Summary of presentation at 6th UKMAC, 15 Dec 2014, M. Giles, P. Kelly, S. McIntosh-Smith, G. Pullan, F. Spiga and D. Thomas, Cambridge.

Improved CUDA 3D Medical Image Registration

W. B. Langdon,
Department of Computer Science, University College London Gower Street, London WC1E 6BT, UK

***Abstract*— A combination of manual and genetic improvement (GI) can optimise a critical component of NiftyReg healthcare industry software across a diverse range of six nVidia graphics processing units (GPUs). The improved K20c kernel gives a speed up > 2000 fold compared to released code on a 3GHz CPU.**

I. Introduction

Genetic programming (GP) and other search based software engineering techniques can automatically optimise the current rate limiting CUDA parallel function in the Nifty Reg open source C++ project used to align or register high resolution nuclear magnetic resonance NMRI and other diagnostic NIfTI images.

Future Neurosurgery techniques will require hardware acceleration, such as GPGPU, to enable real time comparison of three dimensional in theatre images with earlier patient images and reference data. With millimetre resolution brain scan measurements comprising more than ten million voxels the modified kernel can process in excess of 3 billion active voxels per second (see Figure 1). To be used in theatre, the system must be real time [1], [2], ruling out cloud and other off-site solutions. And yet even a simple task of superimposing today's data with pre-surgery data is computationally heavy.

II. Nifty Reg Kernel

The released `reg_spline_getDeformationField3D` CUDA kernel which, even though running in parallel, typically dominates run time since it can be run $\approx 10^5$ times. It takes the current deformation, expressed as a δx, δy, δz vector at regularly spaced grid control points, and returns the corresponding deformation for every active voxel in the image. For a typical 217^3 (10 218 313 voxel) image there are $47^3 = 103\,823$ grid control points. Each individual deformation vector is given by a cubic spline calculation involving 4 control points in each of the three dimensions (a total of $4 \times 4 \times 4 = 64$ neighbouring control points).

The released code was modified by hand to enforce a fixed grid spacing allowing spline co-efficients to be precalculated and the data access pattern changed to increase texture cache locality. The new kernel was then machine optimised, allowing both tuning key CUDA parameters (e.g. block size) and further coding changes.

As an efficiency measure, instead of each integer referring to a single voxel it now represents a $1 \times 5 \times 5$ volume of 25 voxels, with y,z corners lying at grid control points and thus having the same 64 neighbouring control points. Thus an individual warp (32 threads) now calculates 25 voxels in parallel. Despite discarding 7 threads of every 32, considerable performance gain is made as all the data for the 25 voxels is read once, rather than being read individually 25 times.

Each of the twenty five active voxels needs data from the same 64 control points. In fact since they share the same x-location, these reduce to just 16 values. These are calculated in parallel by 16 (of 32) threads and stored in on-chip fast shared memory.

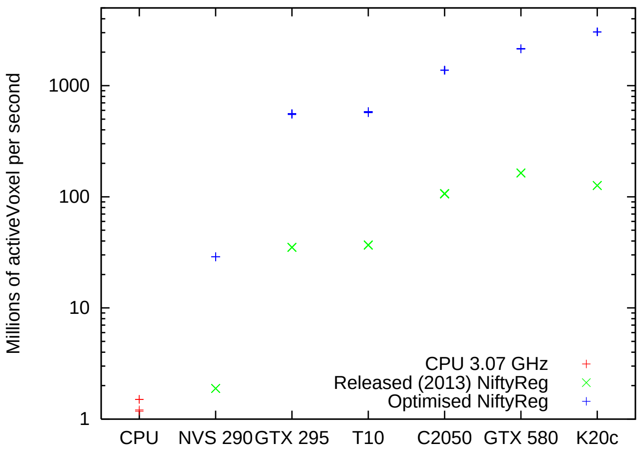

Fig. 1. Performance of modified `reg_spline_getDeformationField3D` CUDA kernel after optimisation by GP, bloat removal and with optimal block size and `-arch`. On a K20c Tesla the original kernel (×) was 93 times faster than its host CPU. The new kernel is 2 243 times faster. Note log vertical scale.

III. Validation

The speeds quoted are given by running the CPU and kernels on 16 816 875 test cases not used during optimisation. In all cases the optimised kernel had an error of 0.000107 or less.

IV. An Example of Code Optimisation

The original condition `(threadIdx.x & 31) < 16` was designed to use 16 threads to calculate 16 values and store them in on-chip shared memory. But if it is removed all threads run (avoiding divergence) and calculate the 16 values twice. When the 32 threads all try to write to 16 shared locations, it is defined that only 16 will succeed. But 16 threads will write the correct values to the 16 shared locations. (The writes from the other 16 threads will simply be discarded.) I.e., GP has removed an operation (`(threadIdx.x & 31) < 16`) and also made the following 23 lines of code more efficient.

References

[1] Youquan Liu and De Suvranu, "CUDA-based real time surgery simulation," *Stud Health Technol Inform*, vol. 132, pp. 260–262, 2008.
[2] W. B. Langdon, et al., "Improving 3D medical image registration CUDA software with genetic programming," in *GECCO '14*, pp. 951–958, ACM.