An OpenCL implementation of the Gaussian pyramid and the resampler

Release 1.00

Denis P. Shamonin[1] and Marius Staring[1]

December 18, 2012

[1]Leiden University Medical Center, The Netherlands

Abstract

Nonrigid image registration is an important, but resource demanding and time-consuming task in medical image analysis. This limits its application in time-critical clinical routines. In this report we explore acceleration of two time-consuming parts of a registration algorithm by means of parallel processing using the GPU. We built upon the OpenCL-based GPU image processing framework of the recent ITK4 release, and implemented Gaussian multi-resolution strategies and a general resampling framework. We evaluated the performance gain on two multi-core machines with NVidia GPUs, and compared to an existing ITK4 CPU implementation. A speedup factor of $\sim$ 2-4 was realized for the multi-resolution strategies and a speedup factor of $\sim$ 10-46 was achieved for resampling, for larger images ($\approx 10^8$ voxels).

Latest version available at the Insight Journal [`http://hdl.handle.net/10380/3393`]

Contents

1 Introduction

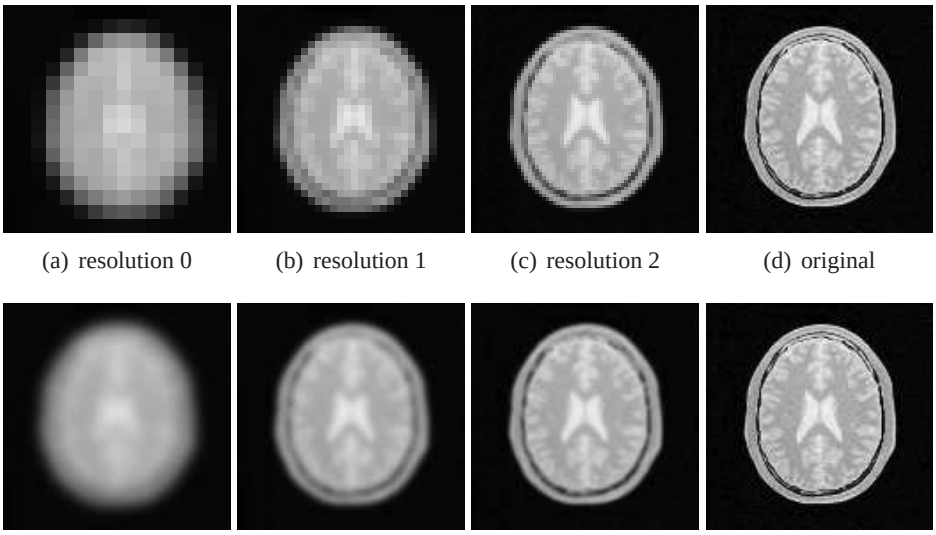
Image registration is the process of aligning images, and can be defined as an optimization problem:

$$\hat{\boldsymbol{\mu}} = \arg\min_{\boldsymbol{\mu}} \mathcal{C}(I_F, I_M; \boldsymbol{\mu}), \tag{1}$$

with I_F and I_M the d-dimensional fixed and moving image, respectively, and $\boldsymbol{\mu}$ the vector of parameters of size N that model the transformation $\boldsymbol{T}$. The cost function $\mathcal{C}$ consists of a similarity measure $\mathcal{S}(I_F, I_M; \boldsymbol{\mu})$ that defines the quality of alignment. Examples are the mean square difference (MSD), normalised correlation (NC), and mutual information (MI) measure. Image registration is usually embedded in a multi-resolution framework, and after the optimization procedure (1) has finished a resampling of the moving image is usually desired to generate the registration result $I_M(\boldsymbol{T}_{\hat{\boldsymbol{\mu}}})$.

Currently existing nonrigid image registration methods have poor computational performance. The poor computational performance is in contrast with the great demand for image registration in several time-critical clinical applications. Compensation of this motion needs to be online, i.e. in a few seconds, but is currently unavailable.

In this document we describe GPU-based solutions that address the run-time of two parts of the registration framework: the Gaussian multi-resolution framework and the resampling framework. To develop GPU acceleration of these two image registration steps, the open standard OpenCL (Open Computing Language)



(a) resolution 0 (b) resolution 1 (c) resolution 2 (d) original

(e) resolution 0,$\sigma = 8.0$ (f) resolution 1,$\sigma = 4.0$ (g) resolution 2,$\sigma = 2.0$ (h) original

Figure 1: Two multi-resolution strategies using a Gaussian pyramid ($\sigma = 8.0, 4.0, 2.0$ voxels). The first row shows multi-resolution with down-sampling (`FixedRecursiveImagePyramid`), the second row without (`FixedSmoothingImagePyramid`). Note that in the first row, for each dimension, the image size is halved every resolution, but that the voxel size increases with a factor 2, so physically the images are of the same size every resolution.

was chosen because of its wide applicability on graphical cards (Intel, AMD, and NVIDIA). The Gaussian pyramid implementation acted as a learning example to OpenCL, since it mostly consists of Gaussian blurring, which is a straightforward filtering technique comparing to resampling. This work is addressed in the context of the open source registration software `elastix`.

The remainder of the paper is organized as follows. We first provide more detail on the pyramids and resampling step in Section 2. In addition, the rationale for the focus on these two components is given in Section 3. Then, the OpenCL architecture and the ITK4 GPU acceleration framework is revisited in Section 4. Section 5 gives details on the GPU programming designs. The experiment results with real images are given in Section 6, and Section 7 is the conclusion.

2 Methods

It is common to start the registration process (1) using images that have lower complexity, e.g., images that are smoothed and optionally downsampled. This increases the chance of successful registration. After computation of these lower complexity images, the core of the registration is started, i.e. Equation (1) is solved by some optimization scheme (usually a gradient descent like scheme). At the end of this optimization the resulting image is computed, using what is called a resampling step.

2.1 Multi-resolution: Gaussian image pyramids

There are several ways of computing the lower complexity images. A series of images with increasing amount of smoothing is called a scale space. If the images are not only smoothed, but also downsampled,

the data is not only less complex, but the *amount* of data is actually reduced. Several scale spaces or pyramids are found in the literature, amongst others Gaussian and Laplacian pyramids, morphological scale space, and spline and wavelet pyramids. The Gaussian pyramid is by far the most common one for image registration, and the computation of this pyramid we target to accelerate. Figure 1 shows the Gaussian pyramid with and without downsampling. In `elastix` we have three kinds of pyramids:

Gaussian pyramid: (`FixedRecursiveImagePyramid` and `MovingRecursiveImagePyramid`) Applies smoothing and down-sampling.

Gaussian scale space: (`FixedSmoothingImagePyramid` and `MovingSmoothingImagePyramid`) Applies smoothing and *no* down-sampling.

Shrinking pyramid: (`FixedShrinkingImagePyramid` and `MovingShrinkingImagePyramid`) Applies *no* smoothing, but only down-sampling.

2.2 Image resampling

Resampling is the process of computing the value $I_M(\boldsymbol{T}(\boldsymbol{x}))$ for every voxel $\boldsymbol{x}$ inside some domain. Usually, the fixed image domain Ω_F is chosen, meaning that the computational complexity is linearly dependent on the number of voxels in the fixed image. The procedure is simple: 1) loop over all voxels $\boldsymbol{x} \in \Omega_F$, 2) compute its mapped position $\boldsymbol{y} = \boldsymbol{T}(\boldsymbol{x})$, 3) since $\boldsymbol{y}$ is generally a non-voxel position, intensity interpolation of the moving image at $\boldsymbol{y}$ is needed, and 4) fill in this value at $\boldsymbol{x}$ in the output image.

Notice from above that the procedure is dependent on a choice of the interpolator and the transform. Several methods for interpolation exist, varying in quality and speed. Examples also available in `elastix` are nearest neighbor, linear and B-spline interpolation. Nearest neighbor interpolation is the most simple technique, low in quality, requiring little resources. The intensity of the voxel nearest in distance is returned. The B-spline interpolation quality and complexity depends on its order: 0 order equals nearest neighbor, 1-st order equals linear interpolation, and higher order generally gives better quality. The higher the order, the higher the computational complexity. For resampling usually an order of 3 is used. There are also many flavors of transformations. The ones available in `elastix` in order of increasing flexibility, are the translation, the rigid, the similarity, the affine, the nonrigid B-spline and the nonrigid thin-plate-spline-like transformations.

3 Preliminaries

3.1 Project focus

In this project we focus on accelerating the image registration procedure by means of parallelization of certain components, exploiting the GPU. As this is our first encounter with GPU programming we identified two independent components that are time-consuming and allow for parallelism: the Gaussian pyramids and the resampling step. Both components are intrinsically parallellizable: The Gaussian filtering relies on a line-by-line causal and anti-causal filtering, where all image scan lines can be independently processed; The resampling step requires for every voxel the same independent operation (transformation followed by interpolation). The Gaussian filtering does consume some time, but not really a lot and is a good starting point to learn GPU programming. Depending on the input images sizes, transformations and interpolation the resampling part can take a considerable time of the total registration for large images, sometimes even dominating the runtime. In this project we did not focus on the core of the registration algorithm.

We aim for high quality software that is portable, open source, with the code separated into the distinct components for maintainability.

3.2 elastix, ITK and CMake

Parallelization is performed in the context of the image registration software, called elastix [7], available at http://elastix.isi.uu.nl. The software is distributed as open source via periodic software releases under a BSD license, which means that it can be used by other researchers and industry for free, without any restrictions. The software consists of a collection of algorithms that are commonly used to solve (medical) image registration problems. The modular design of elastix allows the user to quickly configure, test, and compare different registration methods for a specific application. A command-line interface enables automated processing of large numbers of data sets, by means of scripting.

elastix is based on the open source Insight Segmentation and Registration Toolkit (ITK) [6] available at www.itk.org. This library contains a lot of image processing functionality, and delivers an extremely well tested coding framework. It is implemented in C++, nightly tested, has a rigorous collaboration process, and works on many platforms and compilers. The use of the ITK in elastix implies that the low-level functionality (image classes, memory allocation, etc.) is thoroughly tested. Naturally, all image formats supported by the ITK are supported by elastix as well. elastix can be compiled on multiple operating systems (Windows, Linux, Mac OS X), using various compilers (MS Visual Studio, GCC), and supports both 32 and 64 bit systems.

Both elastix and ITK employ the CMake build system[3]. This amongst others allows easy incorporation of external components on all the supported platforms, such as the OpenCL library.

The Kitware Wiki [2] provides information to understand the outline of the GPU acceleration framework. Read more about OpenCL at http://www.khronos.org/opencl/.

3.3 GPU programming platform

There are two major programming frameworks for GPU computing, i.e. OpenCL and CUDA, which have been competing in the developer community for the past few years. Until recently, CUDA has attracted most of the attention from developers, especially in the high performance computing realm. However, the OpenCL software has now matured to the point where developers are taking a second look.

Both OpenCL and CUDA provide a general-purpose model for data/task parallelism, but only OpenCL provides an open, industry-standard framework. As such, it has garnered support from nearly all processor manufacturers including AMD, Intel, IBM and NVIDIA, as well as others that serve the mobile and embedded computing markets. As a result, applications developed in OpenCL are now portable across a variety of GPUs and CPUs.

In this project we decided to adopt OpenCL for algorithm implementation for two reasons: i) OpenCL solutions are independent of the GPU hardware vendor available at the elastix-user site, thereby broadening the applicability of this work; ii) Our image registration package elastix is largely based on the Insight Toolkit (ITK) [6], who recently adopted OpenCL.

4 GPU programming with OpenCL

4.1 GPUs

Multi-core computers and hyper-threading technology have enabled the acceleration of a wide variety of computationally intensive applications. Nowadays, another type of hardware promises even higher computational performance: the graphics processing unit (GPU). Originally used to accelerate the building of images in a frame buffer intended for output to a display, GPUs are increasingly applied to scientific calculations. Unlike a traditional CPU, which includes no more than a few cores, a programmable GPU has a highly parallel structure, as well as dedicated, high-speed memory. This makes them more effective than general purpose CPUs for algorithms where processing of large blocks of data is done in parallel.

The increasing computing power of GPUs gives them considerably higher peak computing power than CPUs. For example, NVIDIAs GTX280 GPU provide 933 Gflop/s with 240 SIMD cores, the GeForce GTX 680 provides 1581 Gflop/s with 1536 SIMD cores while Intels Xeon processor X5675 (3.06GHz 6-cores) reaches 144 Gflop/s. Intels next generation of graphics processors will support more than 900 Gflop/s and AMDs latest GPU HD7970 provides 3788 Gflop/s 2048 SIMD cores.

Writing parallel programs to take full advantage of this GPU power is still a big challenge. The execution time of an application is sometimes dominated by the latency of memory instructions. Optimizing usage of memory accesses, memory hierarchy, threads and clearly understanding various features of the underlying architecture could improve application performance.

4.2 OpenCL

The OpenCL C programming language (http://www.khronos.org/opencl/) is used to create programs that describe data-parallel kernels and tasks that can be executed on one or more heterogeneous devices such as CPUs, GPUs, FPGAs and potentially other devices developed in the future. An OpenCL program is similar to a dynamic library, and an OpenCL kernel is similar to an exported function from the dynamic library. Applications cannot call an OpenCL kernel directly, but instead queue the execution of the kernel to a command-queue created for a device. The kernel is executed asynchronously with the application code running on the host CPU. OpenCL is based on the ISO/IEC 9899:1999 C language specification (referred to in short as C99) with some restrictions and specific extensions to the language for parallelism. There is a standard defined for the C++ Bindings but this is only to wrap the OpenCL 1.2 C API in classes. Note that C++ on the kernels level is not supported by the standard, posing restrictions on the implementation of kernels.

OpenCL is an open standard and was initially developed by Apple Inc., which holds trademark rights, and refined into an initial proposal in collaboration with technical teams at AMD, IBM, Intel, and NVidia. OpenCL is maintained by the non-profit technology consortium Khronos Group. The latest OpenCL 1.2 specification has been released November 2011, the updated OpenCL 1.2 specification revision 19, been released November 14, 2012.

Availability

It is possible to install several OpenCL implementations (*platforms*) at the same system, to develop against any one of them, and then choose at run time which devices from which platforms to use. There are five OpenCL implementations available at this time: 1) AMD (supports both CPUs and AMD GPUs), 2) Apple

(supports both CPU and GPU), 3) Intel (supports Intel CPUs and GPUs, Intel HD Graphics 4000/2500), 4) NVidia (supports only GPUs), 5) IBM (supports CPUs, IBM's Power processor). Our implementation targets the first four systems.

The OpenCL Architecture

The OpenCL architecture has a strict specification and uses a hierarchy of four main ingredients (**models**):

Platform model: Defines the relationship between the **host** and the **device**. An OpenCL application runs on a host (CPU) and submits commands from the host to execute computations on the processing elements within a device (GPU or CPU). Examples of available platforms are Intel, NVidia, AMD, and within these platforms the available **devices**: NVidia with GeForce GTX 260, Quadro FX 1800; AMD with Radeon HD6630M, HD7970 or the AMD Phenom II x4 CPU. The user or application decides which device to use as an accelerator.

Execution model: Defines a flexible execution model that incorporates both task and data parallelism which are coordinated via **command queues**. The command queues provide a general way of specifying relationships between tasks, ensuring that tasks are executed in-order or out-of-order. Using this model the OpenCL application gets configured on the host and it is instructed how kernels are executed on the device. This includes setting up an OpenCL context on the host for the execution of the kernels, sets an memory objects visible to the host and the devices, and defines a command queue (in-order execution, out-of-order execution) to coordinate the execution.

Memory model: Defines the memory hierarchy that kernels use, regardless the actual underlying memory architecture for the device. Four distinct memory regions exists (Global Memory, Local Memory, Constant Memory and Private Memory). Each compute device has a global memory, which is the largest memory space available to the device, and typically resides in off-chip DRAM (NVidia GeForce GTX 260 has 896MB). There is also a read-only, limited-size constant memory space (NVidia GeForce GTX 260 has 64Kb), which allows for efficient reuse of read-only parameters in a computation. Each compute unit on the device has a local memory (NVidia GeForce GTX 260 has 16Kb), which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Additionally, each processing element has private memory, which is typically not used directly by programmers, but is used to hold data for each work-item that does not fit in the processing elements registers. To achieve maximum performance of the application different memories are used.

Programming model: The OpenCL execution model supports data parallel and task parallel programming models. Synchronization is achieved using command queue barriers and waiting on events.

A typical OpenCL application starts by queueing available platforms (Intel/NVidia/AMD), and within each platform the available devices (GeForce GTX 260, Quadro FX 1800; the user or the application decides which device to use as an accelerator). The platform model defines this relationship between the host and device. The application allocates and transfers memory from CPU to GPU, kernels allocate local or private memory. The host and OpenCL device memory models are, for the most part, independent of each other. The interaction occurs in one of two ways: by explicitly copying data or by mapping and unmapping regions of a memory object. The OpenCL supports two patterns of memory access (Write/Execute/Read) and (Unmap/Execute/Map). Choosing a pattern is based on application needs, the goal is to minimize copies/allocations. If the application receives and sends buffers with varying addresses, choose read/writes,

if the application processes the buffer (for example, analyze it), choose map/unmap to avoid additional memory allocation. All this happens within the memory model. Kernels have to be compiled or loaded from binaries and subsequently set for the selected device. This happens at runtime and is necessary because you may not know in advance just what sort of platform you're going to target. In case of success, kernels are scheduled to execute on the device. This gets controlled within the execution model. Finally, hardware thread contexts that execute the kernel must be created and mapped to actual GPU hardware units. This is done using the programming model.

4.3 ITK4 and GPU acceleration

The ITK4 GPU Acceleration module wraps the OpenCL 1.1 API in an ITK4-style API. It takes care about the OpenCL initialization, program compilation, and kernel execution. It also provides convenience classes for interfacing to ITK image classes and filtering pipeline such as `itk::GPUImage`, `itk::GPUImageToImageFilter` and `itk::GPUInPlaceImageFilter`. The ITK GPU acceleration is still a work in progress, suggestions and patches are welcomed to make it better.

To enable GPU support within ITK4 you should enable the `ITK_USE_GPU` flag during CMake configuration of ITK. The core ITK4 OpenCL architecture (platform model, execution model, memory model, see Section 4.2) is presented by the following classes

- `itk::GPUContextManager`: Manages context and command queues.
- `itk::GPUKernelManager`: Load, setup, compile and run OpenCL kernels.
- `itk::GPUDataManager`: Provides functionalities for CPU-GPU data synchronization.

One of the core requirements of ITK is its ability to create data flow pipelines that are capable of ingesting, processing, analyzing and streaming data. ITK is organized around **data objects** and **process objects**. A pipeline is a series of process objects that operate on one or more data objects. The data objects "flows" along the pipeline. The core pipeline classes `itk::GPUImage`, `itk::GPUImageToImageFilter` and `itk::GPUInPlaceImageFilter` are responsible for combining CPU and GPU filters and efficient CPU/GPU pipeline synchronization. The generic design of the `itk::GPUImageToImageFilter` allows extending existing ITK filters for GPU implementation. ITK4 uses a commonly accepted design pattern to instantiate GPU filters in a program using an object factory method.

In order to make a GPU filter you have to first inherit your filter from `itk::GPUImageToImageFilter`, create OpenCL kernel code and register the GPU filter in your program.

4.4 Modifications to ITK4's OpenCL support

For this project several enhancements to ITK4's GPU code were needed.

Find OpenCL

To build ITK for your system the cross-platform, open-source build system CMake [3] is used. In order to locate OpenCL on the system (CL/cl.h, OpenCL.lib) the `FindOpenCL.cmake` has to be defined. Currently the `FindOpenCL` module is not part of CMake's standard distribution due to the under development status. For our work we modified an existing module [8]. The complete list of added functionality is found in Appendix A.

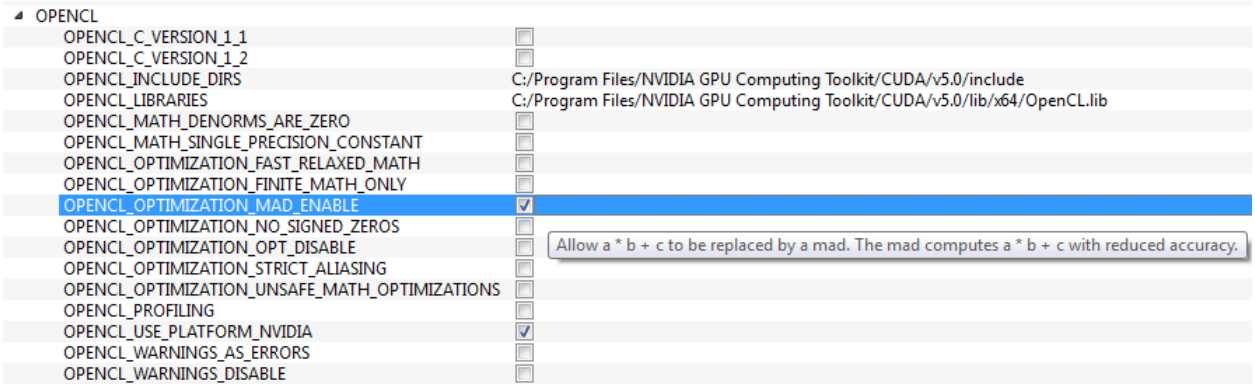


Figure 2: CMake OpenCL options to control platform, math intrinsics and optimization options.

We added support for a number of build options categorized as pre-processor options, options for math intrinsics, options that control optimization and miscellaneous options. These CMake options are passed to the `clBuildProgram()` command in the `itk::GPUKernelManager` to compile OpenCL kernels. In this way we are able to control building and optimization of an OpenCL application similar to the compile settings `CMAKE_<C,CXX>_FLAGS` used for C/C++. The process of configuring is illustrated in Figure 2. These options allow us to develop under different platforms as well as control a number of optimizations.

Modifications to ITK core GPU classes

For our implementation we copied a number of classes from ITK 4.1.0 and introduced some changes. The changes introduced to these copied classes may later be merged back to the ITK mainstream. Here is a complete list of the copied files:

```
itkGPUContextManager.(cxx,h)          itkGPUDataManager.(cxx,h)
itkGPUFunctorBase.h                   itkGPUImage.(txx,h)
itkGPUImageDataManager.(h,txx)        itkGPUImageToImageFilter.(h,txx)
itkGPUInPlaceImageFilter.(h,txx)      itkGPUKernelManager.(cxx,h)
itkGPUUnaryFunctorImageFilter.(h.hxx) itkOclUtil.(cxx,h)
```

The following main functionality has been added:

- Support for Intel/AMD/NVidia OpenCL platforms. We extended the `itk::GPUContextManager` to handle other platforms via CMake configuration.
- Possibility to Debug OpenCL kernels with Intel's OpenCL implementation. The Intel SDK for OpenCL supports Microsoft Visual Studio Debugger via a plug-in interface. This enables us to debug into OpenCL kernels using the familiar graphical interface of the Microsoft Visual Studio software debugger [1].
- OpenCL execution profiling. Event objects can be used to capture profiling information that measure execution time of OpenCL commands. Profiling of OpenCL commands can be very handy in understanding performance bottlenecks of GPU architectures.
- CMake math intrinsics and optimization options for OpenCL compilation (See section 4.4). These options controls the optimizations for floating-point arithmetic.
- Work-around for incorrect `itk::GPUImage::Graft()` implementation. The original implementation [8] contains some logical error in one of the key ITK pipeline methods.

- Some protected variables of the itk::GPUImage class have been made publicly accessible with Get-methods (IndexToPhysicalPoint, PhysicalPointToIndex). These functions are needed for the resampler to convert from index to physical world.
- itk::GPUImageToImageFilter::GenerateData() has been modified to resemble CPU ITK pipeline execution.
- The itk::OpenCLSize has been introduced to itk::GPUKernelManager::LaunchKernel() to support configurable kernel execution logic with global_work_size, local_work_size and global_work_offset.
- Two new classes itk::OpenCLEvent and itk::OpenCLEventList were introduced. These event objects are used to synchronize execution of multiple kernels, in case a filter requires multiple kernels to be scheduled.
- The signature of the itk::GPUKernelManager::LaunchKernel() function was modified to return itk::OpenCLEvent instead of simply a boolean. LaunchKernel() is basically an ITK wrap around an OpenCL function that enqueues the kernel, but no guarantee is given about the order of execution. Therefore, flavors of LaunchKernel() were added that wait for an event list to have finished before executing the current kernel. This is useful to support in-order execution of lists of kernels, which is essential for complex OpenCL filtering operations like the itk::GPUResampleImageFilter.
- We added locking mechanisms for itk::GPUDataManager to prevent unnecessary updates of CPU/GPU buffers.
- A number of modifications have been made to the itk::GPUKernelManager to improve design, code, debugging support and integrate CMake OpenCL math and optimization options.

The changes, bugs and other modifications has been reported to the ITK Jira bug tracking system, see https://issues.itk.org/jira/browse/ITK.

4.5 OpenCL building program

We would like to highlight a particular issue you may encounter during developing. The final assembly code from OpenCL kernels will be generated by the NVidia or AMD compiler at the moment clBuildProgram is called. The compilation often happens from text strings constructed at runtime. The NVidia compiler will cache assembly kernels in /AppData/Roaming/NVIDIA/ComputeCache (Windows) or ~/.nv/ (Unix), so that after the first time the program is called the cached binary can be used. Compiling a simple kernel could take up to 0.5 s for the first time, while loading the same kernel from cache takes almost nothing (10^{-4} s). Try to avoid defining kernels that may change at runtime due to a user setting such as the image properties, constant variables (_constant) or defines (#define). This may create unexpected performance bottlenecks in your programm by triggering recompilation of the kernel every time a new image, parameters or definitions are passed. Note that OpenCL also allows applications to create a program object as a pre-built offline binary. This was however not explored in this project.

5 Accelerating the two image registration components

5.1 Unify the existing Gaussian image pyramids

As described in Section 2.1 there are three Gaussian pyramids available in elastix. Each of them has a different way of dealing with smoothing and resizing of the data. The recursive pyramid smoothes according

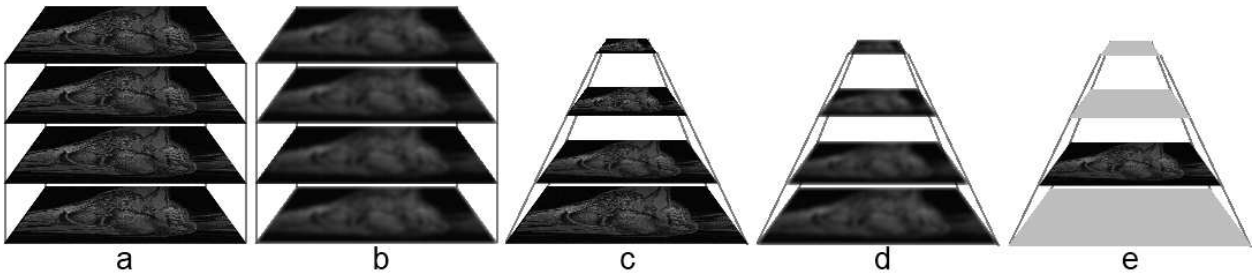


Figure 3: Multi-resolution pyramids with four levels. (a) The rescale and smoothing schedule are not used; (b) Only a smoothing schedule is used; (c) Only the rescale schedule is used; (d) Both the rescale and smoothing schedule are used; (e) Memory consumption option is used to create the image at level 1, while other images have not been allocated.

to a schedule, and applies a fixed resizing derived from the smoothing schedule; The smoothing pyramid smoothes and does not perform resizing; The shrinking pyramid does not perform smoothing and only performs resizing according to a resizing schedule. As a first step in this project we unified the three Gaussian pyramids to a single class that separately takes a smoothing schedule and a down-sampling schedule. The new 'generic' pyramid is implemented in C++ in ITK style in a multi-threaded fashion, using the CPU. We have dubbed this class the `itk::GenericMultiResolutionGaussianImageFilter`.

The multi-resolution rescale schedule is specified in terms of shrink factors (unit-less) at each multi-resolution level for each dimension. The smoothing schedule defines the standard deviation of the Gaussian kernel (in millimeters), again for each resolution level and for each dimension. As an additional feature, we introduced an option to only compute the pyramid results for a given current resolution level. The previous pyramids computed the results for all levels, and stored all data in memory, thereby having a large memory footprint. The filter results is illustrated at Figure 3.

This new module was tested and is already integrated in `elastix`; it can be found in the directories

```
src/Components/FixedImagePyramids/FixedGenericPyramid
src/Components/MovingImagePyramids/MovingGenericPyramid
```

5.2 Gaussian pyramid GPU implementation

To implement the smoothing of images we have used the `itk::RecursiveGaussianImageFilter` available in ITK. The filter computes an infinite impulse response convolution with an approximation of the Gaussian kernel

$$G(x;\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right). \tag{2}$$

So, it implements the recursive filtering method proposed by Deriche *et al.* [4]. The filter smoothes the image in a single, user-defined direction only. It should be called for each direction in a sequence to perform full smoothing. The main control methods are `SetSigma()` and `SetDirection()`; the latter sets the direction in which the filter is to be applied.

The filter takes an `itk::Image` as input, and produces a smoothed `itk::Image` as output. The filter is implemented as a subclass of the `itk::InPlaceImageFilter` and thus can be executed

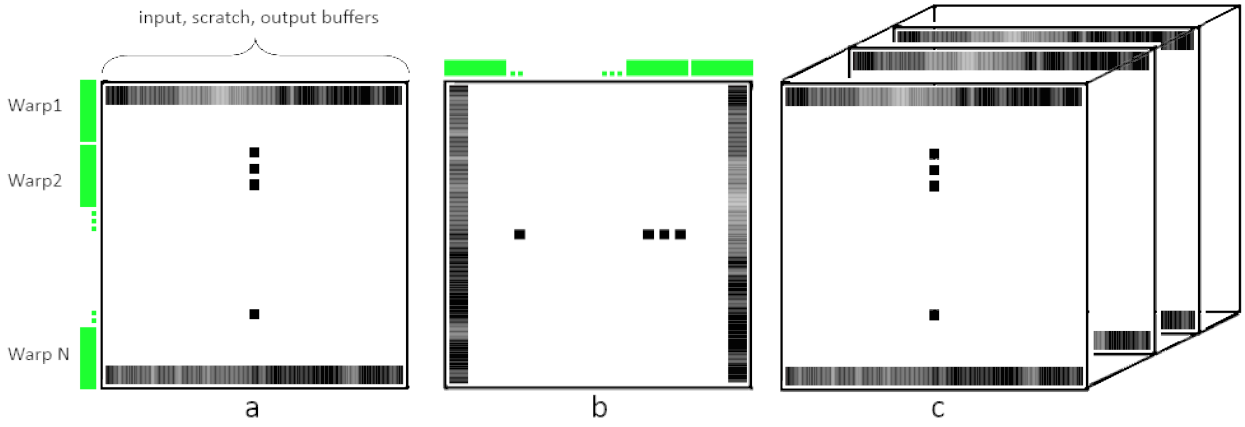


Figure 4: `itk::GPURecursiveGaussianImageFilter` threads alignment for the output image: (a) The 2D image for direction *x*; (b) The 2D image for direction *y*; (c) The 3D image direction *x*.

in place, saving memory (not the default). Therefore, for the GPU implementation we derived our `itk::GPURecursiveGaussianImageFilter` from `itk::GPUInPlaceImageFilter` to support this behavior on the GPU as well.

To parallelize the work load, the image is split in several regions and each thread works on its own region. The `itk::RecursiveGaussianImageFilter` performs execution row-by-row for the direction *x* or column-by-column for the direction *y*. All rows or columns can be processed independently, but columns can only be processed when all rows have finished. Figure 4 illustrates the process. The internal algorithm implementation allocates an input and output buffer the size of the row or column, and additionally requires a scratch buffer of the same size. The filtering kernel performs a causal and anti-causal pass, and the result is copied to the output. This execution model is suitable for a GPU implementation, where several rows or columns can be executed simultaneously.

To achieve maximum performance each thread uses the local GPU memory, although this introduces a limitation on the input image size, since only 16kB is available (see Section 4.2). The maximum supported image size is then calculated as follows: there are three floating point buffers (input, scratch, output), the maximum space per buffer is then 16kB divided by three, which equals a maximum of 1365 pixels. In other words, the current implementation works for images of maximum size [1365,1365] or [1365,1365,1365]. This limitation could be avoided by changing the internal computation or by using other platforms with a larger local memory (for example Intel's OpenCL implementation allows 32kB).

5.3 Image resampling

As noted in Section 2.2, resampling is the procedure to create a new version of the (moving) image that is geometrically transformed and possibly has a different size and resolution. To perform this operation `elastix` uses the `itk::ResampleImageFilter` from the ITK. This filter resamples an existing image through a specified transform provided with `SetTransform()` and interpolate via some image function provided with `SetInterpolator()`. There are many different transformation classes available (translation, rigid, affine, B-spline)[1], as well as a number of interpolation techniques (nearest neighbor, linear, B-spline)[2].

[1] See the directory `ITK/Modules/Core/Transform` from the ITK repository for more examples
[2] See the directory `ITK/Modules/Core/ImageFunction` from the ITK repository for more examples

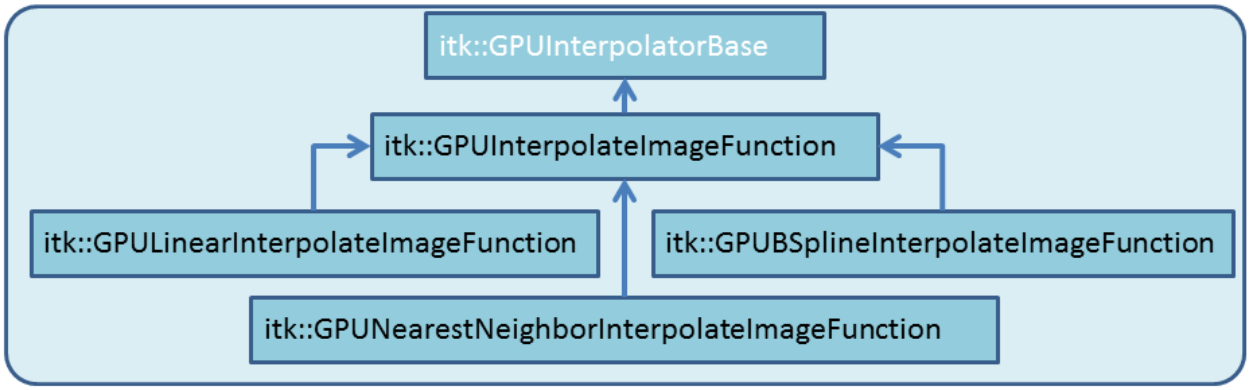


Figure 5: Design for GPU Interpolators.

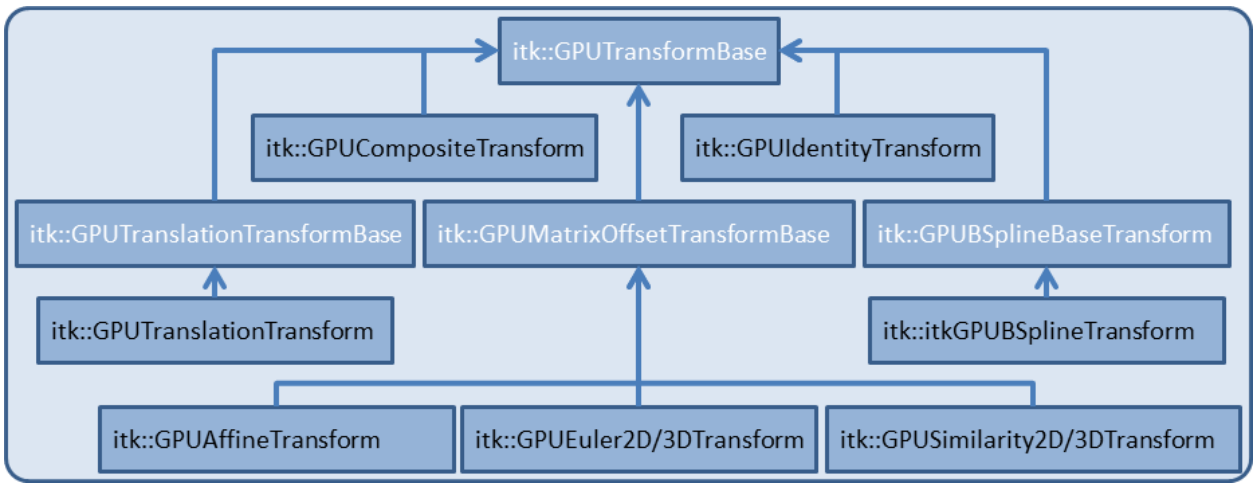


Figure 6: Design for GPU transforms.

Expanding this filter for GPU implementation, a set of GPU classes have been developed to resample images within the ITK framework.

Currently, we implemented the following interpolators

- `itk::GPUNearestNeighborInterpolateImageFunction`
- `itk::GPULinearInterpolateImageFunction`
- `itk::GPUBSplineInterpolateImageFunction`

and the following transforms

- `itk::GPUIdentityTransform`
- `itk::GPUTranslationTransform`
- `itk::GPUEuler2DTransform`
- `itk::GPUEuler3DTransform`
- `itk::GPUSimilarity2DTransform`
- `itk::GPUSimilarity3DTransform`
- `itk::GPUAffineTransform`

- itk::GPUBSplineTransform
- itk::GPUCompositeTransform

These are the most common choices. It is relatively easy to add implementations of other transform components, since base functionality is in place. Note that the identity transform is used in the Gaussian pyramids when shrinking is performed through the resampler instead of the itk::ShrinkImageFilter (an option in the resample filter). Several classes are provided, that cover base implementations for the most common usages of the transforms and interpolators:

- itk::GPUBSplineBaseTransform is a base class for the itk::GPUBSplineTransform.
- itk::GPUImageBase is the OpenCL definition of the ITK image properties such as direction, spacing, origin, size, index_to_physical_point and physical_point_to_index. This class also provides common coordinates transformation for OpenCL kernels identical to the itk::ImageBase class.
- itk::GPUImageFunction is OpenCL implementation of itk::ImageFunction.
- itk::GPUInterpolatorBase is a base class for all GPU interpolators.
- itk::GPUMatrixOffsetTransformBase is a base class for the following transforms: itk::GPUEuler2DTransform, itk::GPUEuler3DTransform, itk::GPUSimilarity2DTransform, itk::GPUSimilarity3DTransform, and itk::GPUAffineTransform.
- itk::GPUTransformBase is a base class for all GPU transforms.
- itk::GPUTranslationTransformBase is a base class for the itk::GPUTranslationTransform.
- itk::GPUInterpolateImageFunction contains some attributes related to the itk::Image coordinate system definition.

The classes itk::GPUTransformBase and itk::GPUInterpolatorBase have been created to provide a generic way of passing parameters to the itk::GPUResampleImageFilter from transforms and interpolators with the function GetParametersDataManager(). They also provide access to the OpenCL code via a virtual GetSourceCode() function, which returns code as a string object. In addition, access to the transformation matrixes performing conversion between pixel and physical coordinate systems is provided by the itk::GPUInterpolateImageFunction with the function GetParametersDataManager(). See Figure 5 and 6 for an inheritance diagram.

In addition to the above classes, some other classes required a GPU version. In case of a B-spline interpolator or transform, calculation of the B-spline coefficients of an image is required [10]. This action is performed by the itk::BSplineDecompositionImageFilter. The ITK implementation of this filter does not support multi-threading. This appeared to be a bottle-neck in the OpenCL code and therefore an OpenCL implementation of this class was made called the itk::GPUBSplineDecompositionImageFilter. In addition, itk::GPUCastImageFilter and itk::GPUShrinkImageFilter were made, as they were used by the resampler.

In the ITK CPU implementation the flexibility to use any transformation in combination with any interpolator is achieved using classes and virtual methods. This flexibility introduces a major challenge when implementing a GPU version of this filter. As mentioned earlier, OpenCL is a simplified C language specification, which does not provide any way of implementing virtuality on kernels. In order to solve this issue, we propose to split the final OpenCL kernel for the itk::GPUResampleImageFilter in three kernels and use the enqueue mechanism with synchronization in combination with an intermediate deformation field:

Initialization: ResampleImageFilterPre is an OpenCL kernel responsible for the initialization of the

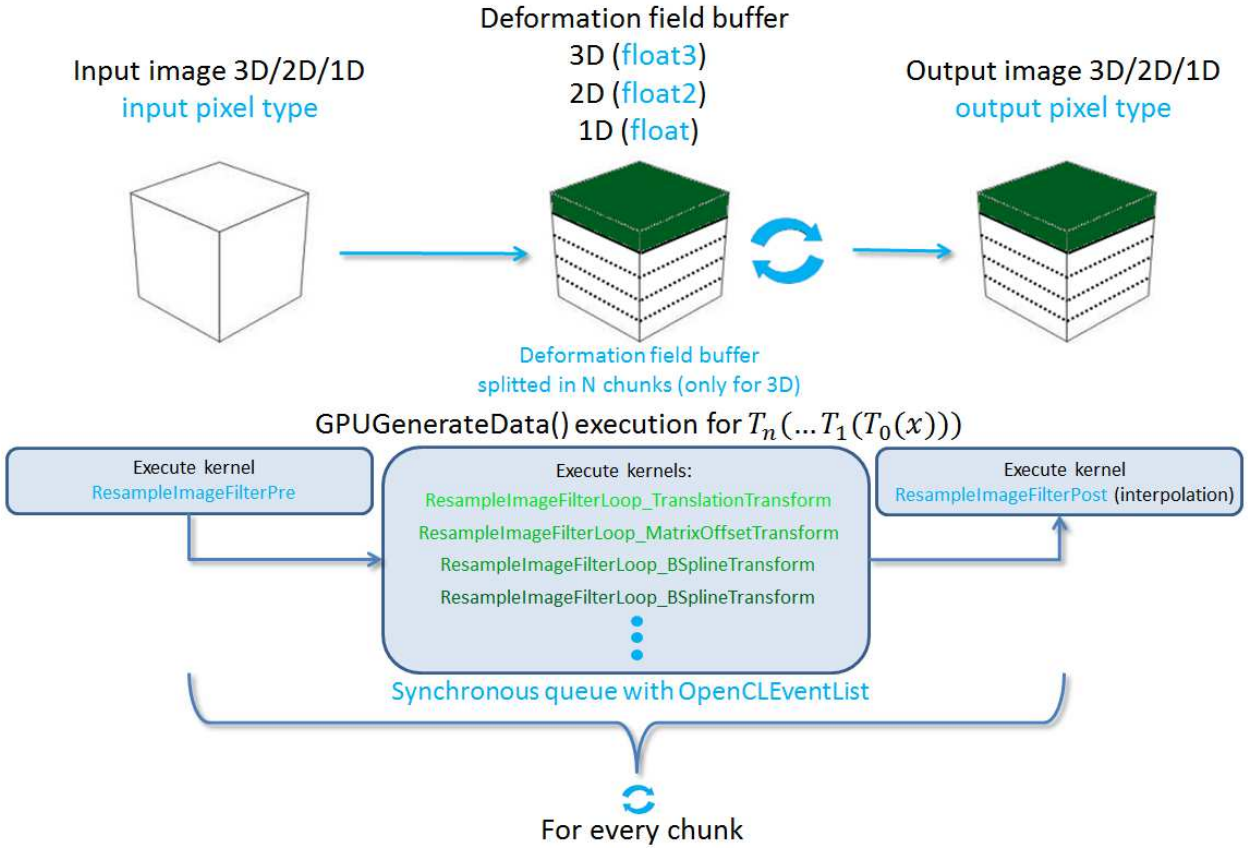


Figure 7: Design of `itk::GPUResampleImageFilter::GPUGenerateData()`. We select a chunk of the output image, and for that chunk a series of transformations $\boldsymbol{T}_n(\ldots\boldsymbol{T}_1(\boldsymbol{T}_0(\boldsymbol{x})))$ are computed and stored in the deformation field. After these transformation kernels have finished, the input image is interpolated and the result is stored in the output image chunk. The we proceed to the next chunk.

deformation field buffer.

Transformation: `ResampleImageFilterLoop` is an OpenCL kernel performing the transformation $\boldsymbol{T}$.

Interpolation: `ResampleImageFilterPost` is an OpenCL kernel performing the interpolation $I_M(\boldsymbol{T}(\boldsymbol{x}))$.

The design is illustrated in Figure 7.

The `ResampleImageFilterPre` kernel is compiled when the resample filter is constructed. The `ResampleImageFilterLoop` kernel and the `ResampleImageFilterPost` kernel are compiled when `GPUResampleImageFilter::SetTransform()` and `GPUResampleImageFilter::SetInterpolator()` are called, respectively. The code needed to compile these kernels is retrieved through the `GetSourceCode()` functionality provided by the base classes. When a B-spline transform or B-spline interpolator has been selected, we provide the additional parameters (B-spline coefficients) as images to the `ResampleImageFilterLoop` kernel. At the moment the `GPUGenerateData()` method is called, all kernels are ready for use.

As mentioned, next to the input and output images, we introduce an intermediate vector-typed image representing for each point the deformation (deformation field buffer). The input, output and intermediate vector image are all stored in global memory. The input and output image are stored completely in GPU memory.

In order to reduce memory consumption and to support larger image sizes only a part (chunk) of the deformation field is stored in memory. This chunk is reused until the full image is resampled. For performance these chunks should be as large as the GPU can fit. This chunk-procedure is only implemented for 3D, as this is not needed in 2D.

The `GPUResampleImageFilter::GPUGenerateData()` function is mainly responsible for setting parameters to the kernels, splitting the deformation field buffer in chunks and scheduling of the kernels. For the scheduling two dedicated classes `OpenCLEvent` and `OpenCLEventList` were developed, and the `GPUKernelManager` was modified to support it.

The above described implementation of the resampling framework not only supports single transformations $\boldsymbol{T}(\boldsymbol{x})$, but also any compositions of transformations $\boldsymbol{T}_n(\ldots\boldsymbol{T}_1(\boldsymbol{T}_0(\boldsymbol{x})))$. This feature is frequently used in image registration, for example when a rigid or affine registration is performed prior to a nonrigid B-spline registration. In `elastix` we always use these composed transformations, also when only a single transformation is selected by the user. An `itk::GPUCompositeTransform` was created to store all GPU transformations. The `ResampleImageFilterLoop` kernel was then modified to sequentially schedule a list of transformations, again exploiting the event lists.

A number of issues where encountered during developments and are documented here.

Kernel arguments The number of function arguments used in the kernel function appeared to be limited in NVidia OpenCL implementation (execution problem). Therefore, we minimized this number by grouping all parameters passed to the `itk::GPUResampleImageFilter` kernel in a struct and set it in constant memory. This generalizes the implementation and solved the NVidia issue[3].

Double vs float The current design of the ITK `itk::BSplineTransform` does not allow to store the coefficients parameter as a `float` type. It is hardcoded to be of type `double`, which poses problems on some GPUs. To overcome this problem we added an extra copy and cast operation from double to float to the `itk::GPUBSplineTransform`, by defining the `itk::GPUCastImageFilter`. This poses some unnecessary overhead when using B-spline interpolators and transforms.

6 Experimental results

6.1 Experimental setup

In our experiments we have used various images with different sizes together with two different computer systems. Details of the systems are given in Table 1. As can be seen, we have used NVidia's GTX 260 and 480 graphical cards, while currently (end 2012) the 690 generation is available with much more GPU acceleration power. In addition to GPU tests. The four images were of dimensions 2 and 3, with sizes: a small 2D image of 256×256 which has $\approx 10^4$ pixels, small 3D ($100 \times 100 \times 100 \approx 10^6$), medium 3D ($256 \times 256 \times 256 \approx 10^7$) and large 3D ($512 \times 512 \times 256 \approx 10^8$).

To evaluate the performance and accuracy of the OpenCL implementation, we compared the results with the original ITK CPU implementation. Two quality criteria were chosen. The speedup factor was used to measure the performance gain. For evaluation of the accuracy we used the root mean square error (RMSE)

[3] We tried to get some explanation of this problem on NVidia's forum, providing a minimal example illustrating the problem. We did however not get a reasonable explanation, leaving us with the conclusion that this is an NVidia driver issue. See http://forums.nvidia.com/index.php?showtopic=215769&st=0&p=1326194&hl=clEnqueueNDRangeKernel&fromsearch=1&#ent
Note that this issue did not occur on other OpenCL platforms.

Table 1: Test systems details.

	System 1	System 2
OS	Windows 7, 64 bit	Linux Ubuntu 10.04.3 LTS, 64 bit
CPU	Intel Core i7, 4 cores @ 2.6 GHz	Intel Xeon E5620, 8 cores @ 2.4 GHz
GPU	Nvidia Geforce GTX 260	Nvidia Geforce GTX 480
Compiler	MS VS2010	gcc 4.4.3
OpenCL version	OpenCL 1.1 CUDA 5.0, driver 306.97	NVIDIA UNIX x86_64 Kernel Module 290.10

between the results of the ITK CPU implementation (which acts as a ground truth) and that of the GPU:

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=0}^{n}\left(I_{\mathrm{CPU}}(\boldsymbol{x}_i) - I_{\mathrm{GPU}}(\boldsymbol{x}_i)\right)^2} \tag{3}$$

The test was considered to be successful if the RMSE was smaller than some threshold.

All timings were measured on a second run of the program, where the pre-compiled GPU kernel is loaded from cache. In the first run the GPU program is somewhat slower due to the run-time compilation.

6.2 Multi-resolution: Gaussian image pyramids

Several rescale and smoothing schedules were tested on the four datasets and compared with the CPU implementation. We have used four resolution levels to generate the Gaussian pyramid. A default scaling schedule was used, which downsizes the images by a factor of 8, 4, 2 and 1 for the four resolutions, respectively. Also a default smoothing schedule was used with $\sigma = 4, 2, 1$ and 0 for the four resolutions, respectively. Tables 2 and 3 shows the timing and accuracy results for the two test systems, respectively.

The accuracy as measured by the RMSE was quite small, meaning that the smoothing results were almost exactly equal. We assume that the residual error is due to numerical differences between the CPU and the GPU.

Moderate speedup factors in the range 2 - 4 were measured on both systems, for larger images. Small images are actually slower on the GPU, because of the copying overhead involved. This could be partially hidden by overlapping memory transfer with kernel execution, but this requires adaptations to the source code. Note that the CPU implementation was already quite fast. Only the large image took more than 1s to process. For that size of images, however, the GPU is beneficial.

Notice from the tables that when only smoothing is performed and no rescaling, a speedup of 3.0 - 3.7 was obtained, but when rescaling was added the performance dropped to 1.6 - 2.0. So, most acceleration comes from smoothing and not from the rescaling operation. Further investigation is needed to find the bottleneck related to the resizing operation. Also note from the tables that a further tweak is to completely skip execution when no smoothing or no resizing is needed.

Table 2: Results of the multi-resolution pyramid filter on System 1, which has 4 cores. Timings shown are for all four levels in total. Failure was due to insufficient GPU memory for the large data sets, and was marker as na.

image size	CPU (s)	GPU (s)	ratio	RMSE	passed	rescale	smooth	shrink
256x256	0.005	0.070	0.1	0.0000	Yes	Off	Off	Off
256x256	0.013	0.193	0.1	0.0003	Yes	Off	On	Off
256x256	0.003	0.060	0.1	0.0000	Yes	On	Off	Off
256x256	0.007	0.151	0.0	0.0003	Yes	On	On	Off
256x256	0.006	0.140	0.0	0.0003	Yes	Off	On	On
256x256	0.002	0.068	0.0	0.0000	Yes	On	Off	On
256x256	0.008	0.145	0.1	0.0003	Yes	On	On	On
100x100x100	0.017	0.030	0.6	0.0000	Yes	Off	Off	Off
100x100x100	0.063	0.225	0.3	0.0071	Yes	Off	On	Off
100x100x100	0.018	0.030	0.6	0.0000	Yes	On	Off	Off
100x100x100	0.077	0.245	0.3	0.0071	Yes	On	On	Off
100x100x100	0.065	0.229	0.3	0.0071	Yes	Off	On	On
100x100x100	0.012	0.025	0.5	0.0000	Yes	On	Off	On
100x100x100	0.070	0.251	0.3	0.0071	Yes	On	On	On
256x256x256	0.271	0.221	1.2	0.0000	Yes	Off	Off	Off
256x256x256	1.461	0.660	2.2	0.0076	Yes	Off	On	Off
256x256x256	0.318	0.197	1.6	0.0011	Yes	On	Off	Off
256x256x256	1.930	0.706	2.7	0.0076	Yes	On	On	Off
256x256x256	1.537	0.531	2.9	0.0076	Yes	Off	On	On
256x256x256	0.185	0.144	1.3	0.0000	Yes	On	Off	On
256x256x256	1.801	0.643	2.8	0.0076	Yes	On	On	On
512x512x256	1.046	na	na	0.0000	No	Off	Off	Off
512x512x256	7.145	na	na	0.0000	No	Off	On	Off
512x512x256	1.941	0.500	3.9	0.0057	Yes	On	Off	Off
512x512x256	9.208	na	na	0.0000	No	On	On	Off
512x512x256	7.173	na	na	0.0000	No	Off	On	On
512x512x256	0.837	0.367	2.3	0.0000	Yes	On	Off	On
512x512x256	8.342	na	na	0.0000	No	On	On	On

Table 3: Results of the multi-resolution pyramid filter on System 2, which has 8 cores. Timings shown are for all four levels in total. Due to insufficient GPU memory two test failed, but they were rerun with the memory conservation flag. These tests are marked with Yes*.

image size	CPU (s)	GPU (s)	ratio	RMSE	passed	rescale	smooth	shrink
256x256	0.002	0.002	0.7	0.0000	Yes	Off	Off	Off
256x256	0.005	0.007	0.7	0.0000	Yes	Off	On	Off
256x256	0.002	0.004	0.6	0.0000	Yes	On	Off	Off
256x256	0.005	0.011	0.5	0.0000	Yes	On	On	Off
256x256	0.006	0.007	0.9	0.0000	Yes	Off	On	On
256x256	0.002	0.002	1.1	0.0000	Yes	On	Off	On
256x256	0.005	0.008	0.7	0.0000	Yes	On	On	On
100x100x100	0.009	0.024	0.4	0.0000	Yes	Off	Off	Off
100x100x100	0.043	0.022	2.0	0.0007	Yes	Off	On	Off
100x100x100	0.014	0.017	0.8	0.0000	Yes	On	Off	Off
100x100x100	0.056	0.035	1.6	0.0007	Yes	On	On	Off
100x100x100	0.043	0.021	2.0	0.0007	Yes	Off	On	On
100x100x100	0.006	0.010	0.6	0.0000	Yes	On	Off	On
100x100x100	0.045	0.028	1.6	0.0007	Yes	On	On	On
256x256x256	0.086	0.226	0.4	0.0000	Yes	Off	Off	Off
256x256x256	0.779	0.211	3.7	0.0010	Yes	Off	On	Off
256x256x256	0.155	0.122	1.3	0.0010	Yes	On	Off	Off
256x256x256	0.915	0.370	2.5	0.0010	Yes	On	On	Off
256x256x256	0.792	0.210	3.8	0.0010	Yes	Off	On	On
256x256x256	0.080	0.085	0.9	0.0000	Yes	On	Off	On
256x256x256	0.847	0.329	2.6	0.0010	Yes	On	On	On
512x512x256	0.308	0.783	0.4	0.0000	Yes	Off	Off	Off
512x512x256	3.611	0.384	9.4	0.0000	Yes*	Off	On	Off
512x512x256	0.759	0.432	1.8	0.0065	Yes	On	Off	Off
512x512x256	4.264	1.404	3.0	0.0018	Yes	On	On	Off
512x512x256	3.532	0.383	9.2	0.0000	Yes*	Off	On	On
512x512x256	0.279	0.305	0.9	0.0000	Yes	On	Off	On
512x512x256	3.756	1.252	3.0	0.0004	Yes	On	On	On

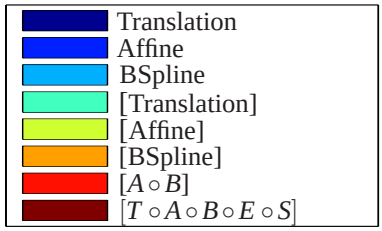
Table 4: Results of the resampling filter on System 1, which has 4 cores, the composite transform was not used. Timings are shown in seconds.

image size	CPU (s)	GPU (s)	ratio	RMSE	passed	interpolator	transform
100x100x100	0.015	0.022	0.7	40.2	Yes	Nearest	Translation
256x256x256	0.192	0.085	2.3	28.7	Yes	Nearest	Translation
512x512x256	0.488	0.271	1.8	0.0	Yes	Nearest	Translation
100x100x100	0.019	0.023	0.8	0.1	Yes	Linear	Translation
256x256x256	0.292	0.085	3.4	0.1	Yes	Linear	Translation
512x512x256	1.277	0.259	4.9	0.2	Yes	Linear	Translation
100x100x100	0.293	0.133	2.2	0.1	Yes	BSpline	Translation
256x256x256	5.279	1.075	4.9	0.2	Yes	BSpline	Translation
512x512x256	23.164	5.223	4.4	0.2	Yes	BSpline	Translation
100x100x100	0.008	0.017	0.5	1.1	Yes	Nearest	Affine
256x256x256	0.131	0.077	1.7	2.5	Yes	Nearest	Affine
512x512x256	0.519	0.247	2.1	6.3	Yes	Nearest	Affine
100x100x100	0.029	0.019	1.5	0.1	Yes	Linear	Affine
256x256x256	0.342	0.086	4.0	0.2	Yes	Linear	Affine
512x512x256	1.293	0.263	4.9	0.6	Yes	Linear	Affine
100x100x100	0.285	0.134	2.1	0.1	Yes	BSpline	Affine
256x256x256	5.216	1.098	4.8	0.2	Yes	BSpline	Affine
512x512x256	24.147	5.130	4.7	0.7	Yes	BSpline	Affine
100x100x100	0.968	0.053	18.3	0.2	Yes	Nearest	BSpline
256x256x256	11.015	0.475	23.2	0.6	Yes	Nearest	BSpline
512x512x256	43.053	1.643	26.2	0.5	Yes	Nearest	BSpline
100x100x100	0.677	0.053	12.7	0.0	Yes	Linear	BSpline
256x256x256	10.980	0.490	22.4	0.0	Yes	Linear	BSpline
512x512x256	43.276	1.624	26.6	0.0	Yes	Linear	BSpline
100x100x100	0.927	0.164	5.6	0.1	Yes	BSpline	BSpline
256x256x256	16.017	1.473	10.9	0.2	Yes	BSpline	BSpline
512x512x256	65.470	6.278	10.4	0.0	Yes	BSpline	BSpline

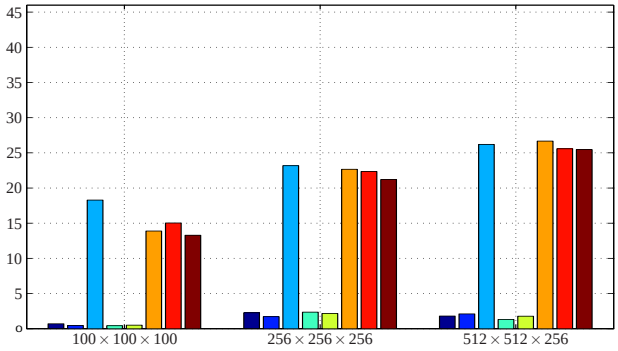
6.3 Image resampling

We tested the GPU resampling filter with different combinations of interpolators and transformations. For the B-spline interpolator and B-spline transform we have used third order splines. Detailed results are shown in Tables 4 and 5 for system 1 and 6, and 7 for system 2. Tables 4 and 6 show the results for a single transformation not using the `itk::GPUCompositeTransform`, and Tables 5 and 7 show the results for the composite transformations. Finally, in Figure 8 we show the speedup more graphically. In the figure and in the tables sometimes the following abbreviations were used: Affine = *A*, Translation = *T*, B-spline = *B*, Euler3D = *E* and Similarity3D = *S*.

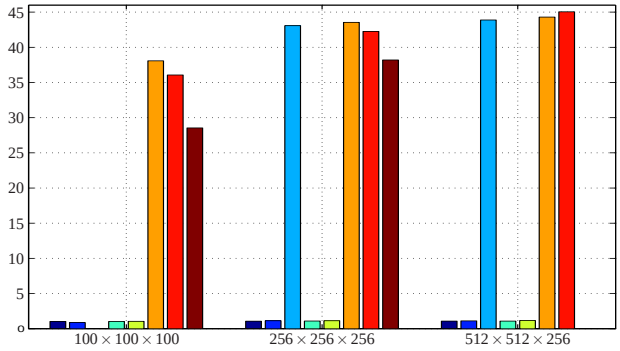
The results for resampling were very close in terms of RMSE to the output produced by ITK. The differences were due to floating point differences, but they were acceptable. The GPU implementation is particularly beneficial on the larger images, where the algorithm performs very slow even on modern systems. On smaller images the performance gain was less dramatic, or sometimes even smaller than 1. For large images, using a B-spline interpolator and transform the execution time was up to 1.5 minute on the CPU, while with a two generation old graphic card this was reduced to about 2 s. Speedups of 10 - 46 were achieved in those cases.



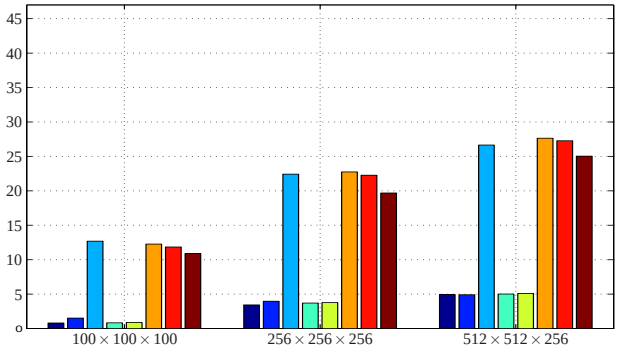
(a) Legend



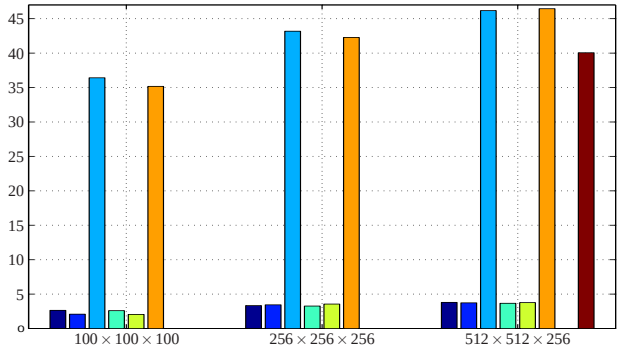
(b) System 1, nearest neighbor interpolator



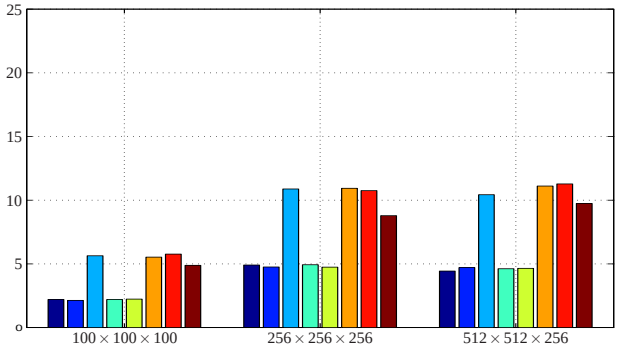
(c) System 2, nearest neighbor interpolator



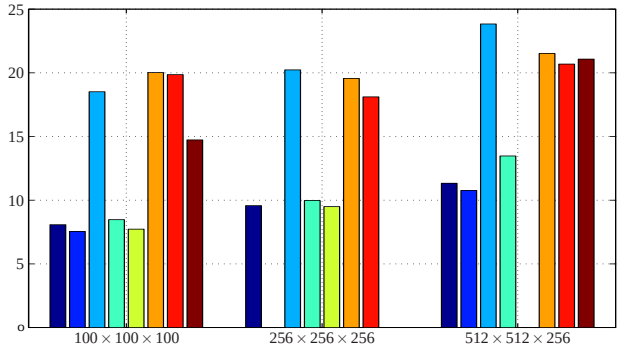
(d) System 1, linear interpolator



(e) System 2, linear interpolator



(f) System 1, B-spline interpolator



(g) System 2, B-spline interpolator

Figure 8: Speedup factors. Left and right column show results for system 1 and 2, respectively. First, second and third row show results for the nearest neighbor, linear and B-spline interpolator, respectively. Square brackets indicates that the composite transform `itk::GPUCompositeTransform` is used.

Table 5: Results of the resampling filter on System 1, which has 4 cores, the composite transform was used. Timings are shown in seconds.

image size	CPU (s)	GPU (s)	ratio	RMSE	passed	interpolator	transform
100x100x100	0.01	0.02	0.4	40.2	Yes	Nearest	[Translation]
256x256x256	0.19	0.08	2.4	28.7	Yes	Nearest	[Translation]
512x512x256	0.49	0.37	1.3	0.0	Yes	Nearest	[Translation]
100x100x100	0.02	0.02	0.8	0.1	Yes	Linear	[Translation]
256x256x256	0.31	0.08	3.7	0.1	Yes	Linear	[Translation]
512x512x256	1.27	0.25	5.0	0.2	Yes	Linear	[Translation]
100x100x100	0.29	0.13	2.2	0.1	Yes	BSpline	[Translation]
256x256x256	5.25	1.06	4.9	0.2	Yes	BSpline	[Translation]
512x512x256	23.20	5.03	4.6	0.2	Yes	BSpline	[Translation]
100x100x100	0.01	0.02	0.5	1.1	Yes	Nearest	[Affine]
256x256x256	0.19	0.09	2.2	2.5	Yes	Nearest	[Affine]
512x512x256	0.51	0.29	1.8	6.3	Yes	Nearest	[Affine]
100x100x100	0.02	0.02	0.9	0.1	Yes	Linear	[Affine]
256x256x256	0.32	0.08	3.8	0.2	Yes	Linear	[Affine]
512x512x256	1.32	0.26	5.1	0.6	Yes	Linear	[Affine]
100x100x100	0.29	0.13	2.2	0.1	Yes	BSpline	[Affine]
256x256x256	5.20	1.10	4.7	0.2	Yes	BSpline	[Affine]
512x512x256	22.87	4.92	4.6	0.7	Yes	BSpline	[Affine]
100x100x100	0.75	0.05	13.9	0.2	Yes	Nearest	[BSpline]
256x256x256	10.88	0.48	22.7	0.6	Yes	Nearest	[BSpline]
512x512x256	44.04	1.65	26.7	0.5	Yes	Nearest	[BSpline]
100x100x100	0.68	0.06	12.3	0.0	Yes	Linear	[BSpline]
256x256x256	11.16	0.49	22.7	0.0	Yes	Linear	[BSpline]
512x512x256	44.93	1.63	27.6	0.0	Yes	Linear	[BSpline]
100x100x100	0.95	0.17	5.5	0.1	Yes	BSpline	[BSpline]
256x256x256	16.20	1.48	10.9	0.2	Yes	BSpline	[BSpline]
512x512x256	66.69	6.00	11.1	0.0	Yes	BSpline	[BSpline]
100x100x100	0.84	0.06	15.0	0.1	Yes	Nearest	$[A \circ B]$
256x256x256	11.10	0.50	22.3	0.3	Yes	Nearest	$[A \circ B]$
512x512x256	44.17	1.73	25.6	0.5	Yes	Nearest	$[A \circ B]$
100x100x100	0.68	0.06	11.8	0.0	Yes	Linear	$[A \circ B]$
256x256x256	11.24	0.50	22.3	0.0	Yes	Linear	$[A \circ B]$
512x512x256	45.52	1.67	27.3	0.0	Yes	Linear	$[A \circ B]$
100x100x100	0.97	0.17	5.8	0.1	Yes	BSpline	$[A \circ B]$
256x256x256	16.23	1.51	10.8	0.2	Yes	BSpline	$[A \circ B]$
512x512x256	70.59	6.26	11.3	0.0	Yes	BSpline	$[A \circ B]$
100x100x100	0.77	0.06	13.3	0.1	Yes	Nearest	$[T \circ A \circ B \circ E \circ S]$
256x256x256	10.03	0.47	21.2	0.3	Yes	Nearest	$[T \circ A \circ B \circ E \circ S]$
512x512x256	42.34	1.66	25.5	0.5	Yes	Nearest	$[T \circ A \circ B \circ E \circ S]$
100x100x100	0.62	0.06	10.9	0.0	Yes	Linear	$[T \circ A \circ B \circ E \circ S]$
256x256x256	10.35	0.53	19.7	0.0	Yes	Linear	$[T \circ A \circ B \circ E \circ S]$
512x512x256	43.34	1.73	25.0	0.0	Yes	Linear	$[T \circ A \circ B \circ E \circ S]$
100x100x100	0.87	0.18	4.9	0.1	Yes	BSpline	$[T \circ A \circ B \circ E \circ S]$
256x256x256	14.97	1.70	8.8	0.1	Yes	BSpline	$[T \circ A \circ B \circ E \circ S]$
512x512x256	64.94	6.66	9.7	0.0	Yes	BSpline	$[T \circ A \circ B \circ E \circ S]$

Table 6: Results of the resampling filter on System 2, which has 8 cores, the composite transform was not used. Timings are shown in seconds.

image size	CPU (s)	GPU (s)	ratio	RMSE	passed	interpolator	transform
100x100x100	0.01	0.00	1.0	40.2	Yes	Nearest	Translation
256x256x256	0.05	0.05	1.1	28.7	Yes	Nearest	Translation
512x512x256	0.19	0.18	1.1	0.0	Yes	Nearest	Translation
100x100x100	0.01	0.01	2.6	0.1	Yes	Linear	Translation
256x256x256	0.16	0.05	3.3	0.1	Yes	Linear	Translation
512x512x256	0.69	0.18	3.8	0.2	Yes	Linear	Translation
100x100x100	0.22	0.03	8.1	0.1	Yes	BSpline	Translation
256x256x256	4.09	0.43	9.6	0.2	Yes	BSpline	Translation
512x512x256	16.77	1.48	11.3	0.2	Yes	BSpline	Translation
100x100x100	0.00	0.00	0.9	1.1	Yes	Nearest	Affine
256x256x256	0.05	0.05	1.1	2.3	Yes	Nearest	Affine
512x512x256	0.20	0.18	1.1	6.5	Yes	Nearest	Affine
100x100x100	0.01	0.01	2.1	0.1	Yes	Linear	Affine
256x256x256	0.17	0.05	3.4	0.1	Yes	Linear	Affine
512x512x256	0.68	0.18	3.7	0.6	Yes	Linear	Affine
100x100x100	0.22	0.03	7.5	0.1	Yes	BSpline	Affine
256x256x256	3.91	na	na	0.0	No	BSpline	Affine
512x512x256	16.92	1.57	10.8	0.7	Yes	BSpline	Affine
100x100x100	0.37	na	na	0.0	No	Nearest	BSpline
256x256x256	5.74	0.13	43.1	0.6	Yes	Nearest	BSpline
512x512x256	21.23	0.48	43.9	0.5	Yes	Nearest	BSpline
100x100x100	0.38	0.01	36.4	0.0	Yes	Linear	BSpline
256x256x256	5.95	0.14	43.2	0.0	Yes	Linear	BSpline
512x512x256	22.64	0.49	46.2	0.0	Yes	Linear	BSpline
100x100x100	0.59	0.03	18.5	0.1	Yes	BSpline	BSpline
256x256x256	10.44	0.52	20.2	0.2	Yes	BSpline	BSpline
512x512x256	42.61	1.79	23.8	0.0	Yes	BSpline	BSpline

Another interesting point is that the linear interpolator showed a larger performance gain than the B-spline interpolator. This is surprising, since commonly more complex operations give more speedup. It may be due to the fact that the B-spline GPU implementation required additional casting and memory transfer operations, or that the GPU code is suboptimal compared to the CPU code, or the more heavily (random) memory access required by the B-spline get penalized more by a GPU.

Profiling the resampler

The execution time was sub-divided into its parts using profiling. The more detailed results are given in Table 8. The B-spline interpolator and transform work with an underlying B-spline coefficient image. This image is hard-coded on the CPU to be of type `double` and requires converting to `float` and copying to the GPU, taking almost 20% of the time. Execution of the kernel consumed 75% of the time. This means that when a form of memory transfer hiding is employed the run time can be reduces to about 1.5s, which would result in a speedup of 30 instead of 23, only a minor gain in this case.

Table 7: Results of the resampling filter on System 2, which has 8 cores, the composite transform was used. Timings are shown in seconds. Failure was due to exception and was marker as na.

image size	CPU (s)	GPU (s)	ratio	RMSE	passed	interpolator	transform
100x100x100	0.01	0.00	1.0	40.2	Yes	Nearest	[Translation]
256x256x256	0.05	0.05	1.1	28.7	Yes	Nearest	[Translation]
512x512x256	0.19	0.18	1.1	0.0	Yes	Nearest	[Translation]
100x100x100	0.01	0.01	2.6	0.1	Yes	Linear	[Translation]
256x256x256	0.16	0.05	3.3	0.1	Yes	Linear	[Translation]
512x512x256	0.68	0.19	3.7	0.2	Yes	Linear	[Translation]
100x100x100	0.23	0.03	8.5	0.1	Yes	BSpline	[Translation]
256x256x256	4.25	0.43	10.0	0.2	Yes	BSpline	[Translation]
512x512x256	19.79	1.47	13.5	0.2	Yes	BSpline	[Translation]
100x100x100	0.01	0.01	1.1	1.1	Yes	Nearest	[Affine]
256x256x256	0.05	0.05	1.1	2.3	Yes	Nearest	[Affine]
512x512x256	0.20	0.18	1.2	6.5	Yes	Nearest	[Affine]
100x100x100	0.01	0.01	2.0	0.1	Yes	Linear	[Affine]
256x256x256	0.17	0.05	3.5	0.1	Yes	Linear	[Affine]
512x512x256	0.69	0.18	3.8	0.6	Yes	Linear	[Affine]
100x100x100	0.22	0.03	7.7	0.1	Yes	BSpline	[Affine]
256x256x256	4.23	0.45	9.5	0.2	Yes	BSpline	[Affine]
512x512x256	16.98	na	na	0.0	No	BSpline	[Affine]
100x100x100	0.39	0.01	38.1	0.4	Yes	Nearest	[BSpline]
256x256x256	5.92	0.14	43.5	0.6	Yes	Nearest	[BSpline]
512x512x256	21.39	0.48	44.3	0.5	Yes	Nearest	[BSpline]
100x100x100	0.38	0.01	35.2	0.0	Yes	Linear	[BSpline]
256x256x256	5.86	0.14	42.3	0.0	Yes	Linear	[BSpline]
512x512x256	22.91	0.49	46.5	0.0	Yes	Linear	[BSpline]
100x100x100	0.64	0.03	20.0	0.1	Yes	BSpline	[BSpline]
256x256x256	10.06	0.51	19.6	0.2	Yes	BSpline	[BSpline]
512x512x256	38.42	1.79	21.5	0.0	Yes	BSpline	[BSpline]
100x100x100	0.38	0.01	36.1	0.1	Yes	Nearest	$[A \circ B]$
256x256x256	5.84	0.14	42.3	0.0	No	Nearest	$[A \circ B]$
512x512x256	22.39	0.50	45.0	0.5	Yes	Nearest	$[A \circ B]$
100x100x100	0.38	na	na	0.0	No	Linear	$[A \circ B]$
256x256x256	6.62	na	na	0.0	No	Linear	$[A \circ B]$
512x512x256	22.17	na	na	0.0	No	Linear	$[A \circ B]$
100x100x100	0.68	0.03	19.9	0.1	Yes	BSpline	$[A \circ B]$
256x256x256	9.77	0.54	18.1	0.0	No	BSpline	$[A \circ B]$
512x512x256	39.06	1.89	20.7	0.0	Yes	BSpline	$[A \circ B]$
100x100x100	0.33	0.01	28.5	0.1	Yes	Nearest	$[T \circ A \circ B \circ E \circ S]$
256x256x256	5.11	0.13	38.2	0.3	Yes	Nearest	$[T \circ A \circ B \circ E \circ S]$
512x512x256	20.77	na	na	0.0	No	Nearest	$[T \circ A \circ B \circ E \circ S]$
100x100x100	0.33	na	na	0.0	No	Linear	$[T \circ A \circ B \circ E \circ S]$
256x256x256	5.08	na	na	0.0	No	Linear	$[T \circ A \circ B \circ E \circ S]$
512x512x256	21.35	0.53	40.0	0.0	Yes	Linear	$[T \circ A \circ B \circ E \circ S]$
100x100x100	0.53	0.04	14.7	0.1	Yes	BSpline	$[T \circ A \circ B \circ E \circ S]$
256x256x256	8.79	na	na	0.0	No	BSpline	$[T \circ A \circ B \circ E \circ S]$
512x512x256	37.94	1.80	21.1	0.0	Yes	BSpline	$[T \circ A \circ B \circ E \circ S]$

Table 8: Detailed execution of the `itk::GPUResampleImageFilter` on the 512x512x256 image using a B-spline transform and a B-spline interpolator (both third order), on system 2. The B-spline related entries consist of double-float casting and transfer to the GPU.

Operation	**Time (s)**	**%**
B-spline interpolator	0.086	4.8
B-spline transform	0.243	13.6
CPU → GPU copying	0.027	1.5
Executing OpenCL kernel	1.341	75.0
GPU → CPU copying	0.084	4.7
Other operations	0.009	0.5
Total time	**1.788**	**100**

Comparison to previous results

We compared the resampling performance gain with our previous result [5], where the resample algorithm was implemented using CUDA. There we reported a speedup of 10 - 65, which is a bit higher that we currently achieved with OpenCL. This is mostly contributed to a special CUDA implementation [9] used for the B-spline computation, where the 3^{rd} order splines are decomposed in a number of 1^{st} order splines, i.e. linear interpolation. Linear interpolation is hardwired on the GPU, leading to a considerable performance gain. This was not implemented in OpenCL. Further comparing the two implementations, the CUDA resampler only supports 3D images, using 3rd order B-spline interpolation and transformation, while the OpenCL code is much more generic, supporting several dimensions, interpolators and transformations. In addition, OpenCL runs on all kinds of GPUs, while CUDA only supports NVidia ones. See detailed speed up graphs 8.

7 Conclusion and Discussion

7.1 Conclusions

We developed a generic OpenCL framework to create pyramids and resamplers on the GPU, exploiting the ITK GPU acceleration design. The generic architecture and close integration with ITK will easy adoption by the medical image processing community. The decision to use OpenCL allows targeting most of the graphical devices available today. The developed code is generic and allows extension to other transformations usage during image registration.

We obtained speedup factors of 2 - 4 for the image pyramids and 10 - 46 for the resampling, on larger images, using a Geforce GTX 480 (two new generations of GPUs have since shipped). This shows that the GPU is an efficient computing device for these tasks. The specific kernels were implemented in a straightforward manner, leaving room for performance improvement in future work. Details of the possibilities are given below.

In conclusion, two time consuming parts of the registration algorithm were accelerated using GPU programming. The knowledge obtained in this project and the resulting OpenCL coding designs, will be of direct benefit in future work, where the core of the registration algorithm can also be accelerated using the GPU.

7.2 Limitations and future work

There are several areas in which our work can be improved and extended. We subdivided them in several categories: transfer the work to the community, performance improvements, coding improvements, and miscellaneous. We detail them in the list below.

Community adoption

1. `elastix` integration. The GPU extensions are currently only initially integrated in `elastix`. The source code is there, and many GPU tests are in place. Outside the tests it is not yet used. Code needs to be added that supports good user experience. The latter includes robust detection OpenCL existence on the system, whether a suitable GPU is installed, and registering the GPU factories to `elastix`. In addition, one could think of some helper functionality that predicts if either the CPU or the GPU will give the best performance, and subsequently selects the fastest option.
2. Synchronization with the ITK. During development we significantly extended the ITK GPU design and fixed some bugs. These changes need to be transferred back to the ITK repository. This process will also generate feedback, thereby improving the code see 4.4.

Optimizations

1. Find and investigate the current bottlenecks with GPU profilers.
2. Using specialized hardware functions, such as the `mad` functions that gives hardware access to the operation '$a \times b + c$'. Related to the CMake flag `OPENCL\_OPTIMIZATION\_MAD\_ENABLE`
3. Investigate the usage of the different types of memory. We took a straightforward implementation approach, but these memory-related choices are known to have a large effect on performance.
4. Experiment with different setups of the warps and the local size, etc.
5. Hide the memory transfer overhead. In the current design, we copy the data from CPU to GPU, perform the processing on the GPU, and copy the result back. In an a-synchronous implementation, the data copying can be partially hidden by already starting kernel execution on the data that has already been copied, and proceed as data enters the GPU.
6. Circumvent B-spline coefficient image overhead by enabling a `float` typed image on the host.
7. Transfer the special B-spline interpolation CUDA code [9] to OpenCL.
8. In OpenCL, compilation is frequently performed at run-time. This however introduces overhead of the compilation at run-time, especially for more complicated and extended pieces of code. It is possible however to compile before run-time, which is to be investigated.

Coding improvements

1. Split ResampleImageFilterPost kernel into two kernels: a general one and one specific for the B-spline interpolator.
2. Add extrapolation support, see `itk::ResampleImageFilter::SetExtrapolator()`.
3. Improve ITK's CPU-GPU synchronization. It is not always needed.
4. The new classes `itk::OpenCLEvent` and `itk::OpenCLEventList` have to be more deeply integrated in the logic of the GPU processing pipeline to achieve better control over the order of the kernel execution.

5. In the current implementation, filters use the `itk::GPUKernelManager` to start kernels by calling `LaunchKernel()`. In OpenCL launching kernels does not necessary mean that they will be executed immediately. Instead, the command is queued for execution. Therefore, the functions `OpenCLEvent::WaitForFinished()` or `OpenCLEventList::WaitForFinished()` have to be used more consistently. This has to be properly resolved to achieve correct executions of ITK OpenCL pipelines, which usually implicitly assume synchronous execution.
6. After kernel pipeline execution, the GPU memory, OpenCL kernels, the device and other resources have to be properly released or stopped, which is currently not done. The functions `free(clDevices)`, `clReleaseEvent()`, `clReleaseKernel()`, `clReleaseProgram()`, `clReleaseCommandQueue()`, `clReleaseContext()`, `clReleaseMemObject()`, etc, can be used for that.

Miscellaneous

1. Add AMD graphical card experimental results.
2. We experienced some stability problems when running the code using a Linux NVidia OpenCL driver. These are marked as na in the experimental results. We are not sure about its cause. It could be that our code has some issues that were only revealed on this specific platform, or that current Linux NVidia OpenCL drivers are not correctly implemented. Future investigations are needed.

7.3 Acknowledgments

This work was funded by the Netherlands Organisation for Scientific Research (NWO NRG-2010.02 and NWO 639.021.124). This work benefitted greatly from the use of the Insight Segmentation and Registration Toolkit (ITK), especially the new OpenCL GPU design.

A Find OpenCL extensions

The following functionality has been added.

- Support for Intel/AMD/NVidia OpenCL platforms, and the ability to switch between these platforms. In case of an AMD or Intel platform, the CMake variable `OPENCL_USE_PLATFORM_AMD_GPU_CPU` or `OPENCL_USE_PLATFORM_INTEL_GPU_CPU` controls the CPU or GPU device selection.
- OpenCL Math Intrinsics options. These options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness.
  - `OPENCL_MATH_SINGLE_PRECISION_CONSTANT` Treat double precision floating-point constant as single precision constant.
  - `OPENCL_MATH_DENORMS_ARE_ZERO` This option controls how single precision and double precision de-normalized numbers are handled.
- OpenCL Optimization options. These options control various sorts of optimizations. Turning on optimization makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

  - `OPENCL_OPTIMIZATION_OPT_DISABLE` This option disables all optimizations. The default is optimizations are enabled.
  - `OPENCL_OPTIMIZATION_STRICT_ALIASING` This option allows the compiler to assume the strictest aliasing rules.
  - `OPENCL_OPTIMIZATION_MAD_ENABLE` Allow a * b + c to be replaced by a mad. The mad computes a * b + c with reduced accuracy.
  - `OPENCL_OPTIMIZATION_NO_SIGNED_ZEROS` Allow optimizations for floating-point arithmetic that ignore the signedness of zero.
  - `OPENCL_OPTIMIZATION_UNSAFE_MATH_OPTIMIZATIONS` Allow optimizations for floating-point arithmetic.
  - `OPENCL_OPTIMIZATION_FINITE_MATH_ONLY` Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +-infinity.
  - `OPENCL_OPTIMIZATION_FAST_RELAXED_MATH` Sets the optimization options -cl-finite-math-only and -cl-unsafe-math-optimizations.

- OpenCL profiling `OPENCL_PROFILING` with `CL_QUEUE_PROFILING_ENABLE`. With this option event objects can be used to capture profiling information that measure execution time of a command.
- OpenCL options to request or suppress warnings.
  - `OPENCL_WARNINGS_DISABLE` This option inhibit all warning messages.
  - `OPENCL_WARNINGS_AS_ERRORS` This option make all warnings into errors.
- OpenCL options controlling the OpenCL C version.
  - `OPENCL_C_VERSION_1_1` This option determine the OpenCL C language version to use. Support all OpenCL C programs that use the OpenCL C language 1.1 specification.
  - `OPENCL_C_VERSION_1_2` This option determine the OpenCL C language version to use. Support all OpenCL C programs that use the OpenCL C language 1.2 specification.

References

[1] Intel OpenCL SDK. http://software.intel.com/en-us/vcsource/tools/opencl-sdk. 4.4

[2] ITK4 GPU acceleration. http://www.vtk.org/Wiki/ITK/Release_4/GPU_Acceleration. 3.2

[3] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, and Alexander Neundorf. CMake. http://www.cmake.org, 1999. 3.2, 4.4

[4] R. Deriche. Fast algorithms for low-level vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(1):78–87, January 1990. 5.2

[5] T. Farago, H. Nikolov, S. Klein, J.H.C. Reiber, and Marius Staring. Semi-automatic parallelisation for iterative image registration with b-splines. In Leiguang Gong et al., editor, *Medical Image Computing and Computer-Assisted Intervention*, Lecture Notes in Computer Science, pages –, Beijing, China, 2010. 6.3

[6] L. Ibánez, W. Schroeder, L. Ng, and J. Cates. The ITK software guide. 2005. 3.2, 3.3

[7] S. Klein, M. Staring, K. Murphy, M.A. Viergever, and J.P.W. Pluim. `elastix`: a toolbox for intensity-based medical image registration. *IEEE Transactions on Medical Imaging*, 29(1):196 – 205, 2010. 3.2

[8] Jim Miller, Won-Ki Jeong, and Baohua Wu. ITK4 GPU-Alpha branch. https://github.com/graphor/ITK/tree/GPU-Alpha, 2011. 4.4, 4.4

[9] Daniel Ruijters, Bart M. ter Haar Romeny, and Paul Suetens. Efficient GPU-based texture interpolation using uniform B-splines. *Journal of Graphics Tools*, 13(4):61 – 69, 2008. 6.3, 7

[10] M. Unser, A. Aldroubi, and M. Eden. B-Spline signal processing: Part II—Efficient design and applications. *IEEE Transactions on Signal Processing*, 41(2):834 – 848, 1993. 5.3