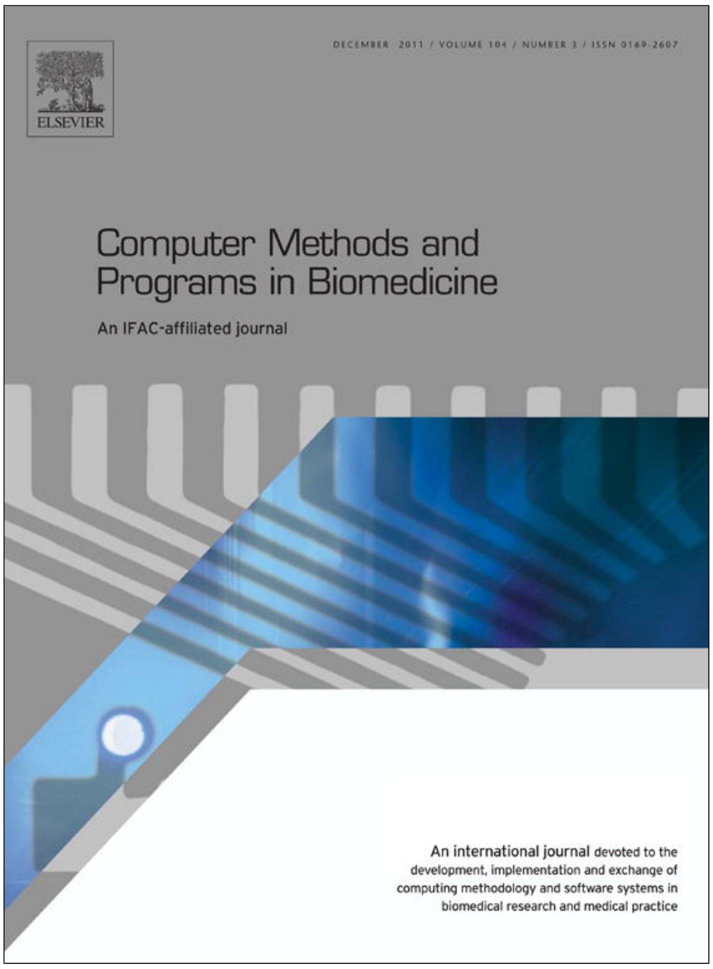

DECEMBER 2011 / VOLUME 104 / NUMBER 3 / ISSN 0169-2607
ELSEVIER
Computer Methods and Programs in Biomedicine
An IFAC-affiliated journal
An international journal devoted to the development, implementation and exchange of computing methodology and software systems in biomedical research and medical practice

computer methods and programs in biomedicine 104 (2011) e45–e57

journal homepage: www.intl.elsevierhealth.com/journals/cmpb

A survey of medical image registration on graphics hardware

O. Fluck[a,b,*], C. Vetter[a,c], W. Wein[a], A. Kamen[a], B. Preim[b], R. Westermann[c]

[a] *Siemens Corporate Research, Princeton, NJ 08540, USA*
[b] *Computer Science Department, Otto-von-Guericke-Universität, D-39106 Magdeburg, Germany*
[c] *Computer Science Department, Technische Universität München, Garching 85748, Germany*

ARTICLE INFO

Article history:
Received 5 February 2010
Received in revised form
4 October 2010
Accepted 14 October 2010

Keywords:
Medical imaging
Image registration
Acceleration
Graphics hardware
GPU

ABSTRACT

The rapidly increasing performance of graphics processors, improving programming support and excellent performance-price ratio make graphics processing units (GPUs) a good option for a variety of computationally intensive tasks. Within this survey, we give an overview of GPU accelerated image registration. We address both, GPU experienced readers with an interest in accelerated image registration, as well as registration experts who are interested in using GPUs. We survey programming models and interfaces and analyze different approaches to programming on the GPU. We furthermore discuss the inherent advantages and challenges of current hardware architectures, which leads to a description of the details of the important building blocks for successful implementations.

1. Introduction

One of the important promises of medical imaging as the eye of medicine is to establish spatial and temporal relationships among anatomical, physiological, and pathological information, which can be brought together for either inter- or intra-patient studies by means of image registration. Image registration is the process of aligning images – possibly from different imaging modalities – with various dimension cardinalities via establishing some common attributes. Image registration is categorized as an inverse problem, since the transformation parameters need to be extracted from the imaging data. It is now a common clinical workflow that the patient is scanned multiple times (various modalities or temporally) for diagnostic purposes. It is also becoming almost routine to have fusion of planning and treatment scans for image-guided interventions (see for example [1,2]). The ever increasing amount of images acquired together with the requirements of some clinical use cases for a quick diagnostic decision, have pushed the research toward making more efficient image registration processes. This requirement can be addressed on two fronts. One is to devise an algorithm with the computational efficiency in mind and second is to make best use of the computational power of the hardware. In this paper, we focus on the latter approach. Specifically we survey a body of research activities targeted at improving the registration time by utilizing the graphics processing units (GPUs).

While GPUs are built to generate rendered 2D-images from three-dimensional scenes, the inverse problem of inferring transformation parameters from images is not straightforward to solve. This registration requires a customized parallelization and a modification on conventional building blocks.

[*] *Corresponding author at*: Siemens Corporate Research, Princeton, NJ 08540, USA. Tel.: +1 609 865 3040.
E-mail addresses: oliver@fluck.de, fluck7@gmail.com (O. Fluck), christoph.vetter@siemens.com (C. Vetter).

0169-2607/$ – see front matter © 2010 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.cmpb.2010.10.009

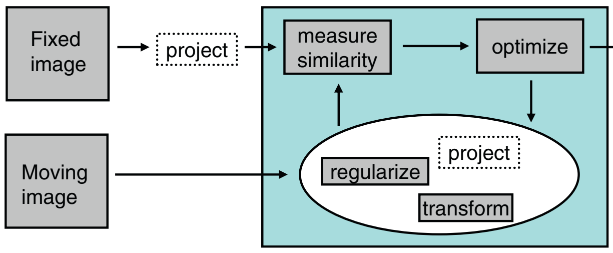


Fig. 1 – Stages of a registration process. The execution order of regularization, projection and transformation may vary among different registration algorithms. We depict this by circling these stages in the diagram. Note that a projection stage (shown in dotted lines) is not necessary in registration tasks of images of equal dimensionalities.

1.1. *Medical image registration in general*

During the registration process correct alignment of images is approached iteratively. Usually, one image is considered the fixed image (or source/reference image) and stays unchanged during iterations. In each iteration, the data of the moving image (target/template image) goes through a number of stages (Fig. 1). The implementation as well as the type of these stages vary depending on the relationship between moving and the fixed image, e.g. modalities and dimensions.

Generally, registration consists of finding a transformation $\widehat{\Phi}$ that maximizes the similarity measure S between the fixed image $I_f(X)$ and the deformed, warped, or transformed moving image $I_m(\Phi(X))$, with $I : \mathbb{R}^n \rightarrow \Omega$. The choice of S depends on the image characteristics.

One way to distinguish types of registration is by the amount of parameters which need to be optimized in order to find an alignment. A registration problem falls into either one of the groups named affine (or rigid if the scale and shear is kept constant), or non-affine (commonly called non-rigid), with increasing parameter space respectively. While a rigid 3D/3D registration aims to find six global parameters (rotation and translation in three dimensions), the parameter-space of a non-rigid registration (i.e. deformable registration) can be as large as the number of voxels multiplied by the cardinality of the moving data set.

When images are of different dimensions, projection operations $\mathcal{P}_f$ and $\mathcal{P}_m$ may be incorporated to project a higher-dimensional image onto the domain of a lower-dimensional one.

One can express this by

$$\hat{\Phi} = arg\ \max S(\mathcal{P}_f(I_f(X)), P_m(I_m(\Phi(X)))). \qquad (1)$$

While we are assuming continuous images for the mathematical discussion above, we are actually dealing with digital images $I : \mathbb{Z}^n \rightarrow \Omega$, therefore we need to assign intensity values at positions in the volume that are not necessarily grid points. This is done by an interpolation function L, with $L : \mathbb{Z}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$, which takes a d-dimensional image, a position vector of length d, and assigns an intensity at this position. The exact interpolation used is important, as it can lead to artifacts and influences the similarity measure [3,4].

1.2. *How suitable is graphics hardware for image registration?*

Driven by the gaming industry and the constant demand for more sophisticated graphics effects, graphics hardware has evolved from a simple interface over a fixed function pipeline towards a programmable supercomputer. The standard GPU in today's PCs outperform their CPU counterparts by a large factor in peak computation performance as well as memory bandwidth. Since the introduction of the programmable graphics pipeline a vivid research community [5] is working on strategies for scientific (non-graphics) computing on graphics hardware. A large overview of results in that area can be found in [6], however the work does not disclose most of the efforts in the field of image registration. Several forms of parallel processing in image registration have been reviewed in [7].

The non-standard implementation of floating point data as well as the absence of double precision floating point data types used to be an issue for numerical algorithms. But double formats are starting to appear and emulation techniques using single float format have been adapted to the GPU [8]. Some computations can even be performed with just 16 bits-floating point numbers [9].

Interpolation of pixel intensities is a common step in image registration. Because it is often performed on all pixels of an image, it can become very time consuming on regular serial processors. Fortunately, this operation is a fundamental part of the graphics pipeline (Section 2.1). Hence, GPUs are equipped with special hardware that performs linear interpolation more efficiently than code written for the CPU. Another important difference between CPUs and graphics processors is the degree of flexibility when accessing memory. Historically, read access at arbitrary memory locations has not been necessary for graphics programming, as it is dominated by localized memory access. Therefore texture lookups are typically optimized for localized memory access. Even more problematic is the lack of write-operations to arbitrary memory locations (scatter). This is a common reason for re-design of algorithms ported to the GPU (discussed in [10–12]).

An additional performance bottleneck is often caused by data transfers between main memory and graphics memory. Throughput rates between these two memory types are considerably low. However, once image data is residing in graphics memory, registration methods usually benefit from a very high bandwidth to the graphics processor.

1.3. *Performance gains*

With this work we are striving to report about all the work addressing GPU accelerated medical image registration. As we are looking at over a decade of research in which computing platforms have developed rapidly, we found it to be rather impracticable to list the reported performance gains side by side. The reported numbers in the cited articles depend on many variable factors, which make it difficult to extrapolate numbers from older papers to current hardware. The change in the GPU–CPU performance gap not only depends on the increase in pipelines and clock-rates, but also on memory and/or CPU–GPU bus bandwidth. Based on various characteristics, the performance comparison between an algorithm on CPU and GPU may be affected by the evolution on both platforms. We therefore put aside the reported numbers and refer to the cited articles for algorithmic details.

1.4. *Outline*

This paper is organized as follows. In Section 2 we will give an introduction to GPU programming models. We will look into programming interfaces such as the traditional techniques via graphics programming, as well as newly emerged general purpose interfaces and the GPU memory types they allow access to. In the following sections, we will discuss the registration components depicted in Fig. 1. In Section 3 we look into image registration cornerstones and details on how projection, transformation, and regularization can be implemented. Section 4 covers similarity metric computation and in Section 5 we conclude the survey and have a look at new developments which might have a future impact on this field.

2. GPU programming models and interfaces

The graphics pipeline on GPUs was at first restricted to fixed functionality with limited configuration possibilities, allowing an efficient hardware implementation. These restrictions were subsequently lifted to allow more programmability of different stages, through so-called shaders in high-level programming languages, resulting in increased possibilities to create a unique graphics style for each game. This has enabled researchers to tailor general purpose algorithms to be run at different stages of the rendering pipeline.

Various high-level programming languages are associated with different graphics APIs (application programming interfaces). The OpenGL shading language (GLSL) [13] is the high level shading language for OpenGL [14], HLSL is the equivalent for DirectX [15], whereas the output of a Cg [16] compiler can work with either OpenGL or DirectX. The graphics APIs and shading languages are functionally equivalent.

On the other hand, there are general purpose programming languages for GPU programming that are not geared towards graphics. Generally, these APIs are equally or more suitable for image registration. Brook [17] was one of the first general-purpose languages, which offered developers to strictly follow a stream-processing model (see Fig. 4 for a comparison of the stream processing programming model to the sequential programming model). Stream-processing is a paradigm for highly parallel computations that matches the architecture of modern GPUs (or the CELL processor) very well [18]. The input is a sequence of data of the same type (stream) that is processed element-wise by a so-called kernel. The parallelism is explicit, the programmer has the task to define it. Brook was not designed just for graphics APIs, and maps to various streaming architectures. For GPUs, Brook is built on top of graphics APIs and thus, supports different back ends to compile the code to either DirectX or OpenGL (or a CPU reference code). While Brook is hiding graphics related code, GPU vendors developed new interfaces to allow GPU stream processing by circumventing graphics programming. CTM (Close to the Metal) is an assembly-like language from ATI (now AMD). The next generation after CTM is called CAL (Compute Abstract Layer) [19]. The exposed Instruction Set Architecture (ISA) can be used directly or as target for a compiler. Brook+ is an extension of Brook that targets the ISA exposed by CAL. CUDA (Compute Unified Device Architecture) is a proprietary extension to the C language from NVidia and runs on NVidia hardware only. A more generally applicable framework is OpenCL that is not only supported on GPUs but also on CPUs. DirectCompute is another framework for general purpose programming that is part of the DirectX API. Fig. 2 gives an overview of the different languages. In the following sections we will have a closer look at two common frameworks following the two paradigms of graphics-oriented programming and stream processing. It has to be said that medical image registration algorithms can be implemented with any one of the available programming interfaces.

2.1. *The graphics pipeline*

Graphics processors are tailored to handle the complex three-dimensional scenes of current video games. The graphics pipeline consists of highly parallelizable components, so GPUs have been built to exploit this property. The input for the graphics pipeline are vertices – points in 2D or 3D describe the geometry of a rendered scene – and textures (images) that are mapped onto objects described by vertices.

The input vertices are transformed in the first programmable stage, the vertex stage, according to vertex programs running on the GPU (vertex shaders). The connections between vertices that form lines, triangles or more complicated polygons are only used in the next stage, which is programmable by geometry shaders. In this stage, geometric primitives (called primitives from now on) can be transformed and new primitives generated. The emitted primitives are then rasterized.

The rasterization step fills the primitives with fragments. Same as pixels, fragments are associated with a specific screen location, but in contrast to pixels, the fragments are not necessarily drawn on screen, if they are part of a primitive that is covered by another primitive. The generated fragments are then fed into the fragment stage in which the appearance of each fragment is determined by a fragment program (or fragment shader). During fragment processing, textures can be mapped onto the primitives. Depending on the scaling, one pixel of the texture (texel) might map to exactly one fragment of the rasterized primitive. In cases where texture size and primitive do not match, the GPU can automatically interpolate between texture elements. The supported interpolation modes are nearest-neighbor interpolation and linear interpolation, but more complex interpolation schemes can be implemented with shader programs. When blending is enabled, several fragments might be combined to determine the color of one pixel.

In the final step, the fragments are converted into actual pixels. The vertices, geometric primitive as well as the fragment data are processed independently in a data-parallel fashion. Fig. 3 gives an overview of this process.

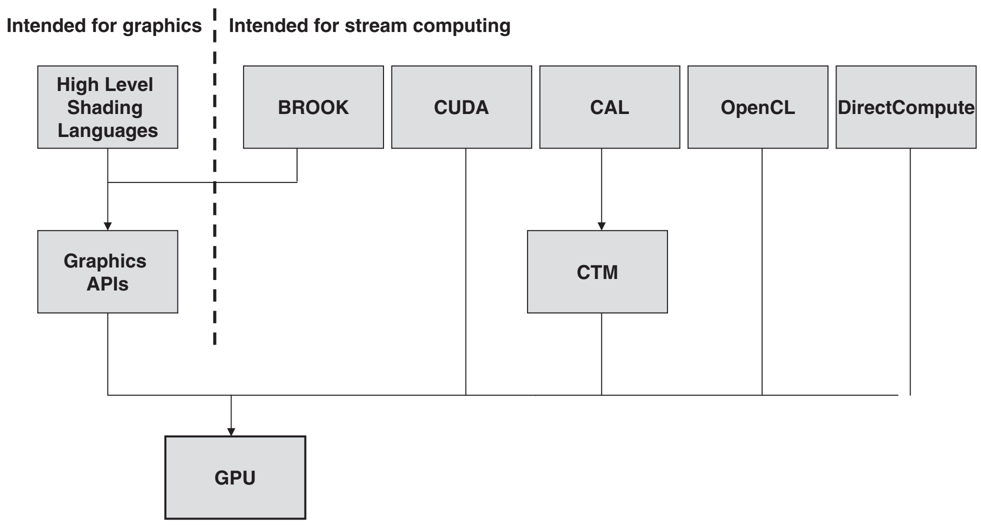


Fig. 2 – GPU programming languages. Higher shading languages are installed on the GPU via graphics APIs (left side). GLSL and HLSL are part of OpenGL and Direct3D respectively, whereas Cg can be used with both APIs. The family of languages especially designed for GPU computing are shown on the right, with BROOK (being the first of this kind) communicating with the GPU via OpenGL.

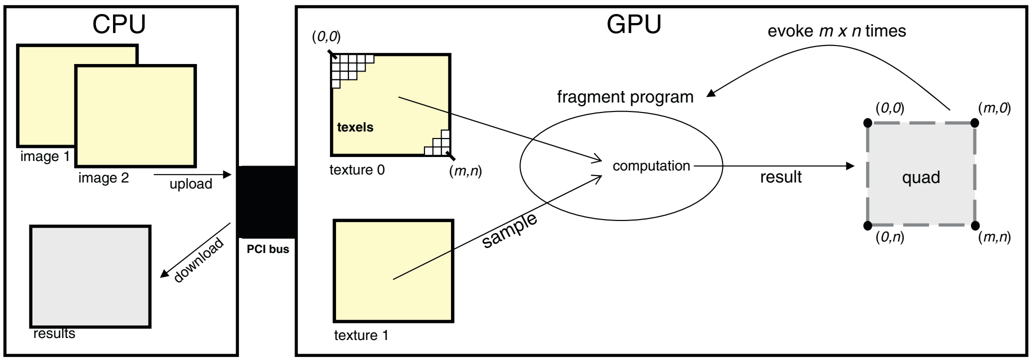
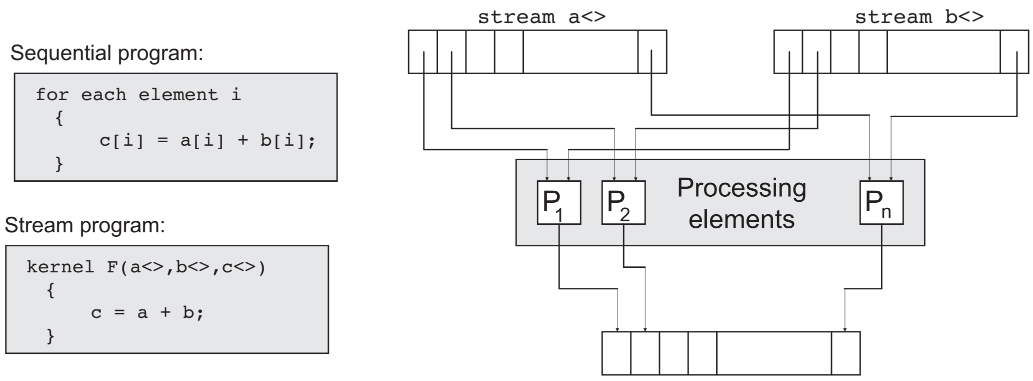


Fig. 3 – GPU computing initiated through a graphics API. A primitive is defined by vertices in space. Each vertex causes the execution of a vertex program. The size of the primitive projected on the image plane is measured in pixels. Each pixel of the output buffer evokes an execution of the loaded fragment shader program.


Sequential program:

```
for each element i
  {
      c[i] = a[i] + b[i];
  }
```

Stream program:

```
kernel F(a<>,b<>,c<>)
  {
      c = a + b;
  }
```

Fig. 4 – Stream programming model [20]. The example code written for serial processing compared to a stream-processing notation is shown on the left side. The right side shows an illustration of the stream-processing scheme.

2.2. *CUDA, OpenCL and direct compute*

CUDA (Compute Unified Device Architecture) is an extension to the C language from NVidia. In contrast to the graphics-oriented approach where the GPU code is strictly separated from the CPU code, CUDA kernels can be contained in .cu files where the GPU code of the kernels may be mixed with CPU code. In a preprocessing step, the contents of the code file are then directed to separate compilers for either CPU or GPU compilation. Alternatively, CUDA provides a lower-level driver interface with a clear separation of GPU code and CPU code. To match the underlying hardware, threads are hierarchically organized (see Fig. 6). Kernels in CUDA are the equivalent to shaders in graphics APIs, the computation of a fragment shader on a fragment is equivalent to a thread in CUDA (see Fig. 6).

OpenCL [21] is a framework for stream processors like GPUs, but supports multicore CPUs as well as the CELL architecture. It has been developed by Apple and is now managed by the consortium Khronos Group that manages OpenGL as well. Conceptually, OpenCL is very similar to the CUDA driver interface and the performance considerations detailed for CUDA apply to OpenCL programmed on NVIDIA GPUs as well. The notation for OpenCL is slightly different however. A work-item in OpenCL is a thread in CUDA and a processing element is called a scalar processor in CUDA. CUDA threads are grouped into blocks that are processed on a streaming multiprocessor. The local memory in OpenCL is called shared memory in CUDA, the private memory is OpenCL is called registers in CUDA. For NVIDIA GPUs, CUDA is the basis for OpenCL as well as for DirectX Compute which is the computing interface for DirectX starting with version 11.

2.2.1. *Memory types*

In contrast to graphics APIs like OpenGL, CUDA gives direct access to video memory which consists of several memory types (see Fig. 5). A number of registration algorithms benefit from having access to these memory types. These are often methods where the parallely accessed locations are not coherent with the thread IDs (e.g. CUDA) or texture coordinates (shading language). For example, the address for a memory write operation may be determined by the intensity of a pixel on which a kernel is executed. An example of such a case is the popular histogram-based similarity metric mutual information (see Section 4). In such cases, algorithms benefit from shared memory that can be used for communication within groups of threads (so called blocks), or global memory. In such cases, developers have to be aware of the possibility of conflicting memory accesses through concurrent threads.

2.3. *The right choice of programming interface*

We have seen that two main families of programming interfaces exist. The programming paradigm of choice may be dependent on the background and preferences of a developer, as all introduced interfaces can be utilized for medical image registration. Characteristics of registration algorithms however may play a role in the decision process. Developers with computer graphics experience may find GPU programming through a graphics API more suitable, especially when classical rendering schemes are utilized within the registration process, such as for projection (see Section 3.2). On the other hand, with general purpose programming interfaces, vendors opened up the underlying memory hierarchy to developers. This gives more freedom and flexibility which ultimately increases the efficiency with which many algorithms may run on graphics hardware. Advantages due to low level memory access have been reported for varying registration stages, such as regularization (see Section 3.5) and histogram computation for similarity computation (Section 4.1). However, the increased flexibility that CUDA (and OpenCL) offer to the programmer comes at the cost of higher demands on the optimization. The amount of resources (like registers) the threads in a kernel require, determines how many threads can be resident on a streaming multi processor at the same time. If the occupancy is too low, the performance suffers. Graphics API like OpenGL distribute the workload automatically to different streaming multi processors.

3. Image registration cornerstones: optimization, projection, transformation, and regularization on the GPU

In the following, we look into image registration cornerstones and details on how projection, transformation, and regularization can be implemented.

3.1. *Optimization*

Optimization algorithms can be divided into gradient-free and gradient-based approaches. Gradient-based approaches usually converge faster to an optimum, but are more complex, since the gradient has to be computed, either through a closed form or by numerical approximation. For GPU-based registration, it is important whether the registration performed is affine or deformable. Since affine registration have a very limited set of parameters (translation, rotation, shear, scale), computing the derivatives on the GPU does not offer any advantages, so only the cost function is evaluated on graphics hardware. For deformable registration, more parameters are available, especially if the deformation is described by a dense vector field with dimensions equal to the image dimensions. In this case, the computation of the gradient is usually performed on the GPU (see [9,10,20,22–32]). On the whole, however, optimization is not a focus of registration research on the GPU due to its low computational intensity and lack of computations that can be parallelized [7].

3.2. *Projection*

A projection stage is usually needed for example when registering 3D with 2D image data. Registration and fusion of these dimensionalities is common in minimal invasive interventions, computer aided surgery, or radiation therapy [1,2]. While 3D imaging such as computed tomography (CT) or magnetic resonance imaging (MRI) often requires relatively long acquisition times and is used for diagnostics and intervention planning, 2D images can be acquired in real-time (e.g. through fluoroscopy) during interventions and allow for guidance when registered with the planning data. With respect to Eq. (1) a projection of 3D to 2D corresponds to P_m, and P_f equals the identity transformation. Thus, $\mathcal{P}_f : \mathbb{R}^2 \to \mathbb{R}^2$ and $\mathcal{P}_m : \mathbb{R}^3 \to \mathbb{R}^2$.

In this case P_m refers to a reconstruction of the lower dimensional image by means of the higher dimensional one.

In a large body of work, rendering of Digitally Reconstructed Radiographs (DRRs) [33] is utilized for registration of CT and X-

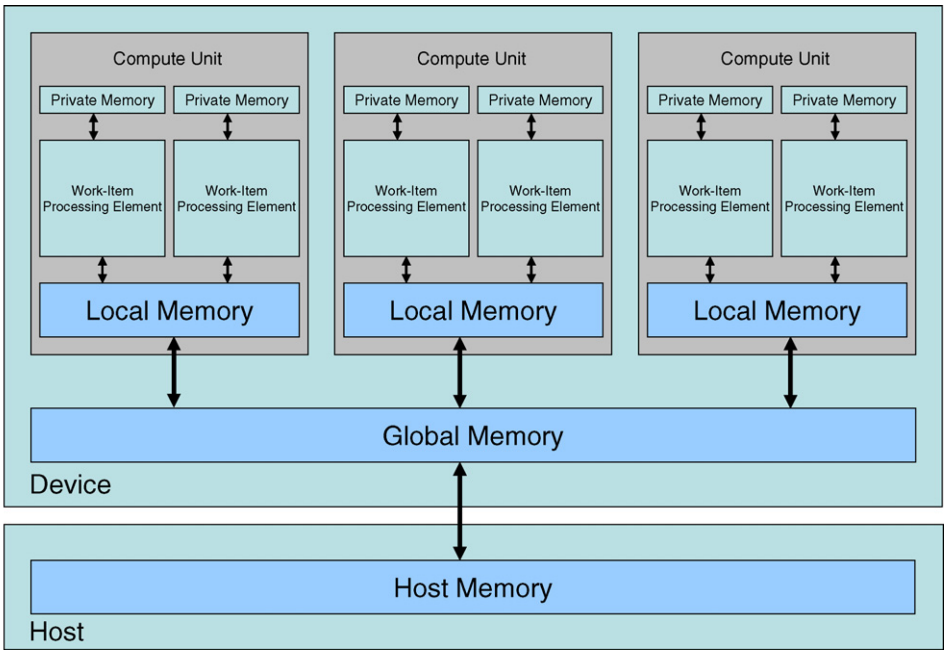


Fig. 5 – Conceptual overview of a compute Device in OpenCL: A work-item runs on a processing element. Work-Items are grouped into work groups that are processed on a compute unit. Local memory is accessible by all work-items within a work group, private memory only by the work-item it belongs to.

ray data. During X-ray acquisition simulation, integration of intensity information within CT data follows the same rules as if X-rays traverse through a patient's body. During the 'real' acquisition, a ray $r(x, y)$ traverses from its source to a detector plane while photons get attenuated on their path through different body substances and tissues. The energy intensity $I(x, y)$ reaching the detector plane determines the intensity of the final image pixel. The ray integral can be described by

$$I(x) = I_0 e^{\left(-\int_{r(x)} \mu(X, E_{eff}) dr\right)}, \qquad (2)$$

with $x \in \mathbb{R}^2$ and $X \in \mathbb{R}^3$. Note that Eq. (2) is slightly simplified with an effective energy E_{eff} and the assumption of a monochromatic X-ray source, and does not include scattering and other physical effects (we refer to [34] for more details). The varying attenuation factor along a ray's path is denoted by $\mu(X, E_{eff})$.

During the rendering process one can approximate the integral of Eq. (2) by building a discrete sum along ray directions. Taking this into account, radiographs can be reconstructed following popular volume rendering schemes [35]. Common techniques for this task are slice-based volume rendering and ray casting. GPU volume rendering algorithms first

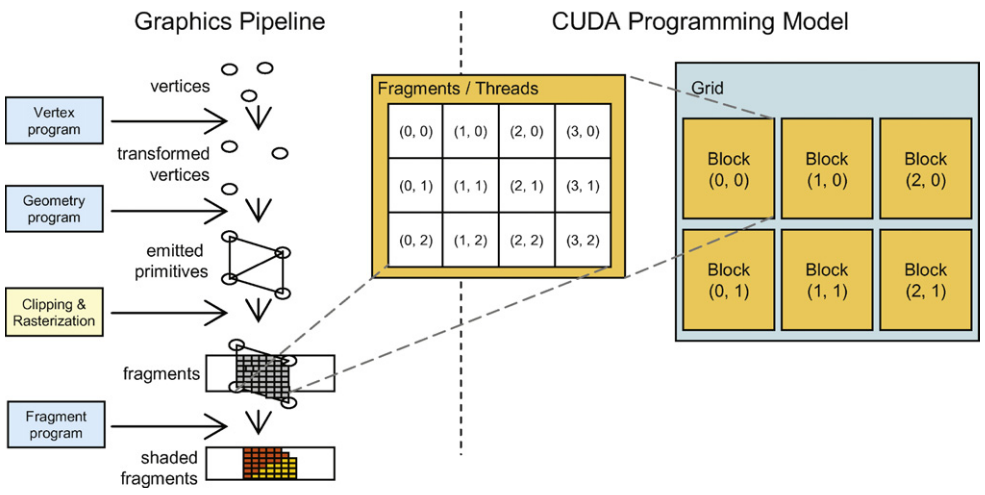


Fig. 6 – Design of modern graphics pipelines in relation to the CUDA framework. Using CUDA, programmers are allowed to avoid graphics related code. Furthermore, threads within a block are able to access shared memory as each block is executed on one of several multiprocessors of a GPU.

specify a volume's proxy geometry, either a stack of polygons in slice-based methods [36] or a bounding box for ray casting [37]. In older literature, we have found use of 2D textures [35]. 3D textures have been used later [38] where addressing of voxels became more straight forward and trilinear interpolation became available. In [38], the authors also include comparisons to multiple CPUs in high performance networks [39]. The problem of dissimilarities between CTA (CT Angiography) data including contrast material and X-ray images during DRR rendering is addressed in [40]. In [41], strategies on how to compute DRRs on older graphics hardware generations with lower render-buffer precision by employing Z-Buffers were presented. A fast method based on splat rendering has been proposed in [42].

3.3. *Rigid and affine transformation*

As a rigid or affine transformation does not require non-linear displacements of pixels, it only depends on a few parameters that globally govern image size, position, and orientation. Changes on these parameters can, for example, be conveniently mapped to manipulations on coordinates of the texture that holds the moving image. Following the graphics API approach for GPU computing, an arbitrary rigid overlay of two images can be achieved by first rendering one quadrilateral (2D) or several quadrilateral (3D) and then texture-mapping the images (or volume slices) onto it. For the moving image, size and boundaries of the texture should match with those of the quadrilateral. Texture coordinates of the moving image are allowed to change throughout the registration process. Texture coordinates can be defined during the setup stage before each rendering pass to allow for different pixel correspondences during rendering. These correspondences are implicitly established due to the GPU texture unit. Using general-purpose APIs, the offset is explicitly encoded within the kernel. After sampling from both images, the quality of the alignment will then be evaluated (see Section 4).

3.4. *Non-rigid transformation*

In the following sections we will describe non-rigid registration algorithms which have been successfully ported to graphics processors.

3.4.1. *Regularized gradient flow*

As to our knowledge, the first 2D deformable image registration on graphics hardware was proposed in 2003 [22], a hardware accelerated regularized gradient flow (RGF) which aims to minimize the energy

$$E = \frac{1}{2}\int_{I_f,I_m} \left| i_m(\Phi(x)) - i_f(x) \right|^2, \tag{3}$$

where Φ evolves along the gradient

$$E' = (i_m(\Phi(x)) - i_f(x))\nabla i_m(\Phi(x)). \tag{4}$$

The regularization used in this method is described in Section 3.5. Attempts to extend the RGF registration to three dimensions have been presented in [23]. The lack of render-to-3D-texture features was reported as the reason for insufficient speed for a deformable RGF registration in 3D. However, graphics hardware following the new DX10 standard exposes this feature.

3.4.2. *Demons*

The use of demons is a very popular approach for GPU accelerated deformable registration. This algorithm is an optical flow ([43]) variant, a concept from computer vision and based on the assumption that for small movements the intensity at a point remains the same. Based on the constant intensity, a velocity vector can be inferred from gradients of the intensity values. The displacement is computed by solving Eq. (5), with I containing both the moving and fixed image, u and v the unknown components of the displacement vector.

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0. \tag{5}$$

Due to the underlying assumptions of constant intensity, this algorithm is commonly used for single modality registration. It approaches a final alignment in each iteration k by updating for each voxel a displacement field U, such that

$$U(X)_k = U(X)_{k-1} + \frac{(I_m(\Phi(X)) - I_f(X))\nabla I_f(X)}{(\nabla I_f(X))^2 + (I_m(\Phi(X)) - I_f(X))^2} \circ G_\sigma, \tag{6}$$

where $U(X) = \Phi(X) - X$, which is regularized by a Gaussian filter in each iteration. Note that U_k is not updated in cases where the denominator falls under a certain threshold. In [20], the authors implemented the demons algorithm using the Brook programming environment. Later work also follows Eq. (6) for the registration of MR images of the head [24]. Further work we have found on GPU accelerated demons utilizes the CUDA framework [25,26] and includes a performance comparison to [24]. In [44] five Demons variants were implemented using the CUDA framework and tested on CT data.

3.4.3. *Physics based*

An interactive deformable registration method incorporating stiffness models has been presented in [27]. Here, a fast GPU-based segmentation has been utilized to allow users to select different tissues in images and assign physical attributes. The iterative algorithm computes a dense deformation by means of optical flow (see also Eq. (5)) and applies the resulting vector field as external force to compute a physically correct finite element deformation.

A successful implementation of a non-parametric deformable registration method solving a viscous-fluid partial differential equation (PDE) was presented in [28]. Here material properties are assigned manually by specifying anatomic templates. As described in the paper, equations are solved on the GPU numerically by solving a system of linear equations using a finite difference discretization. Closely related to fluid mechanics is the approach of image registration via optimal mass transport (OMT) [29], the authors report a successful implementation of an OMT multigrid algorithm on the GPU.

3.4.4. *Free-form deformation*

The motivation behind free-form deformation (FFD) techniques in image registration is to reduce computational complexity by introducing a lattice of control points. Finding a correct displacement of $n_x \times n_y \times n_z$ elements is then reduced to finding the correct constellation of $m_x \times m_y \times m_z$ control points $\phi_{i,j,k}$ with initial spacing δ. Pixel displacements throughout the image domain are then governed by these control points and in the three-dimensional case computed by the interpolation function

$$\Phi(X) = \sum_{l=0}^{d}\sum_{m=0}^{d}\sum_{n=0}^{d} B_l(u)B_m(v)B_n(w)\phi_{i+l,j+m,k+n}, \tag{7}$$

with basis function B. Control points surrounding X can be addressed with $i = \lfloor x/\delta \rfloor - q$, $j = \lfloor y/\delta \rfloor - q$, $k = \lfloor z/\delta \rfloor - q$, with (x, y, z) being components of X, and q being dependent on degree d (e.g. $q = 1$ for cubic interpolation).

This has been taken into account in [45], where finding optimal control point positions is formulated as the optimization of an energy functional. The approach has been successfully tested on synthetic as well as CT thorax datasets. An application with significant speedup using graphics hardware has been reported in [46]. Here locations of texels relative to control points (u, v, w) can be determined using texture coordinates (when using a graphics API) or thread IDs (when using CUDA or OpenCL).

In [47], the authors demonstrate how the deformation of volumes can be accelerated using graphics hardware. The method is based on slice based volume rendering where control points are defined by vertices and then again vertices are associated with three-dimensional texture coordinates. A linear interpolation method can be chosen during texture initialization and will then be computed rapidly by the GPU during rendering. For this reason, the translation of control points can be stored and updated in texture coordinates whereas the geometry remains static. The method allows the rendering of a deformed volume without constructing an intermediate dataset and allows for subdivision schemes for adaptive refinement. Focusing on brain tissue deformations, a variant of this has been tested on MR images in [48]. In [49], the authors have extended [47] and [48] and allow for additional grid points with positions determined using 3D Bézier functions. Another paper [46] shows how deformed DRRs can be created using ray casting. The proposed method combines fast GPU ray casting [37] with the concept of inverse rays deformation [50], the method allows for arbitrary interpolation techniques. In [51], Bi-cubic Bézier basis functions have been implemented for image warping and used for pre-calculation for finite element basis functions, and Ruijters et al. [52] use cubic B-spline deformation for elastic image registration as well. The work in [53] presents a parallel-friendly re-formulation of free-form deformation. Implementation and evaluation has been conducted using the CUDA framework.

3.5. *Regularization*

Regularization is a means for dealing with ill-posed problems such as deformable image registration. When pixels are allowed to be displaced arbitrarily, deformations have to be smoothed to ensure topology preservation. Furthermore, regularization prevents an optimization function to get stuck in local minima. In other words, regularization makes the problem tractable by restricting the solution space.

In the RGF 2D method of [22] (see Section 3.4.1), gradients are regularized by Jacobi iterations during a multigrid-cycle. In order to prevent faulty results by local minima their work also proposes computation on multiple scales. In [23], the authors proposed the use of the simpler Gaussian filter for faster processing instead of a multigrid regularizer. In fact, optimized versions of Gaussian smoothing have been used for the regularization of the flow field in several papers [23,25,10]. While Gaussian smoothing is relatively straightforward to implement, it poses challenges concerning its performance if the blurring kernels are very large. Improved performance over shader based techniques has been reported using the CUDA framework which provides shared memory and finer-grained memory access [11]. Recursive filtering is often used in CPU based work because it is cache efficient and the computational complexity is independent of the filter width. Recursive filtering [54] has been used for registration on the GPU in [24], but has the drawback that it restricts the parallelism. The integration of a smoothing term into the registration error function in form of a weight image has been proposed in [51]. Other options for efficient smoothing on the GPU are multigrid or Jacobi iterations [30].

4. Similarity metrics on the GPU

In order to evaluate the quality of alignment between multiple images, the similarity S of Eq. (1) of image features needs to be computed.

Similarity metrics fall into two broad classes:

- based on landmarks, and
- based on voxel intensities.

The only example of a landmark-based registration on the GPU we have found is [55]. The authors register microscopic images non-rigidly and have chosen the GPU because of the large amount of data that is processed, in the range of 16k times 16k pixels per image or more. The window size of the selected features has to meet a minimal variance threshold to be considered, in order to find informative features. Feature matching is performed by determining the best match for the selected feature within an alignment window in the second image. The metric used in this case is the normalized cross correlation. The cross correlation is implemented using the fast Fourier transform in CUFFT (part of [54]) provided by NVIDIA. Apart from the metric computation, the registration process runs on the CPU due to the communication overhead between CPU and GPU.

All other surveyed papers on GPU-based registration used voxel intensity based metrics. Since these metrics are performed for each voxel of a volume, the high computational and memory throughput demands are a good fit for the GPU. A large number of different similarity metrics have been implemented on the GPU: Sum of Squared Differences (SSD), Sum

Table 1 – Reported GPU implementations of common similarity measures and their characteristics.

Measure	Reference	Strength	Weakness
SSD	[23,51,56]	Simplicity	Result can be distorted by a minority of pixels with large difference in intensities
SAD	[56]	Reduces the prob. of SSD	Same as with SSD but less strong
NCC	[55,56]	Unaffected by variations in contrast and brightness	Regions with inverse intensity relation can have a compensating effect. Hence, not necessarily suitable for multi-modal registration
MI	[12,9,10,32]	Stable measure for multi-modal registration	Relies exclusively on statistical information of available intensities. No spatial information is considered

of Absolute Differences (SAD), Variance of Differences (VOD), Normalized Cross Correlation (NCC), mutual information (MI) (Table 1). However, all these metrics can be implemented with an element-wise stream processing of two input streams, reduction and histogram generation.

An average of 8 metrics has been implemented in [56]. Another approach [57] uses projection in order to generate similarity metrics in a lower-dimensional space and gives an overview over the speedup for different similarity measures on the GPU. In order to allow the analytic differentiation of energy functions with more complex similarity measures, an automatic differentiation approach has been proposed for GPU code in Cg [31].

One of the simplest similarity metrics in use is the sum of squared intensity differences (SSD). The SSD has been implemented on the GPU for example in [23,51]. Since the SSD (see Eq. (8)) only compares intensity values at corresponding sample points in the images/volumes, this measure is particularly easy to implement on the GPU. A reduction operation that adds all the computed values (see Section 4.1.3) is still necessary to compute the sum, though. $I_{f,m}$ in the equation is the overlap of the two images I_f and I_m.

$$I_{\mathrm{SSD}}(U) = \frac{1}{|I_{f,m}|}\sum_{I_{f,m}}(i_f - i_m)^2. \tag{8}$$

Another popular metric for the GPU is mutual information (MI). The mutual information metric measures the dependency between two random variables, and has been simultaneously proposed in two works [58,59]. The metric can be defined as in the following formula:

$$I_{\mathrm{MI}}(U) = -\int_f\int_m p_U(i_f, i_m)\log\frac{p_U(i_f, i_m)}{p(i_f)p_U(i_m)}\,di_f\,di_m, \tag{9}$$

with U representing the deformation field that aims to align the intensities i_f and i_m of the two images I_f and I_m. The different probabilities are denoted by p, $p(i_f)$ is the probability of intensity i_f in the fixed image, $p_U(i_m)$ is the probability of the intensity i_m in the moving image and $p_U(i_f, i_m)$ is the probability that the intensities i_f and i_m are at corresponding points in the images given a deformation field U. The integration is performed over the number of bins f in the fixed image and the number of bins m in the moving image. In order to compute this similarity metric on the GPU, the histograms are generated first. Using these histograms, the computation consists just of simple arithmetic operations plus a logarithm, followed by a reduction step. The same is true when the gradient $\nabla_U I_{\mathrm{MI}}$ is used. This metric has been implemented in several papers [12,9,10,32]. The mutual information computation benefits from a more complex algorithm. Parzen windowing estimates the underlying probability density function from the observed sample [60,9,10]. Instead of interpolating new intensity values in the moving image at the position of the grid points of the fixed image, the values of the nearest neighbors in the moving image are distributed into the histogram, weighted according to the linear interpolation when partial volume interpolation [61] is used. This technique is implemented on the GPU in [62].

Another example of a specific similarity metric implemented on the GPU is gradient correlation, implemented in [38].

4.1. *Implementation building blocks*

In the following sections we want to discuss the three main concepts for implementing similarity measures on the GPU.

4.1.1. *Interpolation on the GPU*

Interpolation is an important part of the graphics pipeline for applying textures to geometric primitives, in order to add details. Because of its importance, interpolation is directly supported by the hardware. The hardware-supported interpolation modes are nearest-neighbor and linear interpolation [63]. These interpolation modes are very fast, but suffer from two drawbacks: the interpolation type (linear interpolation) itself might not be precise enough and the accuracy with which the interpolation is computed is limited, since only 8 bits are used to encode the fraction between two grid points. Therefore, an implementation of a custom scheme with higher accuracy in the shader/kernel code might be warranted [64,65].

4.1.2. *Element-wise stream processing*

Basic stream processing is the building block of the most straight forward implementation of a metric. Considering the two input streams I_f and I_m, corresponding elements are processed in parallel and written to a location in an output buffer corresponding to its location in the input stream. For many metrics, such basic operations are the first step when evaluating the alignment of two images. Often, not only one element has to be taken into account, but a local neighborhood of values, for example in 2D or 3D.

Table 2 – Application of GPU based image registration.

Modality	Reference	Type	Appl.	Year
CT/MR	[70]	Rigid 3D	Brain	1998
CT	[35]	Non-rigid 3D	Swine lung	2001
	[57]	Rigid 3D	Pelvis, head	2006
	[25]	Non-rigid 3D	Lung	2007
	[26]	Non-rigid 3D	Lung	2008
	[28]	Non-rigid 3D	Head, neck	2008
	[44]	Non-rigid 3D	Thorax	2010
CT/SPECT	[9]	Non-rigid 3D	Cardiac	2007
	[10]	Non-rigid 3D	Cardiac	2008
CT/X-ray	[34]	Rigid 2D/3D	Pelvis	2005
	[38]	Rigid 2D/3D	Spine, femur	2006
	[56]	Rigid 2D/3D	Head, thorax	2007
	[46]	Non-rigid 2D/3D	Thorax	2008
	[31]	Rigid 3D	N/A	2008
CTA/X-ray	[40]	Rigid 2D/3D	Aorta	2008
MR	[70]	Rigid 3D	Brain	1998
	[49]	Non-rigid 3D	Brain	2002
	[79]	Non-rigid 3D	Brain	2004
	[24]	Non-rigid 3D	Head	2008
	[80]	Non-rigid 3D	Brain	2008
	[51]	Non-rigid 2D	Cardiac	2008
	[71]	Non-rigid 3D	Brain	2008
	[28]	Non-rigid 3D	Head, neck	2008
Microscopic	[55]	Non-rigid 2D	Breast	2009
Retinal images	[32]	Rigid 2D	Retina	2008
Not specified	[22]	Non-rigid 2D	N/A	2003
	[23]	(Non)rigid, 2D and 3D	N/A	2006

4.1.3. *Stream reduction*

A stream often needs to be reduced to a single quantity in order to obtain a statistical representation by counting intensities. The reduction operation is one of the first operations on GPUs that have been extensively studied [66,67]. Reduction on the GPU works by a gather approach: With each step the size of the stream is reduced, usually by a multiple of two and for each output element, the corresponding elements in the larger stream are gathered and combined according to the required operation. A detailed description of the optimization steps that can be used for reduction in CUDA can be found in [68]. With these two building blocks, metrics such as SSD (see Eq. (4)) that just takes the sum of the squared distances between the fixed and the moving image, can already be implemented.

4.1.4. *Histogram computation*

For metrics such as MI, a histogram has to be computed. This requires scattered writes to memory addresses and therefore does not fit nicely into the streaming architecture of modern GPUs. The OpenGL extension for computing histograms [69] – used in [70] – is not widely supported anymore. Using graphics APIs, vertices can be used for scattering operations. Texture input comes from vertex-texture fetches [9] or using vertex-buffer-objects that allow to interpret a texture as vertex data [10,71]. Blending is the approach in the graphics pipeline to combine incoming data with data that is already stored at a memory address, this allows scattered writes with a well-defined behavior. As demonstrated in [72], the histogram generation can be converted into a gathering process. For a comparison of different histogram algorithms on the GPU complete with speed comparisons see [73].

In CUDA, OpenCL and CAL scattered writes are supported directly. Atomic functions in CUDA consist of a read and write access to a memory location without interference from any other thread. Atomic operations are guaranteed to succeed, whereas simple write operations that try to access the same memory location simultaneously are serialized by discarding every operation except one. The sequence in which simultaneous write operations to the same memory space are performed is still undefined however. Atomic operations for shared memory can be simulated [11].

Depending on the amount of bins, several strategies [74,11] can be used. For a small amount of bins for the histogram, every thread can allocate a whole sub-histogram in shared memory, avoiding write conflicts but restricting the number of bins. In the case of more bins, a single sub-histogram for several threads is kept instead.

5. Conclusion and outlook

In recent years, the medical imaging registration community has embraced GPUs as powerful yet cost-effective parallel processing hardware. The key advantages of GPU based image registration are a high memory throughput, used for re-sampling huge data sets, as well as the high parallelism in the processing unit and specialized hardware for interpolation. Researchers have managed to overcome the architectural limitations of GPUs with innovative algorithms, achieving tremendous speed-up in numerous cases. Table 2 gives an overview of modalities and application areas.

5.1. *Comparison of many-core architectures*

Compared to single CPU implementations, GPU implementations of registration algorithms remain more challenging, however, the reason for this is mostly the different approach to the algorithm design that has to be divided into a massive amount of work threads. This is true for every many-core platform that is available, whether it is Cell, a many core CPU or a cluster of machines. Implementations on the GPU are potentially limited by the amount of memory available. Current GPUs provide up to 4GB of memory. This limit is sufficient for many registration tasks, especially for typical image dimensions in interventional imaging. The memory limit can furthermore be increased by adding several GPUs to one computer. Compute clusters are in many ways complementary to GPU computing, since the computers in the cluster can be equipped with potentially several GPUs. A drawback at the moment is the slower speed of double precision floating point computations on the GPU, a drawback the GPU shares with the CELL architecture (two reports on registration implementations on CELL processors can be found in the literature: [75,76]). However, full precision is not always necessary [9,10]. Furthermore, mixed-precision solvers have been demonstrated to speed up computations such as in finite element problems while maintaining the accuracy of double precision computations [77]. However, the use of mixed precision registration on the GPU has not been extensively investigated so far.

5.2. *Future directions*

We observed that the rate with which new APIs emerge or change has slowed down. As the bibliography shows, CUDA has established itself as a popular platform not only for registration but also for other tasks in medical imaging such as segmentation and filtering. Since the race for higher clock-rates within the CPU industry has come to an end and processors started to become rather “wider than faster”, research results in general purpose computation on graphics hardware are an important factor towards a possible adoption of GPU technology by CPU manufactures. Novel processor designs with a multitude of cores started to be available (IBM cell processor) or announced [78], which will be interesting platforms for algorithms that require a rather hybrid processor design. At this point, developers still have to decide for a platform for massively parallel processing, be that a single GPU, or multiple processors organized in clusters. As OpenCL receives broad support from all processor manufacturers, it appears as an emerging standard for programming parallel architectures. Such standardization might allow the reduction of processor specific programming efforts.

REFERENCES

[1] U. Pietrzyk, K. Herholz, A. Schuster, H.-M.v. Stockhausen, H. Lucht, W.-D. Heiss, Clinical applications of registration and fusion of multimodality brain images from PET, SPECT, CT, and MRI, Eur. J. Radiol. 21 (3) (1996) 174–182.

[2] T. Peters, K. Cleary (Eds.), Image-Guided Interventions: Technology and Applications, Springer, 2008.

[3] J. Tsao, Interpolation artifacts in multimodality image registration based on maximization of mutual information, IEEE Trans. Med. Imaging 22 (7) (2003) 854–864, doi:10.1109/TMI.2003.815077.

[4] J.P.W. Pluim, J.B.A. Maintz, M.A. Viergever, Interpolation artefacts in mutual information-based image registration, Comput. Vis. Image Underst. 77 (9) (2000) 211–232.

[5] GPGPU, Website of general purpose computation on the GPU, http://www.gpgpu.org.

[6] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Comput. Graph. Forum 26 (1) (2007) 80–113.

[7] R. Shams, P. Sadeghi, R.A. Kennedy, R.I. Hartley, A survey of medical image registration on multicore and the GPU, IEEE Signal Process. Mag. 27 (2) (2010) 50–60.

[8] A. Thall, Extended-precision floating-point numbers for GPU computation, in: SIGGRAPH’06: ACM SIGGRAPH 2006 Research Posters, ACM, New York, NY, USA, 2006, p. 52. doi:http://doi.acm.org/10.1145/1179622.1179682.

[9] C. Vetter, C. Guetter, C. Xu, R. Westermann, Non-rigid multi-modal registration on the GPU, in: Proceedings of SPIE Medical Imaging, vol. 6512, 2007, pp. 651228-1–651228-8.

[10] Z. Fan, C. Vetter, C. Guetter, D. Yu, R. Westermann, A. Kaufman, C. Xu, Optimized GPU implementation of learning-based non-rigid multi-modal registration, in: Proceedings of SPIE Medical Imaging, 2008, pp. 69142Y-1–169142Y-10.

[11] V. Podlozhnyuk, Histogram calculation in CUDA, Tech. rep., NVidia, 2007.

[12] R. Shams, N. Barnes, Speeding up mutual information computation using NVIDIA CUDA hardware, in: DICTA’07: Proceedings of the 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications, IEEE Computer Society, Washington, DC, USA, 2007, pp. 555–560.

[13] R.J. Rost, OpenGL Shading Language, 2nd edition, Addison-Wesley Professional, 2006.

[14] J. Neider, T. Davis, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[15] D. Blythe, The Direct3D 10 system, ACM Trans. Graph. 25 (3) (2006) 724–734, doi:http://doi.acm.org/10.1145/1141911.1141947.

[16] W.R. Mark, R.S. Glanville, K. Akeley, M.J. Kilgard, Cg: a system for programming graphics hardware in a C-like language, in: SIGGRAPH’03: ACM SIGGRAPH, ACM Press, New York, NY, USA, 2003, pp. 896–907, doi:http://doi.acm.org/10.1145/1201775.882362.

[17] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, ACM Trans. Graph. 23 (3) (2004) 777–786, doi:http://doi.acm.org/10.1145/1015706.1015800.

[18] J. Owens, Streaming architectures and technology trends, in: GPU Gems 2, Addison-Wesley Professional, 2005, pp. 457–470.

[19] AMD, CAL SDK, http://ati.amd.com/developer/.

[20] G.C. Sharp, N. Kandasamy, H. Singh, M. Folkert, GPU-based streaming architectures for fast cone-beam ct image reconstruction and demons deformable registration, Phys. Med. Biol. 52 (19) (2004) 5771–5783.

[21] Khronos Group, The OpenCL specification, http://www.khronos.org/opencl/.

[22] R. Strzodka, M. Droske, M. Rumpf, Fast image registration in DX9 graphics hardware, J. Med. Inform. Technol. 6 (2003) 43–49.

[23] A. Köhn, J. Drexl, F. Ritter, M. Koenig, H.-O. Peitgen, GPU accelerated image registration in two and three dimensions, in: Bildverarbeitung für die Medizin, Informatik Aktuell, Springer, 2006, pp. 261–265.

[24] N. Courty, P. Hellier, Accelerating 3D non-rigid registration using graphics hardware, Int. J. Image Graph. 8 (1) (2008) 1–18.

[25] P. Muyan-Özçelik, J.D. Owens, J. Xia, S.S. Samant, Fast deformable registration on the GPU: a CUDA implementation of demons, in: The 2008 International Conference on Computational Science and its Applications, ICCSA 2008, IEEE Computer Society, 2008, pp. 223–233.

[26] S.S. Samant, J. Xia, P. Muyan-Özçelik, J.D. Owens, High performance computing for deformable image registration: towards a new paradigm in adaptive radiotherapy, Med. Phys. 35 (8) (2008) 3546–3553.

[27] T. Schiwietz, J. Georgii, R. Westermann, Interactive model-based image registration, in: Proceedings of Vision, Modeling and Visualization 2007, 2007, pp. 213–221.

[28] K. Noe, K. Tanderup, J. Lindegaard, C. Grau, T. Sørensen, Gpu accelerated viscous-fluid deformable registration for radiotherapy, Stud. Health Technol. Inform. 132 (2008) 327–332.

[29] T. ur Rehman, G. Pryor, J. Melonakos, A. Tannenbaum, Multi-resolution 3d nonrigid registration via optimal mass transport on the gpu, Proc. Comput. Biomech. Med.-II (2007) 122–132.

[30] R. Strzodka, M. Droske, M. Rumpf, Image registration by a regularized gradient flow. A streaming implementation in DX9 graphics hardware, Computing 73 (4) (2004) 373–389.

[31] M. Grabner, T. Pock, T. Gross, B. Kainz, Automatic differentiation for GPU-accelerated 2D/3D registration, in: C.H. Bischof, H.M. Bücker, P. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, vol. 64 of Lecture Notes in Computational Science and Engineering, Springer, 2008, ISBN 978-3-540-68935-5, pp. 259–269.

[32] Y. Lin, G. Medioni, Mutual information computation and maximization using gpu, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08, 2008, pp. 1–6, doi:10.1109/CVPRW.2008.4563101.

[33] G.W. Sherhouse, K.L. Novins, E.L. Chaney, Computation of digitally reconstructed radiographs for use in radiotherapy treatment design, Int. J. Radiat. Oncol. Biol. Phys. 18 (3) (1990) 651–658.

[34] W. Wein, B. Roeper, N. Navab, 2D/3D registration based on volume gradients, in: Proceedings of SPIE Medical Imaging, 2005, pp. 144–150.

[35] D. LaRose, Iterative X-ray/CT registration using accelerated volume rendering, Ph.D. thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2001.

[36] T.J. Cullip, U. Neumann, Accelerating volume reconstruction with 3D texture hardware, Tech. rep., University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1994.

[37] J. Krüger, R. Westermann, Acceleration techniques for GPU-based volume rendering, in: Proceedings IEEE Visualization 2003, 2003, pp. 287–292.

[38] F. Ino, J. Gomita, Y. Kawasaki, K. Hagihara, A GPGPU Approach for Accelerating 2-D/3-D Rigid Registration of Medical Images, ISPA, 2006, pp. 939–950.

[39] F. Ino, Y. Kawasaki, T. Tashiro, Y. Nakajima, Y. Sato, S. Tamura, K. Hagihara, A parallel implementation of 2D–3D image registration for computer-assisted surgery, Int. J. Bioinform. Res. Appl. 2 (4) (2006) 341–358, doi:http://dx.doi.org/10.1504/IJBRA.2006.011034.

[40] S. Demirci, O. Kutter, F. Manstad-Hulaas, R. Bauernschmitt, N. Navab, Advanced 2D–3D registration for endovascular aortic interventions: addressing dissimilarity in images, in: Proceedings of SPIE Medical Imaging, San Diego, California, USA, 2008, pp. 69182S-1–169182S-8.

[41] D. Ruijters, B.M. ter Haar-Romeny, P. Suetens, GPU-accelerated digitally reconstructed radiographs, in: BioMED'08: Proceedings of the Sixth IASTED International Conference on Biomedical Engineering, ACTA Press, Anaheim, CA, USA, 2008, pp. 431–435.

[42] W. Birkfellner, R. Seemann, M. Figl, J. Hummel, C. Ede, P. Homolka, X. Yang, P. Niederer, H. Bergmann, Wobbled splatting—a fast perspective volume rendering method for simulation of X-ray images from ct, Phys. Med. Biol. 50 (9) (2005) N73–N84.

[43] B.K. Horn, B.G. Schunck, Determining optical flow, Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1980.

[44] X. Gu, H. Pan, Y. Liang, R. Castillo, D. Yang, D. Choi, E. Castillo, A. Majumdar, T. Guerrero, S.B. Jiang, Implementation and evaluation of various demons deformable image registration algorithms on a gpu, Phys. Med. Biol. 55 (1) (2010) 207, URL http://stacks.iop.org/0031-9155/55/i=1/a=012.

[45] A. Khamene, C. Schaller, J. Hornegger, J.C. Celi, B. Ofstad, E. Rietzel, X.A. Li, A. Tai, J. Bayouth, A novel image based verification method for respiratory motion management in radiation therapy, in: Eleventh IEEE International Conference on Computer Vision, DVD Proceedings, 2007, pp. 1–7.

[46] O. Fluck, S. Aharon, A. Khamene, Efficient framework for deformable 2d–3d registration, in: Proceedings of SPIE Medical Imaging, vol. 6918, SPIE, 2008, pp. 69181E-1–69181E-10, doi:10.1117/12.772911.

[47] C. Rezk-Salama, M. Scheuering, G. Soza, G. Greiner, Fast volumetric deformation on general purpose hardware, in: Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2001, pp. 17–24.

[48] G. Soza, P. Hastreiter, M. Bauer, C. Rezk-Salama, C. Nimsky, G. Greiner, Intraoperative registration on standard PC graphics hardware, in: Bildverarbeitung für die Medizin, 2002, pp. 334–337.

[49] G. Soza, M. Bauer, P. Hastreiter, C. Nimsky, G. Greiner, Non-rigid registration with use of hardware-based 3D Bézier functions, in: MICCAI'02: Proceedings of the 5th International Conference on Medical Image Computing and Computer-Assisted Intervention-Part II, Springer-Verlag, London, UK, 2002, pp. 549–556.

[50] H. Chen, J. Hesser, R. Männer, Fast free-form volume deformation using inverse-ray-deformation, in: VIIP, ACTA Press, 2001, pp. 163–168.

[51] B. Li, A.A. Young, B.R. Cowan, Gpu accelerated non-rigid registration for the evaluation of cardiac function, in: MICCAI 2008: Proceedings of the 11th International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 880–887.

[52] D. Ruijters, B.M. ter Haar-Romeny, P. Suetens, Efficient GPU-accelerated elastic image registration, in: BioMED'08: Proceedings of the Sixth IASTED International Conference on Biomedical Engineering, ACTA Press, Anaheim, CA, USA, 2008, pp. 419–424.

[53] 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, Comput. Methods Prog. Biomed. 98 (3) (2010) 278–284, doi:http://dx.doi.org/10.1016/j.cmpb.2009.09.002.

[54] NVidia, CUDA SDK, http://www.nvidia.com/object/cuda_get.html.

[55] A. Ruiz, M. Ujaldon, L. Cooper, K. Huang, Non-rigid registration for large sets of microscopic images on graphics processors, J. Signal Process. Syst. 55 (1–3) (2009) 229–250, doi:http://dx.doi.org/10.1007/s11265-008-0208-4.

[56] A. Kubias, F. Deinzer, T. Feldmann, S. Paulus, D. Paulus, B. Schreiber, T. Brunner, 2d/3d image registration on the gpu, Int. J. Pattern Recognit. Image Anal. 18 (3) (2008) 381–389, URL http://springerlink.com/content/l173468423228282/.

[57] A. Khamene, P. Bloch, W. Wein, M. Svatos, F. Sauer, Automatic registration of portal images and volumetric CT for patient positioning in radiation therapy, Med. Image Anal. (2006) 96–112.

[58] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Suetens, G. Marchal, Automated multi-modality image registration based on information theory, Inf. Process. Med. Imaging (1995) 263–274.

[59] W. Wells, P. Viola, H. Atsumi, S. Nakajima, R. Kikinis, Multi-modal volume registration by maximization of mutual information, Med. Image Anal. 1 (1) (1996) 35–51.

[60] C. Guetter, C. Xu, F. Sauer, J. Hornegger, Learning based non-rigid multi-modal image registration using kullback-leibler divergence, in: MICCAI, no. 2, 2005, pp. 255–262.

[61] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, P. Suetens, Multimodality image registration by maximization of mutual information, IEEE Trans. Med. Imaging 16 (2) (1997) 187–198, doi:10.1109/42.563664.

[62] W.-H. Cheng, C.-C. Lu, Acceleration of medical image registration using graphics process units in computing normalized mutual information, in: Fifth International Conference on Image and Graphics, 2009. ICIG'09, 2009, pp. 814–818, doi:10.1109/ICIG.2009.48.

[63] NVidia, NVIDIA CUDA – compute unified device architecture – programming guide – version 2.3, Tech. rep., NVidia, 2007.

[64] C. Sigg, M. Hadwiger, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005, Ch. Fast Third-Order Texture Filtering, pp. 313–329.

[65] D. Ruijters, B.M. ter Haar Romeny, P. Suetens, Efficient gpu-based texture interpolation using uniform b-splines, J. Graph. gpu Game Tools 13 (4) (2008) 61–69.

[66] D. Horn, Stream reduction operations for GPGPU applications, in: GPU Gems 2, Addison-Wesley Professional, 2005, pp. 573–589.

[67] D. Roger, U. Assarsson, N. Holzschuch, Efficient stream reduction on the GPU, in: Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[68] M. Harris, Optimizing parallel reduction in cuda, Tech. rep., NVIDIA Corporation, 2008.

[69] ARB, EXT_histogram. URL http://opengl.org/registry/specs/EXT/histogram.txt.

[70] P. Hastreiter, T. Ertl, Integrated registration and visualization of medical image data, in: Proceedings of Computer Graphics International (CGI), 1998, pp. 78–85.

[71] M. Teßmann, C. Eisenacher, F. Enders, M. Stamminger, P. Hastreiter, Gpu accelerated normalized mutual information and b-spline transformation, in: Proceedings of the Eurographics Workshop on Visual Computing for Biomedicine (EG VCBM), The Eurographics Association, 2008, pp. 117–124.

[72] O. Fluck, S. Aharon, D. Cremers, M. Rousson, GPU histogram computation, in: SIGGRAPH'06: ACM SIGGRAPH 2006 Research Posters, 2006.

[73] T. Scheuermann, J. Hensley, Efficient histogram generation using scattering on GPUs, in: ACM Symposium on Interactive 3D Graphics and Games, 2007, pp. 33–37.

[74] R. Shams, R.A. Kennedy, Efficient histogram algorithms for NVIDIA CUDA compatible devices, in: Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS), Gold Coast, Australia, 2007, pp. 418–422.

[75] M. Ohara, H. Yeo, F. Savino, G. Iyengar, L. Gong, H. Inoue, H. Komatsu, V. Sheinin, S. Daijavaa, B. Erickson, Real-time Mutual-information-based Linear Registration on the Cell Broadband Engine Processor, 2007, pp. 33–36. doi:10.1109/ISBI.2007.356781.

[76] J. Rohrer, L. Gong, Accelerating 3d nonrigid registration using the cell broadband engine processor, IBM J. Res. Dev. 53 (5) (2009), 12:1–12:10. doi:10.1147/JRD.2009.5429078.

[77] D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, Int. J. Parallel Emergent Distributed Syst. 22 (4) (2007) 221–256, doi:10.1080/17445760601122076.

[78] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, J. Rattner, Technology Intel Magazine: Platform 2015: Intel Processor and Platform Evolution for the Next Decade, 2005, http://www.intel.com/technology/magazine/.

[79] P. Hastreiter, C. Rezk-Salama, G. Soza, G. Greiner, R. Fahlbusch, O. Ganslandt, C. Nimsky, Strategies for brain shift evaluation, Med. Image Anal. 8 (4) (2004) 447–464.

[80] T. ur Rehman, E. Haber, G. Pryor, J. Melonakos, A. Tannenbaum, 3d nonrigid registration via optimal mass transport on the gpu, Med. Image Anal. 13 (6) (2009) 931–940.