure 7: Average time per voxel (a) and speedup (b) of BSI for various tile sizes using our implementation of BSI on CPUs. Error bars depict the standard deviation.

than 3 slots are filled. Fourth, the speedup of VV does not increase, as the time per voxel of NiftyReg decreases with faster rate than the time per voxel of VV. VV is the recommended option only for $3 \times 3 \times 3$ tiles.

5.4. Accuracy

Our implementations employ FMA instructions, which are more accurate than regular multiplications [8], in the calculation of linear interpolations. In this section, we show the accuracy improvements that stem from FMA instructions. We create a high precision CPU implementation by using double precision arithmetic (64-bits floating point numbers) and we use this implementation as reference.

Tables 3 and 4 show respectively the average absolute error of all GPU implementations and all CPU implementations with respect to the high precision CPU implementation.

Table 3: Average absolute error of BSI approaches on GPUs with respect to a high precision CPU implementation.

Implementation	Error (e^{-6})
Texture Hardware	9245
Thread per Voxel (Tiling)	5.5
NiftyReg (TV) GPU	5.3
Thread per Tile	5.6
Thread per Tile (Interp.)	2.8

We draw three conclusions. First, our implementations that employ FMA instructions (i.e., TTLI on GPUs, VT and VV on CPUs) are almost two times more accurate than the rest. Second, TH is significantly less accurate than the rest of the implementations, as expected from the low accuracy of interpolation hardware [8]. TH is 3300× less accurate than TTLI. Third, most GPU implementations show accuracy values in the same order of magnitude as CPU implementations.

Table 4: Average absolute difference of BSI approaches on CPUs with respect to a high precision CPU implementation.

Implementation	Error (e^{-6})
NiftyReg (TV) CPU	6.0
Vector per Tile	3.0
Vector per Voxel	3.0

6. Registration evaluation

In this section, we evaluate the performance impact of our BSI implementations on the overall registration process.

6.1. Evaluation methodology

To test the contribution of our BSI implementations to the total time required for the registration of medical images, we integrate our TTLI approach into NiftyReg[3] [7]. The control points in NiftyReg correspond to a coarse deformation field. We calculate the fine deformation field (i.e., the displacement of all voxels) by interpolating the coarse deformation field using BSI. We compare the total registration time with our BSI to the original NiftyReg registration, on our dataset presented in Section 4. We evaluate the performance of non-rigid registration on two platforms: a) a quad-core Intel i7-7700HQ@2.8 GHz CPU (with HyperThreading) and a GTX 1050 GPU, and b) a six-core Intel i7-8700@3.2 GHz CPU (with HyperThreading) and an RTX 2070 GPU. We set the tile size to 5 × 5 × 5, which is the default setting in NiftyReg.

6.2. Performance evaluation

Figures 8 and 9 show the total registration time and the speedup of our approach on the two platforms.

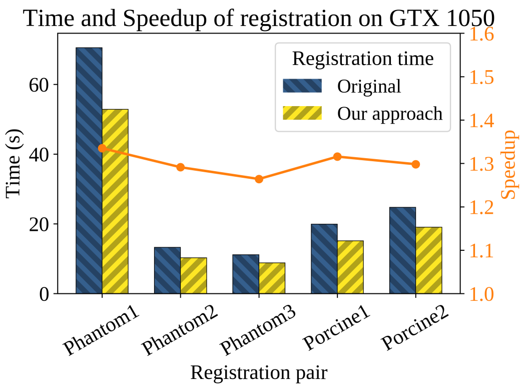


Figure 8: Time and speedup of registration with our improved BSI GPU approach on GTX 1050.

We draw two major conclusions. First, registration with our BSI approach is faster in all images on both platforms. The speedup of registration is 1.30×, on average, on the platform with a GTX 1050 GPU, and 1.14× on the platform with an RTX 2070 GPU. Second, although the performance improvement of our BSI approach is almost the same for both GPUs, we do not observe the same results for the entire image registration. The reason resides in Amdahl's law [39]: while BSI represents 27% of the total registration time on the platform with a GTX 1050 GPU, it takes only 15% on the platform with an RTX 2070 GPU.As a result, the overall performance impact on the registration workflow depends on the characteristics of the compute platform.

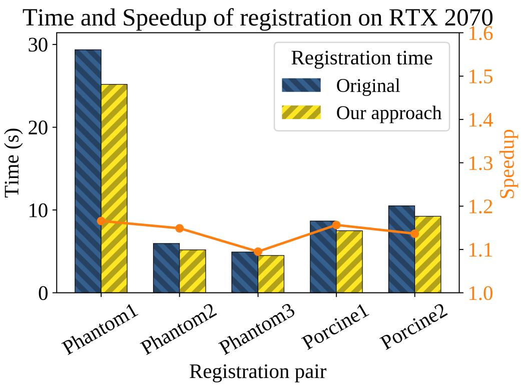


Figure 9: Time and speedup of registration with our improved BSI GPU approach on GTX 1050.

7. Clinical validation of image registration

In this section, we present the validation of our implementation of accelerated FFD on our pre-clinical dataset described in Section 4.

Qualitative assessment. We perform qualitative assessment of the registration using a checkerboard validation procedure [40]. Our method provides accurate registration for the parenchyma (the outer shape of the liver is preserved correctly) for both the liver phantom and porcine model. Tumors and vessel structures of the phantom are consistent between images (Figure 10) and approximately also vessel structures for the porcine model are correctly registered (Figure 11).

Quantitative assessment. We create normalized difference images between the output of the registration and the target intra-operative image for three registration approaches: 1) affine, 2) proposed, and 3) original NiftyReg (Figures 12 and 13). Table 5 shows the mean absolute error (MAE) for all images of our dataset. As expected, the mismatch to the target intra-operative image is greater with affine than with non-rigid registration approaches. The two non-rigid registration approaches perform almost equally (the average MAE across the five image pairs is 0.216 for affine, 0.1240 for our approach and 0.1249 for original NiftyReg.

In order to quantify how the different registration approaches affect the accuracy of the registration as output images, we apply Structured Similarity Index Metric (SSIM) [41] to our dataset. With the SSIM, we measure the similarity between the

[3] https://github.com/oresths/niftyreg_bsi

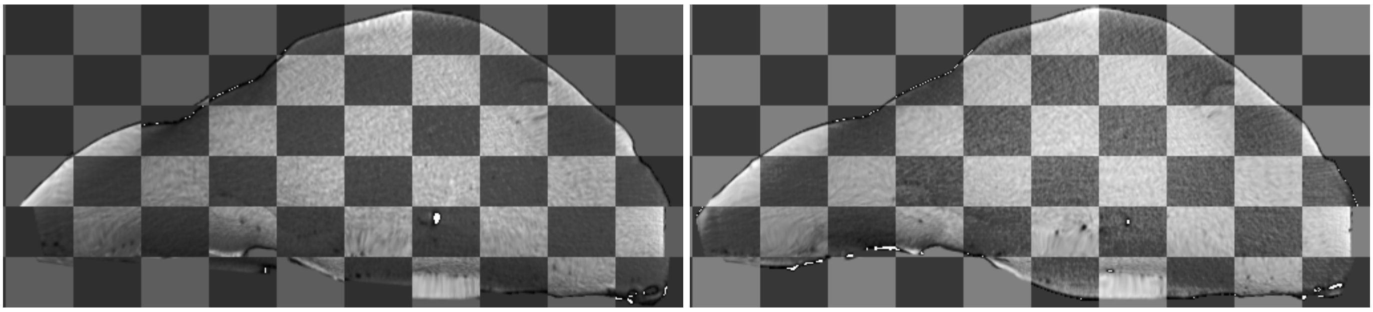


Figure 10: Comparison of registration through qualitative checkerboard assessment on liver phantom scans. (Left) shows the registration results using an affine registration. (Right) shows the results of non-rigid FFD using our BSI implementation.

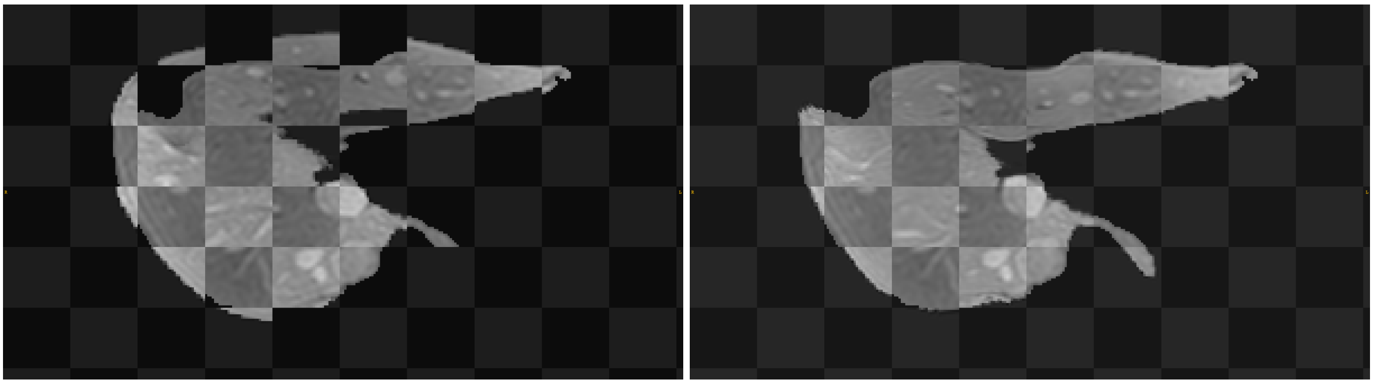


Figure 11: Comparison of registration through qualitative checkerboard assessment on porcine liver scans. (Left) shows the registration results using an affine registration. (Right) shows the results of non-rigid FFD using our BSI implementation.

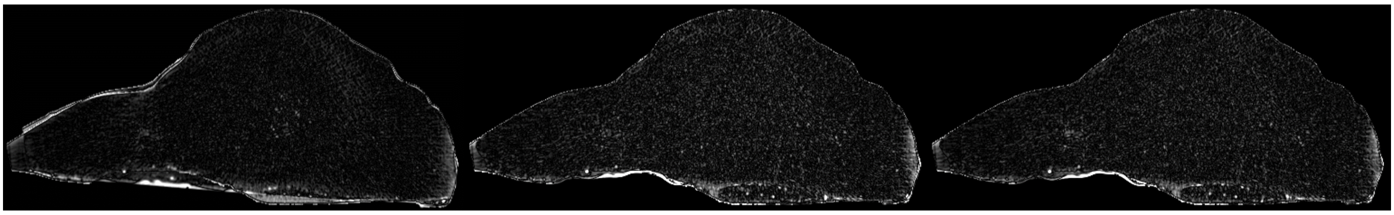


Figure 12: Comparison of registration through quantitative difference image assessment on liver phantom scans. (Left) shows results using an affine registration; (Center) shows the results of non-rigid FFD using our BSI implementation; (Right) shows the results of non-rigid FFD using original NiftyReg.

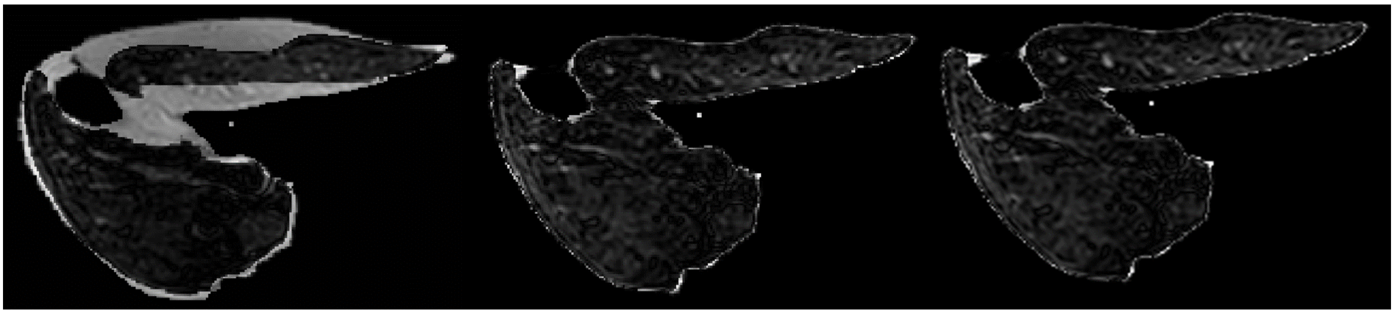


Figure 13: Comparison of registration through quantitative difference image assessment on porcine liver scans. (Left) shows results using an affine registration; (Center) shows the results of non-rigid FFD using our BSI implementation; (Right) shows the results of non-rigid FFD using original NiftyReg.

output of the registration approach and the target intra-operative image (Table 5).

We make three observations. First, the non-rigid registration approaches have much higher similarity than the affine registration approach. Second, our approach and the original NiftyReg have almost equal similarities. Third, our approach gives slightly better similarity than the original NiftyReg approach. Further evaluation of accuracy of the registration can be inferred from the original studies performed by Modat et al. [7].

Table 5: Mean absolute error (Left) on normalised outputs of affine registration and non-rigid registration with our approach and original NiftyReg, using the intra-operative image as reference. Structured Similarity Index Metric (Right) of the registration output, using the intra-operative image as reference).

Registration pair	MAE			SSIM		
	Affine	Proposed	NiftyReg	Affine	Proposed	NiftyReg
Phantom 1	0.229	0.13	0.131	0.865	0.929	0.934
Phantom 2	0.234	0.172	0.179	0.916	0.952	0.946
Phantom 3	0.256	0.174	0.172	0.889	0.952	0.95
Porcine 1	0.201	0.072	0.072	0.797	0.912	0.911
Porcine 2	0.162	0.072	0.071	0.716	0.737	0.737
Average	0.2164	0.1240	0.1249	0.8368	0.8963	0.8956

8. Discussion

In this work we optimize BSI and integrate it to FFD to accelerate the performance of medical image registration. However, our improved BSI can also be used in generic image interpolation applications, e.g., image zooming [42], by using image pixels as the control points.

The performance of image registration can be further improved by merging the other steps of FFD with B-spline interpolation. By optimizing the rest of the registration process, the execution time of the registration further diminishes, enabling new possibilities for fast intra-operative updates without intra-operative CT acquisitions, e.g., through liver models reconstructed with US [43] or through stereo video reconstructions [33].

The speedup of image registration through optimized FFD is important not only for pneumoperitoneum compensation, but also for compensation of several other deformations that the liver commonly undergoes during surgery. If real-time registration is possible, FFD can be used in IGS to compensate for deformations that result from lifting the liver with a surgical instrument or resecting liver ligaments (liver mobilization).

A limitation of our current implementations is that they work only with control point grids that are aligned to the voxel grid and uniformly spaced. Uniform spacing is usually sufficient for medical images [6, 19]. Support for non-uniform grids is possible with minimal changes (e.g., calculating B-spline basis functions weights on-the-fly). We leave this support for future work.

9. Conclusion

This paper presents our approach to B-spline interpolation, which is optimized to reduce data movement. The key idea of our approach is to assign one worker thread per tile of voxels. This has two main advantages. First, data movement during input loading is significantly reduced. Second, the input control points can be kept in registers during the entire computation. To further enhance the performance of our implementation, we rearrange the weighted summation of control points into trilinear interpolations. This results in two key advantages. First, the trilinear interpolations reduce the computational load. Second, they increase the interpolation accuracy.

Our experimental evaluation on two sets of subjects and imaging modalities shows that our BSI approach offers improved performance and accuracy with respect to state-of-the-art implementations. TTLI, our best approach on GPUs, performs up to 7× faster in comparison to the other GPU implementations. Our implementations that use trilinear interpolations perform approximately 2× better than the other in regard to interpolation accuracy.

We integrate our BSI approach into the NiftyReg medical image registration library and validate it in a pre-clinical application scenario. Our approach improves the performance of non-rigid image registration by 30% and 14%, on average, on our two platforms with a GTX 1050 GPU and an RTX 2070 GPU, respectively. The improved performance reduces the computation time of image registration. Therefore faster updates of the organ and its structures are possible during IGS.

As a result, non-rigid registration of medical images can benefit from our BSI approach on GPUs to greatly enhance the performance and accuracy of registration in time-critical applications (e.g., image guided surgery).

Acknowledgment

This work is supported by High Performance Soft-tissue Navigation (HIPERNAV - H2020-MSCA-ITN-2016). HIPERNAV has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 722068. The authors would also like to thank the radiology staff at the Intervention Centre, Oslo University Hospital, who collaborated to perform the animal experiment on the porcine model. Juan Gómez-Luna and professor Onur Mutlu would like to thank VMware and all other industrial partners (especially Facebook, Google, Huawei, Intel, Microsoft) of the SAFARI research group.

Appendix A. Off-chip memory to on-chip memory data movement

We use the external memory model [44] to describe the data movement from off-chip memory to on-chip memory. We consider a 3D image. Let us define M as the total number of voxels, $N = 64$ as the number of control points, T as the number of voxels inside each tile, and L as the size, in words (words are 32-bits long, a common size for storing integer and real numbers), of transactions into the cache (i.e., transactions between off- and on- chip memory). The L sized memory transfers of the three cases we are interested in are:

a) *No tiles*: When we do not have tiles, for each of the M voxels, we need to transfer N control points from global memory to shared memory. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{N \times M}{L} \tag{A.1}$$

b) *Hardware trilinear interpolation*: Each voxel is affected by the 4^3 control points surrounding it. However, if we use the texture unit to get their trilinear interpolations directly, only 2^3 loads are required [16]. Therefore, when we utilize the texture hardware for loading the input, for each of the M voxels, we need to transfer 2^3 control points from global memory to cache memory. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{2^3 \times M}{L} \tag{A.2}$$

c) *A block per tile*: When we use a block for each tile, for each tile we need to transfer N control points from global memory to shared memory. Each tile contains T voxels, thus the total number of tiles is M/T. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{N \times M}{T \times L} \tag{A.3}$$

d) *Blocks of tiles*: When we have 3D blocks of tiles, and each block contains $l \times m \times n$ tiles, for each block we need to transfer $(4 + l - 1) \times (4 + m - 1) \times (4 + n - 1)$ (Section 3.2.1) control points from global memory to shared memory (or cache). Each block contains $l \times m \times n$ tiles and each tile contains T voxels, thus the total number of blocks is $M/(l \times m \times n \times T)$. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{(4 + l - 1) \times (4 + m - 1) \times (4 + n - 1) \times M}{l \times m \times n \times T \times L} \tag{A.4}$$

Observations. We make the following four observations. First, a hardware trilinear interpolation implementation requires fewer memory transfers than a no tiles implementation because $2^3 < N$ in all cases. Second, a block per tile implementation requires fewer memory transfers than a hardware trilinear interpolation implementation because $N/T < 2^3$ when $T > 8$. $T > 8$ is a rare case (T is 125 by default in NiftyReg). Third, a blocks of tiles implementation requires fewer memory transfers than a block per tile implementation because $\frac{(4+l-1)\times(4+m-1)\times(4+n-1)}{l \times m \times n} < N$ as long as a block contains more than one tile. Fourth, the CPU implementations are a special case of Equation (A.4), in which $l = m = 1$, i.e., each thread processes contiguous tiles in the x-axis direction.

Appendix B. Computational complexity

In order to evaluate the arithmetic performance of TTLI and TT, we perform the computational analysis of both implementations in this section.

TT. For every voxel of the output image, we need to calculate the triple sum in Equation (1). Each operand of the summation requires the multiplication of one control point (ϕ) with three weights (B). Thus, each voxel requires $(64\ summands) * (3\ multiplications + 1\ accumulation) - 1 = 255$ vector (ϕ is a 3D vector in deformation fields) arithmetic operations. The calculation of Equation 1 requires $4 + 4 + 4 = 12$ scalar loads for the weights and 64 vector loads for the control points. If we use one weight for the $B_l(u) \cdot B_m(v) \cdot B_n(w)$ product, instead of three individual weights, the required operations decrease to $(64\ summands) * (1\ multiplications + 1\ accumulation) - 1 = 127$ (same as a parallel reduction) and the weights to be loaded increase to $4 * 4 * 4 = 64$. This is not suitable for our register-only implementations, because there are not enough registers to store the 64 weights and the use of one of the caches would impact the performance substantially (Section 3.4).

TTLI. For every voxel of the output image, we reformulate the summation of the $4 \times 4 \times 4$ weighted control points to trilinear interpolations. We divide the $4 \times 4 \times 4$ cubic neighborhood to eight $2 \times 2 \times 2$ sub-cubes, as in Figure 1. Each sub-cube corresponds to a trilinear interpolation. A trilinear interpolation requires seven linear interpolations for its calculation. A linear interpolation has the form $a + w * (b - a)$, which equals to a subtraction and a fused multiply-accumulate (FMA) operation. Thus, for the eight sub-cubes and the ninth final sub-cube that is formed by the eight results of the eight trilinear interpolations, we have $(9\ cubes) \times (7\ linear\ interpolations.) \times (2\ operations) = 126$ operations for each voxel.

Observations. Without taking into consideration instruction dual-issue, $\Theta(n)$ equals to 255*(number of voxels) and 126*(number of voxels) respectively.

References

[1] A. Bartoli, T. Collins, N. Bourdel, M. Canis, Computer assisted Minimally Invasive Surgery: Is medical Computer Vision the answer to improving laparosurgery?, Medical Hypotheses 79 (6) (2012) 858–863. doi:10.1016/j.mehy.2012.09.007.
URL `http://dx.doi.org/10.1016/j.mehy.2012.09.007`

[2] S. Bernhardt, S. A. Nicolau, L. Soler, C. Doignon, The status of augmented reality in laparoscopic surgery as of 2016, Medical Image Analysis 37 (2017) 66–90. doi:10.1016/j.media.2017.01.007.

[3] A. Teatini, T. Langø, B. Edwin, O. Elle, et al., Assessment and comparison of target registration accuracy in surgical instrument tracking technologies, in: 2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), IEEE, 2018, pp. 1845–1848.

[4] A. Sotiras, C. Davatzikos, N. Paragios, Deformable medical image registration: A survey, IEEE transactions on medical imaging 32 (7) (2013) 1153.

[5] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. Hill, M. O. Leach, D. J. Hawkes, Nonrigid registration using free-form deformations: application to breast MR images., IEEE Transactions on Medical Imaging 18 (8) (1999) 712–21. doi:10.1109/42.796284.

[6] N. D. Ellingwood, Y. Yin, M. Smith, C. L. Lin, Efficient methods for implementation of multi-level nonrigid mass-preserving image registration on GPUs and multi-threaded CPUs, Computer Methods and Programs in Biomedicine 127 (2016) 290–300. doi:10.1016/J.CMPB.2015.12.018.

[7] M. Modat, G. R. Ridgway, Z. A. Taylor, M. Lehmann, J. Barnes, D. J. Hawkes, N. C. Fox, S. Ourselin, Fast free-form deformation using graphics processing units, Computer Methods and Programs in Biomedicine 98 (3) (2010) 278–284. doi:10.1016/j.cmpb.2009.09.002.

[8] NVIDIA, CUDA C Programming Guide 9.0 (2017).

[9] E. Smistad, T. L. Falch, M. Bozorgi, A. C. Elster, F. Lindseth, Medical image segmentation on gpus–a comprehensive review, Medical image analysis 20 (1) (2015) 1–18.

[10] J. Gai, N. Obeid, J. L. Holtrop, X.-L. Wu, F. Lam, M. Fu, J. P. Haldar, W. H. Wen-mei, Z.-P. Liang, B. P. Sutton, More impatient: A gridding-accelerated toeplitz-based strategy for non-cartesian high-resolution 3d mri on gpus, Journal of parallel and distributed computing 73 (5) (2013) 686–697.

[11] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, B. P. Sutton, Z.-P. Liang, Accelerating advanced MRI reconstructions on GPUs, Journal of Parallel and Distributed Computing 68 (10) (2008) 1307–1318. doi:10.1016/j.jpdc.2008.05.013.
URL `http://www.sciencedirect.com/science/article/pii/S0743731508000919`

[12] H. Wang, H. Peng, Y. Chang, D. Liang, A survey of gpu-based acceleration techniques in mri reconstructions, Quantitative imaging in medicine and surgery 8 (2) (2018) 196.

[13] T. Kalaiselvi, P. Sriramakrishnan, K. Somasundaram, Survey of using gpu cuda programming model in medical image analysis, Informatics in Medicine Unlocked 9 (2017) 133–144.

[14] R. Palomar, J. Gómez-Luna, F. A. Cheikh, J. Olivares, O. J. Elle, High-performance computation of bézier surfaces on parallel and heterogeneous platforms, International Journal of Parallel Programming 46 (6) (2018) 1035–1062.

[15] N. Satpute, R. Naseem, E. Pelanis, J. Gomez-Luna, F. Alaya Cheikh, O. J. Elle, J. Olivares, Gpu acceleration of liver enhancement for tumor segmentation, Computer Methods and Programs in Biomedicine 184 (2020) 105285. doi:https://doi.org/10.1016/j.cmpb.2019.105285.
URL `http://www.sciencedirect.com/science/article/pii/S016926071931733X`

[16] C. Sigg, M. Hadwiger, Fast third-order texture filtering, GPU gems 2 (2005) 313–329.

[17] D. Ruijters, B. M. ter Haar Romeny, P. Suetens, Efficient GPU-Based Texture Interpolation using Uniform B-Splines, Journal of Graphics, GPU, and Game Tools 13 (4) (2008) 61–69. doi:10.1080/2151237X.2008.10129269.
URL `http://dx.doi.org/10.1080/2151237X.2008.10129269`

[18] X. Du, J. Dang, Y. Wang, S. Wang, T. Lei, A Parallel Non-rigid Registration Algorithm Based on B-Spline for Medical Images, Computational and Mathematical Methods in Medicine (2016). doi:10.1155/2016/7419307.

[19] J. A. Shackleford, N. Kandasamy, G. C. Sharp, On developing B-spline registration algorithms for multi-core processors, Physics in Medicine and Biology 55 (21) (2010) 6329–6351. doi:10.1088/0031-9155/55/21/001.

[20] I. Peterlík, H. Courtecuisse, R. Rohling, P. Abolmaesumi, C. Nguan, S. Cotin, S. Salcudean, Fast elastic registration of soft tissues under large deformations, Medical image analysis 45 (2018) 24–40.

[21] C. P. Lee, Z. Xu, R. P. Burke, R. Baucom, B. K. Poulose, R. G. Abramson, B. A. Landman, Evaluation of five image registration tools for abdominal CT: Pitfalls and opportunities with soft anatomy, in: Medical Imaging 2015: Image Processing, Vol. 9413, International Society for Optics and Photonics, 2015, p. 94131N.

[22] J. S. Heiselman, L. W. Clements, J. A. Collins, J. A. Weis, A. L. Simpson, S. K. Geevarghese, T. P. Kingham, W. R. Jarnagin, M. I. Miga, Characterization and correction of soft tissue deformation in laparoscopic image-guided liver surgery, Journal of Medical Imaging In Press (2) (2018). doi:10.1117/1.JMI.5.2.021203.

[23] S. F. Johnsen, S. Thompson, M. J. Clarkson, M. Modat, Y. Song, J. Totz, K. Gurusamy, B. Davidson, Z. A. Taylor, D. J. Hawkes, S. Ourselin, Database-Based Estimation of Liver Deformation under Pneumoperitoneum for Surgical Image-Guidance and Simulation, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 9350 (2015) 450–458.
URL `https://doi.org/10.1007/978-3-319-24571-3_54`

[24] D. Ruijters, P. Thévenaz, GPU prefilter for accurate cubic B-spline interpolation, Computer Journal 55 (1) (2010) 15–20. doi:10.1093/comjnl/bxq086.

[25] F. Andersson, M. Carlsson, V. V. Nikitin, Fast algorithms and efficient gpu implementations for the radon transform and the back-projection operator represented as convolution operators, SIAM Journal on Imaging Sciences 9 (2) (2016) 637–664.

[26] J. Carron, A. Lewis, Maximum a posteriori cmb lensing reconstruction, Physical Review D 96 (6) (2017) 063510.

[27] V. Volkov, Better performance at lower occupancy, Proceedings of the GPU Technology Conference (2010) 1–75.

[28] N. Whitehead, A. Fit-Florea, Precision & Performance : Floating Point and IEEE 754 Compliance for NVIDIA GPUs, NVIDIA white paper 21 (10) (2011) 767–75. doi:10.1111/j.1468-2982.2005.00972.x.

[29] A. Fog, The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, Technical University of Denmark, 2018th Edition (September 2018).

[30] Intel, Intel intrinsics guide, software.intel.com, retrieved January 17, 2019 (2019).

[31] O. Jakob Elle, A. Teatini, O. Zachariadis, Data for: Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration, Mendeley Data (2019). doi:10.17632/kj3xcd776k.1.
URL `http://dx.doi.org/10.17632/kj3xcd776k.1`

[32] A. Pacioni, M. Carbone, C. Freschi, R. Viglialoro, V. Ferrari, M. Ferrari, Patient-specific ultrasound liver phantom: materials and fabrication method, International Journal of Computer Assisted Radiology and Surgery 10 (7) (2015) 1065–1075. doi:10.1007/s11548-014-1120-y.
URL `http://dx.doi.org/10.1007/s11548-014-1120-y`

[33] A. Teatini, W. Congcong, P. Rafael, A. C. Faouzi, B. Azeddine, E. Bjørn, E. O. Jakob, Validation of stereo vision based liver surface reconstruction for image guided surgery, in: Colour and Visual Computing Symposium (CVCS), IEEE, 2018, pp. 1–6.

[34] PHILIPS, Ingenia: Instructions for use (2014).
URL `https://www.theonlinelearningcenter.com/assets/smiles/el_lp_r517/AW9661_UserDoc_HelpTopics_Ingenia_SA/R517_Documentation/en-US/ifu1_p7i_us_pnl.pdf`

[35] A. Teatini, E. Pelanis, D. L. Aghayan, R. P. Kumar, R. Palomar, Å. A. Fretland, B. Edwin, O. J. Elle, The effect of intraoperative imaging on surgical navigation for laparoscopic liver resection surgery, Scientific Reports 9 (1) (2019) 1–11.

[36] NVIDIA, Nvidia Turing Gpu Architecture Whitepaper (2018).

[37] NVIDIA, Profiler User's Guide (September) (2017).
URL `http://docs.nvidia.com/cuda/profiler-users-guide/index.html`

[38] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, et al., An empirical roofline methodology for quantitatively assessing performance portability, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2018, pp. 14–23.

[39] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, spring joint computer conference, ACM, 1967, pp. 483–485.

[40] J. P. Pluim, S. E. Muenzing, K. A. Eppenhof, K. Murphy, The truth is hard to make: Validation of medical image registration, in: International Conference on Pattern Recognition (ICPR), IEEE, 2016, pp. 2294–2300.

[41] A. Hore, D. Ziou, Image quality metrics: Psnr vs. ssim, in: 2010 20th International Conference on Pattern Recognition, IEEE, 2010, pp. 2366–2369.

[42] M. Unser, Splines: A perfect fit for signal and image processing, IEEE Signal processing magazine 16 (6) (1999) 22–38.

[43] L. W. Clements, J. A. Collins, Y. Wu, A. L. Simpson, W. R. Jarnagin, M. I. Miga, Validation of model-based deformation correction in image-guided liver surgery via tracked intraoperative ultrasound: preliminary method and results, in: Medical Imaging 2015: Image-Guided Procedures, Robotic Interventions, and Modeling, Vol. 9415, International Society for Optics and Photonics, 2015, p. 94150T.

[44] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, W.-m. Hwu, Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU), Synthesis Lectures on Computer Architecture 7 (2012) 1–96. doi:10.2200/S00451ED1V01Y201209CAC020.