A Tutorial on Combining Nonlinear Optimization with CUDA

Release 1.00

Charles R Hatt[1]

April 27, 2015

[1]University of Wisconsin - Madison, hatt@wisc.edu

Abstract

Nonlinear optimization is a key component of many image registration algorithms. Improving registration speed is almost always desirable. One way to do this is to accelerate the optimization cost function using a parallel implementation. The purpose of this document is to provide a tutorial on how to combine the CUDA GPU computing framework with standard nonlinear optimization libraries (VNL) using CMake. The provided code can be used as a starting template for programmers looking for a relatively painless introduction to CUDA-accelerated medical image registration and other nonlinear optimization problems.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/1338]

Contents

Medical image registration is the process of aligning images so that there is a maximal spatial correlation between corresponding anatomical features. Typically, one image (the "moving" image) undergoes a spatial transformation, while the other image remains static (the "fixed" image). The standard process beings with estimating an initial spatial transformation, defined by parameters ϕ. Next, registration proceeds as follows:

1. Transform the moving image.
2. Compute a cost function ($F_C(\phi)$), aka similarity function) that estimates how well the fixed and moving images are aligned.
3. Moving in the direction of decreasing cost, re-estimate ϕ.
4. Repeat until the cost function is minimized.

This process of finding the parameters ϕ that minimize (maximize) the cost (similarity) function is termed "Nonlinear optimization." Nonlinear optimization methods have been reliably implemented and optimized in open-source software packages such as ITK [2]. For optimization, ITK uses the Vision Numerics Libraries (VNL), which wrap Fortran implementations of many of the most popular routines (Nelder-Mead, Powell, Levenburg-Marquardt, LBFGS, et cetera).

Because nonlinear optimization an iterative process, minimizing the computational burden of each iteration can decrease the overall run-time. One of the best methods for speeding-up nonlinear optimization is to accelerate the computation of the cost function, as in many cases this is the most computationally expensive part of the procedure. Recently, general purpose computing on graphics processing units (GPGPU) has been utilized by medical imaging researchers as a way to accelerate algorithms through parallel implementation [1, 3]. CUDA, designed and distributed by NVIDIA for NVIDIA GPUs, is perhaps the most popular GPGPU programming model at the time of this publication.

Huge speed-ups in the cost function computation can often be achieved using CUDA. However, getting the open-source optimization libraries to work with the CUDA Toolkit can be difficult, especially for researchers who are used to writing simple programs that don't require extensive software engineering expertise.

The purpose of this document is to provide a simple example of how to use open-source optimization libraries (VNL) with a custom CUDA-implemented cost function. It is expected that the user already has some basic experience with C++ and CUDA. However, expert knowledge of either is not required. A very simple example is provided, where we try to find the location of a minimum in an image given a sub-optimal initial guess. A class, called `demoInterface`, is implemented that handles all of the memory allocation and host/device transfers seamlessly. Example usage of CUDA `constant` and `texture` memory is provided.

1 The `demo` program

The `demo` program finds the minimal pixel coordinate in an image using nonlinear optimization. The cost function, therefore, is $F_C(\phi) = I(x,y)$, and the parameters ϕ are the x and y coordinates of a point on the image. The image, shown in Fig. 1, is meant to appear like a fairly well-behaved 2D cost function so that the user can easily visualize how the optimization works. An initial guess about the location of the minimal value is provided, an example of which is shown by the red dot in Fig 1. From the initial guess, the optimizer attempts to move in the direction of a minimum by examining neighboring pixels. There are no minima aside from the global minimum at pixel (256,256), so the algorithm should always converge as long as function evaluations are made within the boundaries of the image.

Two optimization methods are available: Powell and Nelder-Mead (aka Amoeba). However, other methods can easily be added by consulting the VNL documentation, as the syntax for other methods is usually very similiar to that used for the `vnl_powell` and `vnl_amoeba` methods.

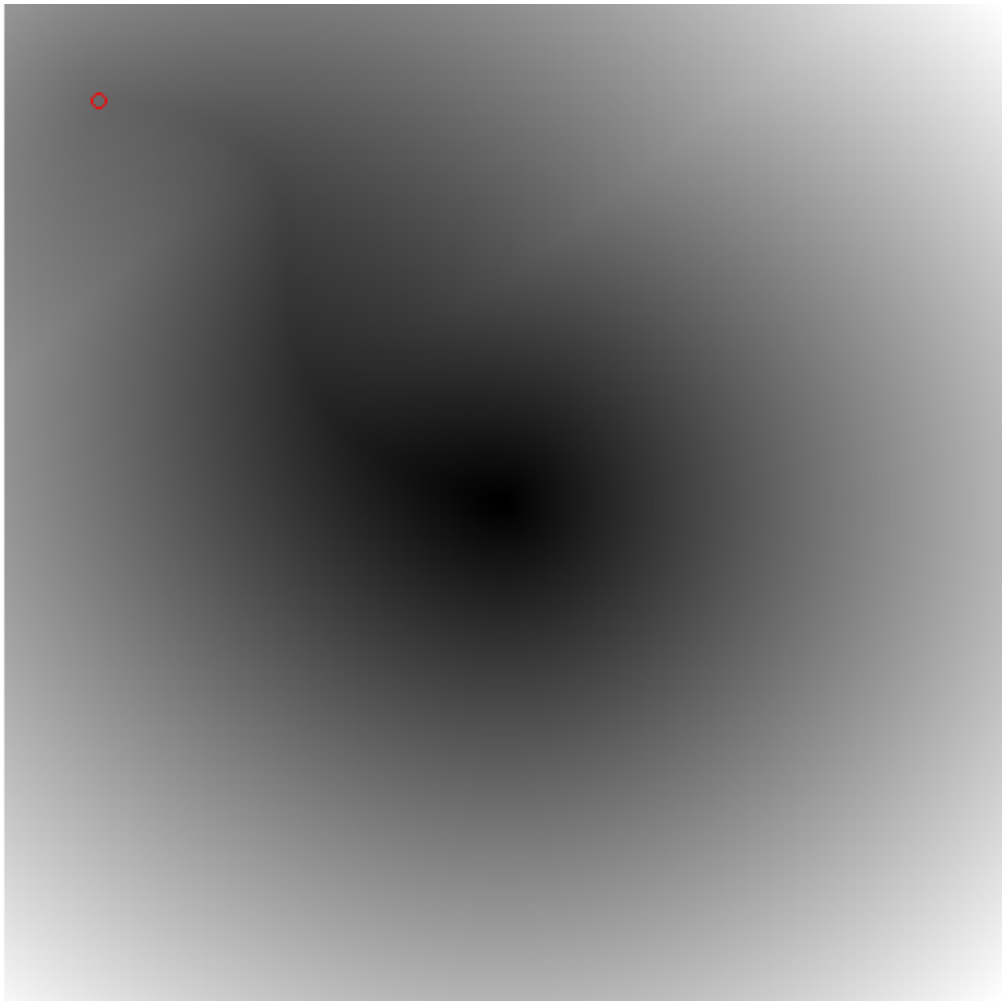

Figure 1: The image used in the example. Nonlinear optimization is used to find the pixel with the smallest value given a starting location (red dot).

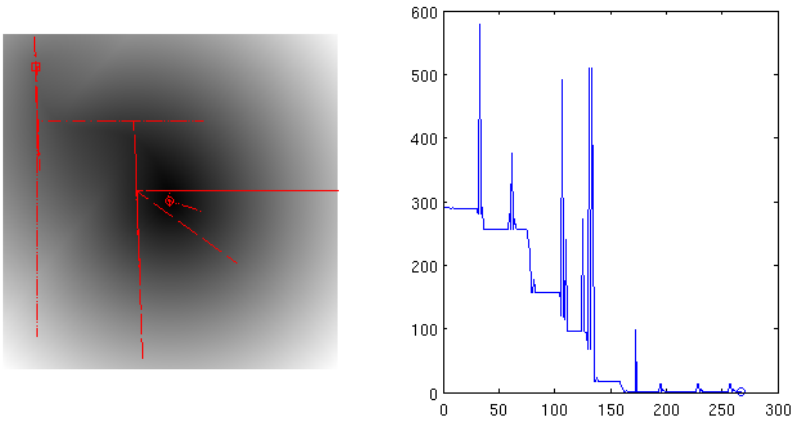

Figure 2: Left: Optimization using the Powell optimizer. Right: Cost function at each step.

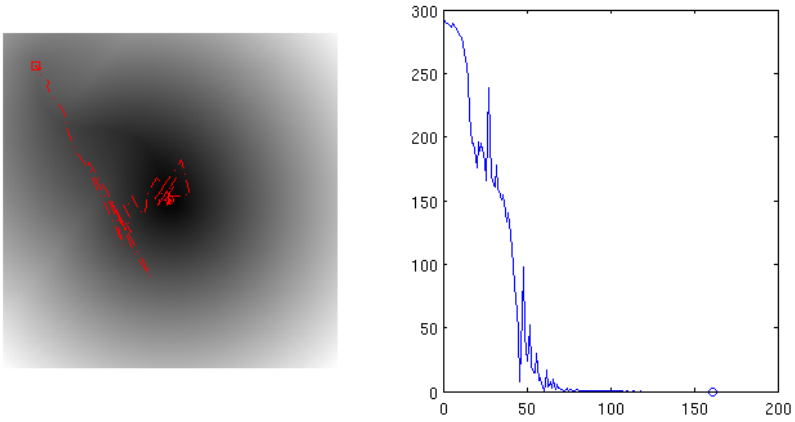

Figure 3: Left: Optimization using the Nelder-Mead optimizer. Right: Cost function at each step.

2 Getting Started

This tutorial assumes that the user knows the basics of CMake, and has installed VXL (which includes the VNL libraries) and the CUDA Toolkit on their system. If not, please use your favorite search engine to find the websites for these packages, and then 1) install CMake, 2) install VXL using CMake, and 3) install CUDA, which can be done using installation scripts or packages from NVIDIA's website. In any case, the starting point for this tutorial is the successful installation of VXL and CUDA.

Once this is done, some changes need to be made to `demoroot/Source/src/CMakeLists.txt`:

1. On line 9, change the path to the location where the vxl binaries were installed on *your* computer. This folder should have a file called "UseVXL.cmake".
2. On line 16, change `sm21` to reflect the proper compute capability of you GPU card (this can be found on NVIDIA's website or using the program "GPU-Z"). For example, if your card's compute capability is 3.5, change `sm21` to `sm35`.

At this point it should be possible to run CMake to generate a solution file or a MakeFile. It is best practice, but not required, to generate binaries in a different directory from the source libraries. We highly recommend generating the binary files in `demoroot/Source/bin/`, as this folder already contains some files that are hard-coded in the software.

3 Code

The code is contained within the following files:

- `demoMain.cxx`. This file contains the `main` function for the program.
- `demoCostFunction.cxx`. This class is required by VNL and is used to define to cost function. In this implementation, it interfaces with the GPU through its member variable `DemoObject`, which is an object of class `DemoInterface`

- demoCostFunction.h. The header file for demoCostFunction.cxx.
- demoInterface.cxx. This class is designed to interface with the GPU. It is designed to handle host and device (GPU) memory allocations and deallocations in it's constructor and destructor functions, which helps to increase ease of use, as memory allocation problems are often a source of bugs in CUDA programs.
- demoInterface.h. The header file for demoInterface.cxx.
- demoKernel.cu. The file where most of the CUDA specific code, such as kernel functions and C++ wrapper functions, are defined.

Briefly, the program is structured in the following way:

1. In the main function, command line arguments are first dealt with. An object of type demoInterface is the declared. The declaration of the demoInterface object invokes the constructor, which:
   - Reads in a file called "img.bin" storing a 512×512 float image.
   - Allocates memory on the GPU for the image and the cost function value.
   - Does all the work necessary to setup texture memory for the image and binds the image to the texture.
2. An object of type demoCostFunction is declared in main, and given a pointer to the demoInterface object.
3. CUDA timing events are started.
4. Either Powell or Amoeba are called, both of which create an appropriate VNL object for minimization (vnl_powell or vnl_amoeba). Optimizer parameters are set, and the minimize member function is called.
   - Once minimize is called, VNL invokes the demoCostFunction member function f at each optimization step.
   - The f function uses the pointer to the demoInterface object (which was declared in main) to invoke the demoInterface member function ComputeCostFunction.
   - ComputeCostFunction calls CUGetCurrentCost, which is declared in demoInterface.h class header file but is defined in the demoKernel.cu file.
   - The CUGetCurrentCost function is essentially a wrapper for the CUDA kernel that computes the cost function. This is the way in which C++ functions that are not defined in the .cu file can indirectly invoke CUDA device kernels.
   - CUGetCurrentCost sends the current x-y coordinate, chosen by the optimizer, to constant memory. constant memory can be a bit faster than global memory in this cases, because every thread must access this memory simultaneously in typical optimization scenarios.
   - Following transfer to constant memory, CUGetCurrentCost invokes the kernel function kernelComputeCostFunction with a single thread and a single thread block. This obviously doesn't take advantage of CUDA's ability to call multiple parallel threads, but makes the code simple to understand. In this example, we are only keeping track of one 2D point, so multiple threads are not needed.

- The kernel performs a fast `texture` read from the image at the current x-y coordinate, and stores this value in `global` device memory.
- The cost function value is transfered from `global` device memory to the host.
- This repeats until the optimizer has converged, which is handled by the VNL functions.

5. CUDA timing events are stopped.
6. The optimization history, which consists of the x-y coordinate and cost function at each iteration, is written to file.
7. The program terminates.

4 Running demo

`demo` is run with the following arguments:

```
demo optimizer x0 y0
```

where `optimizer` should either be `1` for the Powell optimizer or `2` for the Nelder-Mead (Amoeba) optimizer, `x0` is the initial x-coordinate, and `y0` is the initial y-coordinate.

If the user has access to `MATLAB`, a script (`demoroot/Source/bin/DisplayOptimization.m`) can be used to run the program and visualize the optimization process (Figs. 2 and 3). Once in the `MATLAB` environment, change the working directory to `demoroot/Source/bin/` and call `DisplayOptimization(n,x0,y0)`, where `n` is the optimizer to use (Powell=1, Nelder-Mead=2), and `x0` and `y0` are the starting coordinates. For example, `DisplayOptimization(2,50,50)` will produce Fig. 3.

5 Conclusion

A simple program has been presented for CUDA-accelerated nonlinear optimization. This program is meant to help researchers understand how nonlinear optimization works, as well as how to get open-source optimization libraries working with CUDA. This program can be used as a starting template for more complicated optimization problems, such as rigid and non-rigid medical image registration.

References

[1] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. Medical image processing on the gpu–past, present and future. *Medical image analysis*, 17(8):1073–1094, 2013. (document)

[2] Luis Ibanez, Will Schroeder, Lydia Ng, and Josh Cates. The itk software guide: the insight segmentation and registration toolkit. *Kitware Inc*, 5, 2003. (document)

[3] Ramtin Shams, Parastoo Sadeghi, Rodney A Kennedy, and Richard I Hartley. A survey of medical image registration on multicore and the gpu. *Signal Processing Magazine, IEEE*, 27(2):50–60, 2010. (document)