# Improving 3D Medical Image Registration CUDA Software with Genetic Programming

William B. Langdon Marc Modat Justyna Petke Mark Harman
Dept. of Computer Science, University College London Gower Street, WC1E 6BT, UK
W.Langdon@cs.ucl.ac.uk

## ABSTRACT

Genetic Improvement (GI) is shown to optimise, in some cases by more than 35%, a critical component of healthcare industry software across a diverse range of six nVidia graphics processing units (GPUs). 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.
**Categories and Subject Descriptor** I.2.8 [search]: heuristic

## Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

## Keywords

SBSE, GPGPU, Medicine, Software engineering

## 1. INTRODUCTION

Future brain surgery techniques will require comparison of current images with images of the same patient taken earlier and also with reference data, e.g. from medical atlases. To be used in theatre, the system must be real time [8], 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. The currently favoured solution is to take advantage of the impressive parallel processing abilities of high-end graphics processing units. Such GPUs can easily be placed physically near the surgeon to reduce latency and offer affordable tera-flop performance.

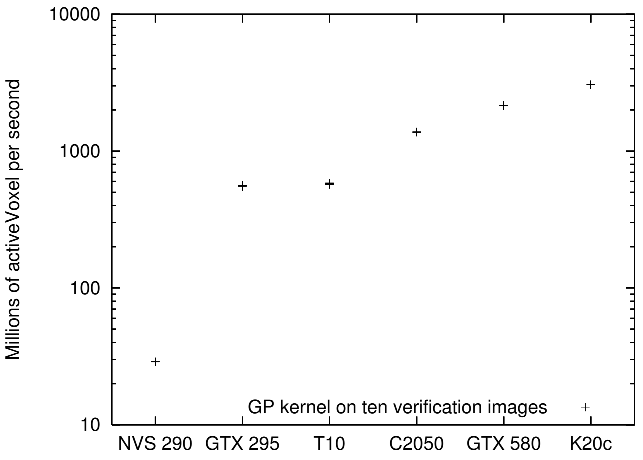

**Figure 1: Performance of modified `reg_spline_get DeformationField3D` CUDA kernel after optimisation by GP, bloat removal and with optimal block size and `-arch`.**

The need to compare or align noisy real-world high resolution three dimensional data occurs widely in industry. Here we concentrate upon the medical domain since this is the target of the open source Nifty Reg software[1]. To take advantage of the cost effectiveness of general computing on graphics processing units GPGPU [10], parts of Nifty Reg have been ported to nVidia's CUDA system. Indeed many CUDA kernels are included in the SourceForge download. However GPGPU programming is acknowledged to be difficult and so only parts of the Nifty Reg tool set can take advantage of GPUs. Even those parts which have been ported to CUDA impose a high software maintenance load and are not necessarily able to make the best use of more recent hardware or software.

Nifty Reg has twin goals of both providing tools and of creating a library of routines which are available to other C++ programmers. These are somewhat in conflict. Tool users want the best tool for what they are doing now and using the hardware they have. Whilst a programmer wants a library which is generic and can be used for many things to support many future users whose aims and environments can only be estimated. Accordingly Nifty Reg has been flexibly coded.

[1]`http://sourceforge.net/projects/niftyreg/`

**Table 1: GPU Hardware. Second column is CUDA compute capability level (as can be used with nvcc's `-arch`). Each GPU chip contains a number of identical and more or less independent multiprocessors (column 3). Each MP contains a number of stream processors (cores, column 4) whose clock speed is given in column 6. Measured data rate (ECC enabled) between the GPU and its on board memory is in the last column.**

| GPU | Capability | MP × | cores | | GHz | Bandwidth |
|---|---|---|---|---|---|---|
| NVS 290 | 1.1 | 2 × | 8 = | 16 | 0.92 | 4 GB/s |
| GTX 295 | 1.3 | 30 × | 8 = | 240 | 1.24 | 92 GB/s |
| Tesla T10 | 1.3 | 30 × | 8 = | 240 | 1.30 | 72 GB/s |
| Tesla C2050 | 2.0 | 14 × | 32 = | 448 | 1.15 | 101 GB/s |
| GTX 580 | 2.0 | 16 × | 32 = | 512 | 1.54 | 161 GB/s |
| Tesla K20c | 3.5 | 13 × | 192 = | 2496 | 0.71 | 140 GB/s |

The Nifty Reg toolset makes heavy use of traditional convex optimisation techniques to find the optimal match between three dimensional images. Since these are based on derivatives, they require many exact calculations. With high resolution images, the basic image calculation may need to be done hundreds of thousands of times. Each one processing millions of voxels. With millimetric resolution NMR images becoming common place this presents a growing need for fast 3D registration. At present this can be met off-line by coarse grained bulk parallel processing in high performance super computer clusters either locally or in the cloud. Such "embarrassingly parallel" approaches work slowly per image but gain performance when each image can be treated independently. However such approaches are not suitable for real-time processing of data for a single patient.

During image registration, with typical hardware and images containing ≈ten million voxels, Nifty Reg is able to fully load a single GPU. In some cases more than 70% of the elapsed time is taken by a single CUDA kernel. We have used search based engineering to improve both the code within the kernel and hardware parameters used to control its parallel running. In so doing we have deliberately traded some of the generic nature of the original code, instead automatically producing code which is tailored to each of six very different graphics cards (see Figure 1 and Table 1).

## 2. NIFTY REG KERNEL

Next we describe the existing CUDA kernel which, even though running in parallel, typically dominates run time and then describe its efficient data access pattern. `reg_spline_getDeformationField3D` is typically used as part of an interactive cycle. Each time it is used it takes the current deformation expressed as a $\delta x$, $\delta y$, $\delta z$ vector at regularly spaced grid control points and returns the corresponding deformation (again as $\delta x$, $\delta y$, $\delta z$ vectors) 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. (47 being 217/5 plus allowance for up to three neighbouring grid points lying outside the image.) 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 calculation of each active voxel requires ≈600 (single precision) floating point calculations.

The original code allowed complete flexibility, with the calling code able to specify any separation of the grid control points. However, in practice, grid control points are not chosen irrationally, instead they often align with voxels. We chose the most popular spacing (each grid point lies every fifth voxel) to demonstrate the effect of using simple integer or half integer spacings. The intention is to support a few popular spacings and that the Nifty Reg library revert to its original non-optimised code if the user chooses others.

Notice this means the 64 distances from each voxel to each of its 64 neighbouring control points now becomes one of a small set (5) of discrete values. Since the 64 spline coefficients depend only on these distances, instead of calculating each on the fly they can be precalculated and stored. By using the fact that the $x$, $y$ and $z$ components are independent only twenty ($4 \times 5$) values need to be stored.

In addition to the 103 823 grid control points, the other large input to the kernel is the list of active voxels for which it must calculate their $\delta x$, $\delta y$, $\delta z$ displacements. The active voxel list is stored as a vector of integer values, which makes it easy for the kernel to process each in parallel. The 32-bit integer values are decoded into the $x$,$y$,$z$ location of the output voxel. This location also readily gives the locations of each of the neighbouring 64 control points.

As an efficiency measure, instead of each integer referring to a single voxel it actually 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) 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. This also reduces the size of the list of active voxels by a factor of 25. At the boundary of the region of interest the lower resolution ($1 \times 5 \times 5$ instead of $1 \times 1 \times 1$) means there is a small increase in the total number of active voxels.

A potential future enhancement would be to use $5 \times 5 \times 5$ volumes all having the same 64 neighbouring control points. So perhaps using 128 threads to calculate 125 active voxels. This has not been tried, partly because the GPU lays data along the $x$-axis so neighbouring data are more likely to be pre-fetched or to remain in cache.

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.

These 16 values are combined with different (but precalculated) spline co-efficients to give the 25 output values that are written simultaneously in parallel by 25 threads. Each output is represented as a `float4`, so 400 bytes are written in one go. Since these are consecutive, the GPU hardware can do this fairly efficiently. (In fact several 32-thread warps are grouped together into blocks so the writes to global memory are even more efficient.)

## 3. EVOLVING CUDA C++ CODE VIA A BNF GRAMMAR USING GP

The CUDA C++ code for the kernel is automatically converted into a BNF grammar of 255 rules. See Figure 2. Essentially each rule represents a line of code. Of these 198 are fixed. Twenty five grammar rules represent ordinary lines of code, six more are C++ #define configuration macros, and four are for C++ if statements. Although not used in the final solutions, for completeness we next describe the more exotic grammar rules.

There are two BNF rules to control CUDA #pragma loop unrolling compiler directives (plus a further 12 rules to control choosing #pragma arguments). #pragma may have been unsuccessful since the kernel contains nested for loops and it appears nvcc may not support #pragma with nested loops.

The CUDA nvcc compiler uses the C++ __restrict__ qualifier, which allows some optimisations. The grammar was automatically generated to include a rule which allows evolution to control the use of __restrict__. Since the kernel has only one argument, which is not read-only, __restrict__ was not expect to have an impact.

The final group of rules control the use of the CUDA __launch_bounds__ qualifier on the kernel. With a further six allowing evolution to control its parameters. Although documented, __launch_bounds__ is fairly obscure. It is thought to have been introduced as an aid to the compiler's register allocation strategy but as the compiler heuristics have improved, newer hardware supports more registers and the CUDA driver overrides the compiler's choices at run time, __launch_bounds__ has fallen out of favour.

## 3.1 Configuration Macros

### 3.1.1 c_UseBSpline *and* c_controlPointVoxelSpacing

In the Nifty Reg code these variables are passed to the kernel via CUDA __constant__ memory. However, as we have evolved efficient kernels for fixed values of c_UseBSpline and c_controlPointVoxelSpacing, they are no longer variable but instead their values are known at compile time. (They are respectively, true and 5). The two grammar rules, c_UseBSpline and c_controlPointVoxelSpacing, give evolution the option of replacing the corresponding variables with their constant values.

### 3.1.2 *BNF rules controlling conditional compilation*

There are four C++ macros which control conditional compilation. These enable evolution to individually control whether code options are enabled. As with c_UseBSpline and c_controlPointVoxelSpacing, the idea is that such code may be beneficial in some cases. This follows some of the ideas in [9], in that the source code is made more generic and we leave it to some later optimisation stage to decide on the exact parameters. However, Merrill et al. [9] were thinking of deterministic optimisation rather than more powerful search based techniques [2].

constantBasis directs the compiler to read pre-calculated spline coefficients from __constant__ memory rather than fetching them from the texture cache. It also forces the use of pre-calculated values for the $x$ dimension. (It may have been better to separate these two aspects into two separated genes.)

Two dimensional arrays or textures are used to store the pre-calculated spline coefficients. Typically the efficiency with which the hardware accesses arrays depends (in an arcane way) on how the array is laid out and how this matches the underlying storage hardware. To avoid thinking deeply about this, the BasisA gene allows evolution to determine which of the two indexes should appear first.

As voxels with multiple $y$ and $z$ coordinates are calculated together, $y$ and $z$ use pre-calculated spline coefficients. Since only one $x$ is in use, the old Nifty Reg code to calculate the associated spline coefficients was left. However the directxBasis option allows evolution to read pre-calculated spline coefficients for $x$ as well as $y$ and $z$. (The compiler may be able to spot that the results of the $x$-calculation are not used and so optimise away their calculation.)

It is often claimed that division and modulus operations are very expensive on nVidia GPUs. Accordingly the RemX configuration macro was introduced to allow the calculation of $x\%5$ to be done just once. However it requires the storage (in new variable int remx) of $x\%5$. It is not clear the reduced calculation justifies the extra storage. RemX is typically used by more than a third of the final population and usually appears in the best of them. However, except in the optimised C2050 kernel, RemX is always removed along with the bloat.

## 3.2 Linear Genetic Representation

The GP works with a linear variable length genome which specifies (via the grammar) changes to the kernel. I.e., the GP evolves not complete programs but changes or patches to programs.

Each genome is stored as a line of text. (See examples on the RHS of Table 4.) The genome is split into genes by spaces. The genes are processed in left right order. Each gene represents one of the 3 fundamental types of mutation: 1) a gene is the name of a BNF rule, meaning that rule, and hence the corresponding line of code, is deleted. 2) a pair of BNF rules, meaning the first rule is replaced by a copy of the second. Typically meaning the first line of code is replaced by the second. 3) As 2) but the pair of BNF rules are separated by a +. This means instead of replacing the first rule, the second rule is inserted before the first. I.e. inserting a copy of the second line of code before the first.

There are several BNF rule types. Where a gene is composed of two rules, they must be of the same type. For example, #pragma rules can only be substituted by other #pragma rules and if condition expressions by other if conditions. Similarly, plain lines of code can be replaced by other simple lines of code.

A new GPU mutant kernel is created by applying the chromosome in left-right order to the BNF grammar and then inverting the modified grammar to create the complete source code of the new kernel. Although all this manipulation is done in plain text, typically it takes less than a second to create a new population and generate 300 new kernels from it. This it is negligible compared to the time taken by nvcc to compile them.

## 3.3 Genetic Operators

### 3.3.1 *Initialisation*

The initial population is composed of 300 different individuals each being exactly one gene. In subsequent generations evolution is free to create duplicate individuals. For each member of the breeding pool one child is created by mutation and one by crossover.

In the rare cases where the number of acceptable parents in the breeding pool is less than half the population size, the missing children are created from scratch by uniformly

Example of grammar rules used to connect GP with C++ code for configuration control parameter `c_UseBSpline` Section 3.1.1

```
<Kkernel.cu_17>            ::= <def_Kkernel.cu_17>
<def_Kkernel.cu_17>        ::= "#define c_UseBSpline 1\n”
```

Grammar rules for code at the start of CUDA kernel function. Also showing grammar for the substitutable conditional expression within an `if` statement.

```
<Kkernel.cu_164>           ::= "//each warp processes 25 z values\n”
<Kkernel.cu_165>           ::= "const int thread = (threadIdx.x & 31); //each warp acts independently\n”
<Kkernel.cu_166>           ::= "const unsigned int tid= (blockIdx.y*gridDim.x+blockIdx.x)*blockDim.x/32+threadIdx.x/32;\n”
<Kkernel.cu_167>           ::= " if” <IF_Kkernel.cu_167> " {\n”
<IF_Kkernel.cu_167>        ::= "(tid<c_ActiveVoxelNumber)”
```

Fragment of grammar rules covering `for` loop and automatically inserted compiler pragma to control loop un-rolling

```
<Kkernel.cu_289>           ::= <pragma_Kkernel.cu_289> "for(c=0;c<4;c++) {\n”
<pragma_Kkernel.cu_289>    ::= ""
<pragma_K0>                ::= "#pragma unroll \n”
<pragma_K1>                ::= "#pragma unroll 1\n”
<pragma_K2 ··· 11>         ::= "#pragma unroll 2 ··· 11\n”
<Kkernel.cu_290>           ::= <pragma_Kkernel.cu_290> "for(b=0;b<4;b++) {\n”
<pragma_Kkernel.cu_290>    ::= ""
```

Fragment of grammar rules covering a simple substitutable line of code, the end of nested `for` loops and code which writes the kernel's output to the GPU's off-chip global memory

```
<_Kkernel.cu_294>          ::= "displacement.z += tempDisplacement(c,b).z * basis;”
<Kkernel.cu_295>           ::= "}\n”
<Kkernel.cu_296>           ::= "}\n”
<Kkernel.cu_298>           ::= "" <_Kkernel.cu_298> "\n”
<_Kkernel.cu_298>          ::= "positionField[tid2] = displacement;”
```

**Figure 2: Fragments of the 255 rule BNF grammar describing the human coded image registration graphics card function which is evolved by GP to create optimised CUDA C++ code.**

randomly selecting from all the possible mutations (i.e., in the same way as those in the initial population).

### 3.3.2 Mutation

Half the new population is created by simply appending another random mutation.

### 3.3.3 Crossover

We use two point crossover. In each of the parents two cut points are randomly chosen. The child is created from the genes at the start and end of the first parent with those from the middle of the second parent inserted between them. Duplicate genes are removed from the offspring and crossovers producing empty children are discarded.

In the very rare cases where, after five trials, crossover is unable to find a legal offspring, its child is created by mutation.

## 3.4 Ensuring for Loop Termination

The kernel code includes two loops, each of which cycle between 0 and 3. To ensure, even badly mutated, kernels always terminate, the GP was prevented from mutating the `for` loop headers or modifying the loop control variables inside the loops.

## 3.5 Protecting Array Indexes

Almost all access to arrays is done via CUDA textures. Textures provide hardware protection against array indexes going out of legal ranges. Only in the case of new GPUs and then only in one case, which accesses on-chip shared memory, was it necessary to provide array bounds protection. This was done by preventing GP from moving lines of code accessing shared memory from inside `for` loops to after them. Inside (and indeed before) the `for` loops the indexes have legal values, problems only arrive in code after the `for` loops, where the loop control values are undefined.

## 3.6 Ensuring Compilation, C++ Scope

To ensure every kernel compiles, GP mutation and crossover are prevented from copying lines of code to parts of the kernel where variables it might contain might be out of scope. Typically this means moving lines of code down the source code is ok. To increase the flexibility allowed to GP, the variable declarations and their initialisation were moved *by hand* to the top of kernel. This increased flexibility increases the search space.

The mutated code causes the nvcc C++ compiler to generate many warnings but the code always compiles, runs, terminates and produces a fitness value.

## 3.7 Efficient Operation of Evolutionary Search

Considerable efficiencies were achieved by compiling and running every member of the GP population together rather than individually. Typically the nVidia compiler (CUDA 5.0) processes in the region of 500 lines of C++ code per second when more than 50 kernels are compiled together. However the curve is fairly flat and nvcc also gives good performance if given more lines of code. Thus, for simplicity, the complete population was compiled in one go [3]. Similarly running the whole population in one image avoids start/stop process overheads. It also makes for easy comparison with both the reference algorithm (i.e. CPU, for correctness) and (for timing) the reference kernel (no mutations). However care must be taken to isolate all read-write data so later mutations cannot take advantage of results created by earlier ones. Omitting all index bounds and loop over run checks, means that to get accurate timings, each kernel need only be run once. I.e., there is no need to run with protection enabled (which has a performance impact, so rendering fitness measurements unusable [7]) and then run successful mutants a second time normally. Typically on a 4GB 2.66GHz desktop it takes about 50 seconds to compile the population and about 30 seconds to run it. The bulk of the runtime being the fitness function which is run on the CPU and compares every answer generated with the correct answer.

## 3.8 Error and Execution Time Based Fitness

Each generation a new random image is created and each GPU kernel is run on it. Fitness is firstly given by the number of GPU clock ticks taken. To avoid overflow, the number of ticks is divided by ten. Notice fitness is minimised. Kernel execution times run from 48 626 to 185 678 246 ticks.

The Nifty Reg CPU code is used to create the ideal correct answer. Typically it contains about 1.7 million active voxels. Each answer created by each GPU kernel is checked against the CPU's. There may be a small discrepancy due to floating point operations. The GP accepts errors smaller than 0.001 without penalty. However if the worst voxel is more than 0.001 from its CPU calculated value a severe penalty is added to the fitness: $213\,748\,364 \times (\text{error} + 0.1)$, subject to not exceeding 213 748 364. In a very small number of cases the mutated kernel may fail to set an answer, in which case there is a penalty of 213 748 364×the fraction of voxels without an answer. If any error exceeds 0.1, it attracts a penalty of 213 748 364×the fraction of wrong voxels. Each of these penalties is added to the original fitness based on the kernel's elapsed time. Since typically Nifty Reg uses the kernel's answer immediately on the same GPU, we use the elapse time on the GPU, excluding any host-GPU interaction time.

## 3.9 50% Truncation Selection

We use 50% truncation selection, with the best half of the population getting two children each and the rest none.

At the start of each generation the original unmutated kernel is run on the current test image. This serves firstly as a sanity check that the GPU is actually running ok and secondly it gives a reference fitness which any improved mutated kernel should beat. Even on the GPU, elapse times are subject to some random variation, therefore when we impose the rule that to be a parent a mutant must be faster than the reference kernel we allow a ten percent tolerance. Once evolution is underway, mutation and crossover typically have no difficulty in generating 150 children which make no errors and take no more than 10% longer than the original kernel. In each case in the initial random population at least 140 random mutants were fast enough to become parents.

## 4. OPTIMISING NIFTY REG KERNELS

## 4.1 Random Fitness Test Cases

Typical high resolution NMR brain scans are represented in the nifti format as 217 × 217 × 217 1 millimetre 3D images with the brain in the centre. We need only processes the region of interest (e.g. the brain). For purposes of testing the software, we can represent the region of interest as a squashed ball.

An effectively unlimited number of test cases can be generated at random. The images are simply deformed spheres of white noise with a diameter of 140 approximately centred in a cube of side 217. The exact radius, centre and degree of compression along each dimension can vary by 5% and each is chosen independently at random. On average 1 700 000 of the 10 218 313 voxels are active. Note each image contains ≈1.7 million test examples, each of which must be correct.

**Table 2: CUDA compiler options. Optimal block size before GP and used during GP runs (column 2). After GP was run and bloat removed block size was re-optimised (column 3) and the best nvcc `-arch` chosen (column 4). The final column gives the run time of the optimised kernels on a 217 cube image.**

| GPU | Block size pre-GP | post-GP | `-arch` | typical mS |
|---|---|---|---|---|
| Quadro NVS 290 | 128 | 256 | sm_11 | 62.40 |
| GeForce GTX 295 | 64 | 512 | sm_13 | 3.22 |
| Tesla T10 | 64 | 480 | sm_13 | 3.09 |
| Tesla C2050 | 128 | 160 | *none* | 2.06 |
| GeForce GTX 580 | 64 | 160 | *none* | 0.84 |
| Tesla K20c | 128 | 128 | *none* | 0.71 |

**Table 3: Genetic Programming for Nifty Reg**

| | |
|---|---|
| Representation: | Variable list of replacements, deletions and insertions into BNF grammar |
| Fitness: | Compile modified CUDA code. Run on about 1.7 million active voxels. Compare its answers with CPU implementation and compare its run time with that of original kernel on the same hardware. |
| Population: | Panmictic, non-elitist, generational. 300 members. New randomly chosen training image each generation. |
| Parameters: | Initial population of random single mutants. 50% truncation selection. 50% two point crossover, 50% mutation. No size limit. Stop after 50 generations. |

## 4.2 Pre-Evolution Parameter Tuning

The kernel was written with one warp per block (i.e. a block size of 32). It is well known that 1) block size plays a critical role in GPU performance and 2) it is almost impossible to calculate the optimal block size from first principles. Thus: for each of the six GPUs, the kernel was individually compiled and run with all sensible legal values of block size. The one giving the fastest elapsed time was chosen and used during evolution. In detail the block size was set to the next multiple of 32 until either the architecture defined limit on block size (which varies between GPUs) or a resource limit, such as number of registers or limit on on-chip shared memory, was exceeded. The used block sizes are given in Table 2.

The GP was run with a population of 300 for fifty generations on each of the six GPUs (see Table 3).

## 4.3 Post Evolution Clean Up and Re-Tune

As expected [11] in all cases the population bloats with rapid approximately linear increase in number of genes per individual. In part this may be due to our mutation operator which increases the length of the child w.r.t. its parent. However since we want the GP to explore different code it is reasonable for it to increase the number of changes it makes. Since these changes include code deletion, increase in GP individual size need not lead to increase in source

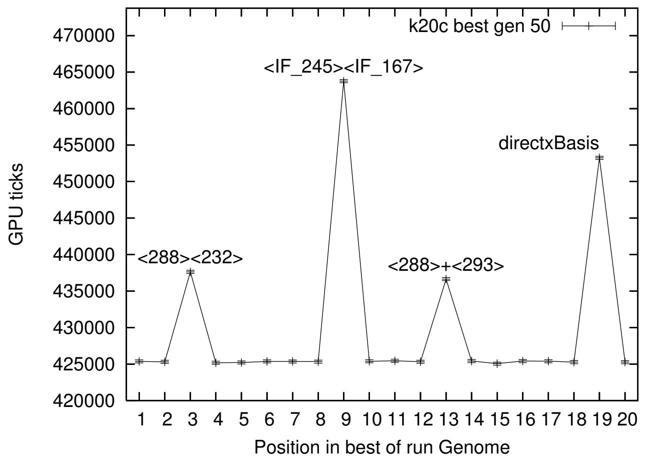

**Figure 3: Example of post-evolution clean up, showing performance of GP evolved Nifty Reg kernel as genes are knocked out. (Tesla k20c). The four spikes correspond to the four of the 19 genes that are vital to its performance. Error bars indicate noise tolerance. Numbers in < > are line numbers in the kernel source code. Line in the left < > is replaced by that in right. < >+< > says insert rather than replace.**

code size, and since nvcc is an optimising compiler, and, especially since we are actively selecting for faster programs, bloat need not lead to an increase in executable GPU code. However in all cases such bloat proved unnecessary for the final solution. We use a simple gene knock out strategy to determine which parts of the best program from the last generation are needed.

The genome of the bloated GP individual is processed one gene at a time in left right order. At each step the next gene is removed; the modified chromosome is applied to the BNF grammar, the grammar reversed to give a new CUDA kernel, which is compiled and run. For repeatability, the same random image is used. The speed of the new kernel is compared against that of the evolved kernel. If it is worse, the gene is retained. If there is no difference the gene is excluded permanently. We then proceed to the next gene, until the whole genome has been processed. In all cases, this leads to a considerable reduction in mutant genome size. To allow for measurement noise, a gene is excluded if it is less than 3.0 standard deviations of the GPU clock measurement noise. See Figure 3.

Finally the optimal block size was determined again (using the technique described in the previous section). Until this point the compiler default architecture setting had been used. This is the setting used by Nifty Reg. In the final step the compiler was run both without the `-arch` switch and with it set appropriately to the target GPU. Surprisingly in some cases, with CUDA 5.0, `-arch` actually produced a slower kernel.

## 4.4 Validation

The final speeds quoted for each GPU are given by running the final kernels on ten configurations not used in the pre- post- or evolutionary stages. In total these amount to 16 816 875 test cases. In all cases the optimised kernel had an error of 0.000107 or less. On the newest GPU, the optimised kernel is more than two thousand times faster than the host CPU (K20c Tesla v. 2.67GHz Intel server).

There is little evidence of over fitting with performance on the unseen examples being close to those used to train the GP population.

Depending upon GPU the genetically improved kernel is from 7 to 39 percent faster than the human code (See Figure 4).

**Table 4: Post-GP Best program length (column 2). After bloat removal length (column 3). Column 4 contains the final automatically generated patch to the manual code.**

| GPU | GP | len | Optimised Program |
|---|---|---|---|
| NVS 290 | 39 | 8 | <291>+<285> <283>+<249> `constantBasis` `directxBasis` <249>+<255> <288>+<283> <249>+<251> <248>+<288> |
| GTX 295 | 25 | 1 | `constantBasis` |
| T10 | 38 | 1 | `constantBasis` |
| C2050 | 26 | 10 | <230>+<283> <252>+<251> `c_UseBSpline` <230>+<291> `RemX` <237>+<292> <IF_245><IF_167> <288>+<293> <IF_239><IF_167> <238><249> |
| GTX 580 | 39 | 3 | `constantBasis` <251>+<283> <IF245><IF167> |
| K20c | 19 | 4 | <288><232> <IF245><IF167> <288>+<293> `directxBasis` |

## 4.5 Evolution of Genes and their Phenotypes

For space, we limit our discussion to common mutations used in the final kernels. `constantBasis` (Section 3.1.2) prospers in all the sm_1x GPUs occurring in almost all members of the final GP populations. However both the later GPUs prefer fetching from textures and `constantBasis` is removed from most of their GP individuals. With the NVS 290, GTX 295, GTX 580 and Tesla T10, `constantBasis` is part of the best of run individual and survives the bloat cull process (Section 4.3) to give a speed up in the optimised kernel (see Table 4).

Except in the case of the C2050 Tesla, `directxBasis` is strongly preferred by evolution and appears in most GP individuals by the last generation, including the best of run individual. (Despite our high mutation rate, `directxBasis` is extinct in the last C2050 population.) With the two powerful high clock rate sm_1x GPUs, `directxBasis` is removed with the bloat by the post evolution clean up phase and only figures in the kernel optimised for the least powerful or slowest clocked GPUs (NVS 290 and K20c Tesla).

## 4.6 Evolved Code and its Implications

The following sections describe in detail the impact of all the mutations that survive the bloat removal stage (see Table 4) and thus (as single genes) have a beneficial impact. Much of the explanation is devoted to why a change does not cause errors. However there are cases (e.g. in Section 4.6.4) where evolution found surprising ways to improve the parallel code.

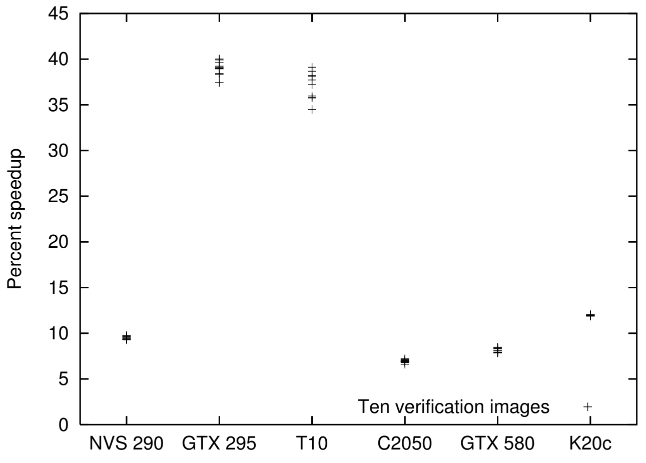

**Figure 4: Speedup of CUDA kernel after optimisation by GP, bloat removal and with optimal block size and `-arch` compared to hand written kernel with default block size (192) and no `-arch`. Unseen data.**

### *4.6.1 Quadro NVS 290*

Inserting additional copies of line 283 before line 283 has no effect since they always set the variable `remz` to the same value. Clearing variable `displacement` multiple times (by inserting copies of line 288) before its first use on line 292 has no effect. Line 251 sets variable `indexXYZ` to its correct value. After line 251 it is used and updated multiple times. However setting its value (by inserting copies of line 251) before line 251 has no effect. Variable `nodeCoefficientD` is set by reading from a texture on line 255 after which its value is used multiple times.

As long as there is one instance of line 249 (at or before line 249) inserting additional copies has no effect as they always overwrite variable `indexYZ` with the same value. Inserting copies of line 249 and line 283 before line 288 has no effect since in both cases the efected variables (indexYZ and remz) are over written with their current values. Inserting another copy of line 285 before line 291 has no effect (it over writes variable tid2 with its current value).

### *4.6.2 GeForce GTX 295*

The GTX 295 is actually a twin GPU (approximately two GTX 280s in a single board). However each of its halves is programmed and used separately. Therefore we report figures for just one half of it. After bloat removal the kernel optimised for the GTX 295 only uses the `constantBasis` configuration control gene (Section 3.1.2).

### *4.6.3 Tesla T10*

Again we report figures for a single GPU, although physically four GPUs are co-located. After bloat removal the kernel optimised for the Tesla T10 only uses the `constant-Basis` configuration control gene (Section 3.1.2).

### *4.6.4 Tesla C2050*

The `c_UseBSpline` and `RemX` genes are both used by the optimised C2050 kernel. They were described in Section 3.1.2. `c_UseBSpline` is only used in one place, in an `if`. Telling the nvcc compiler it will always be true, means it does not have to generate code for the branch itself or for the `else` condition. `c_UseBSpline` appears in between a quarter and three quarters of the final GP populations and in most of the best of run individuals. However, its advantage seems to be quite small since it is removed along with the bloat in all but the C2050.

As above, since the original code to set a variable (`basis` or `remz`) is retained copying the line of code which write to it before it is used has no effect. Evolution used interacting genes to insert multiple lines of code before line 230. It is not clear if this is why our anti-bloat program simplifier was unable to remove these duplicates or if indeed there is some other effect, such as interaction with the hardware or the nvcc optimising compiler's register allocation, that causes the more verbose code to be more efficient.

Inserting a copy of line 292 before line 237 has no effect since the variable (`displacement.x`) is reset by the original code before it is used.

Line 238 `relative=relative>0?relative:0.f;` is designed to ensure that `float relative` is never negative. However the line above will ensure it never is. Hence removing line 238 improves the code. The mutation which deletes it (`<238><249>`) also adds a copy of line 249 in place of line 238. However this has no deleterious effect since the variable `indexYZ` it writes to is reset (by the original code) before `indexYZ` is used.

Replacing the `if` condition on line 245 with that on line 167 is a really innovative optimisation. Firstly line 245 is inside the conditional code controlled by the if on line 167 and therefore the condition must have been true at line 167 (and in fact the compiler can see that it must still be true on line 245). Hence in the mutant, the compiler can remove the code to test the condition. The original condition `(threadIdx.x & 31) < 16` is false half the time. It was designed to use 16 threads to calculate 16 values and store them in on-chip shared memory. It was intentional that the other 16 threads (per warp of 32) do nothing. But if the condition is removed, all threads run in synchrony (which is actually cheaper than having half of them idle) 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 slightly more efficient.

The `if` on line 239 is always true. It appears evolution has repeated its trick of replacing it with another conditional which must also be true (again copied from line 167) and thus allowing the compiler to remove it. However the `c_UseBSpline` gene is also used which has the same effect. It's not clear why the post-evolution bloat removal retained both.

The mutation <252>+<251> makes a copy of line 251 setting variable `indexXYZ` to exactly the same value. Doubtless nvcc can spot this and optimise it away but again it is unclear why the bloat removal stage did not remove it itself.

Mutation <288>+<293> adds updating `displacement.y` immediately before it is set to 0.0f. So it obviously has no effect and hopefully nvcc sees this and removes it. But yet again we would have hoped the anti bloat system would have removed it.

#### 4.6.5 GeForce GTX 580

Pre-calculated spline coefficients are read from `__constant__` memory rather than read from textures (`constantBasis`, Section 3.1.2).

Evolution has again used the trick of removing `(threadIdx.x & 31) < 16`. (Described in detail for the C2050 optimised kernel, Section 4.6.4.)

`<251>+<283>` creates a copy of `remz = thread/5;` before `remz` is used. It is not clear why this is beneficial, perhaps it reduces warp divergence since the copy is executed by all 32 threads, rather than just 25.

#### 4.6.6 Tesla K20c

The `directxBasis` configuration control (see Section 3.1.2) gene is enabled.

Evolution has again used the trick of removing `(threadIdx.x & 31) < 16`. (Described in detail for the C2050 optimised kernel, Section 4.6.4.)

Evolution has used two genes <288><232> <288>+<293> to insert a copy of line 293 in place of line 288. Variable `displacement` is initialised when it is declared and so evolution has spotted clearing it again on line 288 is not needed. However it has been partially replaced by an expression which updates only its `y` component. This appears to be safe only because variable `basis` is zero. Inserting a copy of line 232 is safe since it recalculates the same value. However since its output `nodeAnte.z` is not used after line 232 the nvcc compiler will probably spot this and remove the redundant code. Possibly the change is not removed by the post evolution clean up as there are two genes involved.

## 5. DISCUSSION

Performant parallel programming remains difficult [4]. After several decades of compiler development, it is widely accepted that completely automatic parallelisation using compiler technology is infeasible. Instead nVidia's parallel compiler throws the main tasks of parallel algorithm design and implementation back onto the user. Some in nVidia [9] have advocated this approach be taken further and suggest instead of the programmer coding a solution they should aim to use C++ templates to code generic solutions which the compiler or other tools can tailor, either at compile or run time, to the available hardware. Indeed in future this might take the form of a JIT "just-in-time" approach which dynamically adjusts the code to actual usage [1]. However instead of easing the already insurmountable load on the human programmer a further level of indirection makes it worse.

Ryan [12] and ourselves [6] have shown with very different approaches, that starting with sequential code GP may be able to generate parallel code. We have previously shown GP may substantially improve existing code [5] by tailoring it for a specific task and maintain legacy software for new hardware [7]. Here, instead of creating parallel code from scratch, we use GP to improve existing parallel code. There are some similarities with [9] in that our grammar exposes key parameters to the evolutionary process.

## 6. CONCLUSIONS

A combination of genetic programming acting via a simple automatically produced BNF grammar in combination with simple post evolution bloat removal and tuning of two key compile time CUDA parameters (block size and compute level) can tune state of the art hand coded critical components of medical imaging C++ software. During evolution GP discovered novel and interesting optimisations which may be applied (either automatically or directly) by other software engineers (Section 4.6.4). The resulting graphics card software has been both specialised to the problem and tuned to get the best from a wide range of nVidia hardware (spanning five years since they were launched and more than two orders of magnitude in processing power). The improvement on manual coding varies with GPU type but can exceed 35% on out of sample tests. The evolved and optimised kernels have been run on many millions of tests, all of their answers have been compared to the "gold standard" of running the original Nifty Reg code on the CPU. The difference is always within expected floating accuracy.

## Acknowledgements

I am grateful for the assistance of the anonymous reviewers. Tesla donated by nVidia. Funded by EPSRC grant EP/I033688/1. Code etc. in http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/niftycuda.tar.gz

## 7. REFERENCES

[1] Harman, M., Langdon, W. B., Jia, Y., White, D. R., Arcuri, A., and Clark, J. A. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In *ASE 2012*, ACM, pp. 1–14.

[2] Harman, M., Langdon, W. B., and Weimer, W. Genetic programming for reverse engineering. In *WCRE*, (Koblenz, Germany, 14-17 Oct. 2013), IEEE.

[3] Harris, C. *An investigation into the Application of Genetic Programming techniques to Signal Analysis and Feature Detection.* PhD thesis, UCL. 1997.

[4] Langdon, W. Creating and debugging performance CUDA C. In *Parallel Architectures and Bioinspired Algorithms*, F. Fernandez de Vega, J. I. Hidalgo Perez, and J. Lanchares, Eds., Springer, 2012, ch. 1, pp. 7–50.

[5] Langdon, W. B., and Harman, M. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*. Accepted.

[6] Langdon, W. B., and Harman, M. Evolving a CUDA kernel from an nVidia template. In *WCCI* (Barcelona, 18-23 July 2010), IEEE, pp. 2376–2383.

[7] Langdon, W. B., and Harman, M. Genetically improved CUDA C++ software. In *EuroGP 2014*.

[8] Liu, Y., and Suvranu, D. CUDA-based real time surgery simulation. *Studies in Health Technology and Informatics 132* (2008), 260–262.

[9] Merrill, D., Garland, M., and Grimshaw, A. Policy-based tuning for performance portability and library co-optimization. In *Innovative Parallel Computing (InPar), 2012*, IEEE.

[10] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU computing. *Proc. of the IEEE 96*, 5 (2008), 879–899.

[11] Poli, R., Langdon, W. B., and McPhee, N. F. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[12] Ryan, C. *Automatic Re-engineering of Software Using Genetic Programming.* Kluwer, 1999.