Improving 3D Medical Image Registration CUDA Software with Genetic Programming

W. B. Langdon
Centre for Research on Evolution, Search and Testing Computer Science, UCL, London

GISMOE: Genetic Improvement of Software for Multiple Objectives

10.7.2014
Improving 3D Medical Image Registration 
CUDA Software 
with Genetic Programming

- NiftyReg
- Pre-Post GP tuning of key GPU code
- GP BNF grammar
- Mutation, crossover gives new kernel code
- Fitness: compile, run on random example
- Results: it works, where next?

W. B. Langdon, UCL
Evolving Faster NiftyReg 3D Medical Image Registration CUDA kernels

• What is NiftyReg?
  – UCL CMIC M.Modat sourceForge 16000 C++

• 3D Medical Images
  – Magnetic Resonance Imaging (MRI) brain scans
    1mm resolution → $217^3 = 10,218,313$ voxels

• Registration: NiftyReg nonlinear alignment of 3D images

• Graphics GPU parallel hardware

• CUDA allows C++ functions (kernels) to run in parallel
NiftyReg

- Graphics hardware “ideal” for processing 2 and 3 dimensional images.
- NiftyReg partially converted to run in parallel on GPUs.
- Aim to show GP can help with conversion of remainder or improvement of kernels.
- `reg_bspline_getDeformationField()` 97 lines
reg_bspline_getDeformationField3D

- Chosen as used many times ($\approx 100,000$) 70% GPU (GTX 295) time
- Need for accurate answers (stable derivatives).
- Active region (Brain) occupies only fraction of cube. List of active voxels.
- Kernel interpolates (using splines) displacement at each voxel from neighbouring control points.
CPU v GPU

Note: Log vertical scale

GP K20c 2243 times CPU
Improved Kernels
Original kernel
K20c 93 times CPU

CPU (2.66-3.07 GHz)
NiftyReg 4-Dec-2013
GP kernel on ten verification images
Spline Interpolation

In one dimension displacement is linear combination of displacement at four neighbouring control points:

\[ \text{Displacement} = \alpha d_0 + \beta d_1 + \gamma d_2 + \delta d_3 \]

Spline coefficients \( \alpha, \beta, \gamma, \delta \) given by cubic polynomial of distance from voxel to each control point 0,1,2,3.

In 3D have 64 neighbours, so sum 64 terms. If control points are five times unit distance, there are only \( 4 \times 5 = 20 \) coefficients which can be precalculated.
spline interpolation between $4 \times 4 \times 4 = 64$ neighbours

Control points every $5^{th}$ data point.

$47^3 = 103,823$ control points

All $5^3 = 125$ data points in each control cube have same control point neighbours
reg_bspline_getDeformationField3D

• For each active voxel (≈10^6)
  – Calculate its x,y,z displacement by non-linear B spline (cubic) interpolation from 64 neighbouring control points

• Approximately 600 flops per voxel.
  – Re-use limited by register/shared memory.

• Read voxel list and control points displacement from global memory (via texture cache)

• Write answer δx,δy,δz to global memory
Improve Kernel

• Fixed control grid spline coefficients (20) need be calculate once and then stored.

• GPU has multiple types of memory:
  – Global large off chip, 2 level cache, GPU dependent
  – “Local” large off chip, shares cache with global
  – “Textures” as global but read only proprietary cache (depends on GPU).
  – “Constant” on chip 64K read only cache, contention between threads, GPU dependent
  – “shared” on chip 16-48K, configurable, GPU dependent
  – Registers fast, limited, GPU dependent

• Leave to GP to decide how to store coefficients
GP Automatic Coding

• Target open source system in use and being actively updated at UCL.
• Chose NiftyReg
• GPU already give 15× speedup or more. We get another 25-120× (up to 2243×CPU)
• Tailor existing system for specific use:
  – Images of $217^3$, Dense region of interest,
  – Control points spacing = 5
  – 6 different GPUs (16 to 2496 cores)
## Six Types of nVidia GPUs
### Parallel Graphics Hardware

<table>
<thead>
<tr>
<th>Name</th>
<th>year</th>
<th>MP</th>
<th>Cores</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadro NVS 290</td>
<td>2007</td>
<td>1.1</td>
<td>2 × 8</td>
<td>16</td>
</tr>
<tr>
<td>GeForce GTX 295</td>
<td>2009</td>
<td>1.3</td>
<td>30 × 8</td>
<td>240</td>
</tr>
<tr>
<td>Tesla T10</td>
<td>2009</td>
<td>1.3</td>
<td>30 × 8</td>
<td>240</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>2010</td>
<td>2.0</td>
<td>14 × 32</td>
<td>448</td>
</tr>
<tr>
<td>GeForce GTX 580</td>
<td>2010</td>
<td>2.0</td>
<td>16 × 32</td>
<td>512</td>
</tr>
<tr>
<td>Tesla K20c</td>
<td>2012</td>
<td>3.5</td>
<td>13 × 192</td>
<td>2496</td>
</tr>
</tbody>
</table>
Evolving Kernel

- Convert source code to BNF grammar
- Grammar used to control modifications to code
- Genetic programming manipulates patches
  - Copy/delete/insert lines of existing code
  - Patch is small
  - New kernel source is syntactically correct
- No compilation errors. Loops terminate
  - Scoping rules. Restrict changes to loops and loop variables

W. B. Langdon, UCL
Before GP

• Earlier work (EuroGP 2014) suggested
  – 2 Objectives: low error and fast, too different
  – Easy to auto-tune key parameters:
    • Number of threads, compiler GPU architecture

• Therefore:
  – Single-objective GP: go faster with zero error
  – Pre and post tune 2 key parameters
  – GP optimises code (variable length)
    • Whole population (300) compiled together
Compile Whole Population

Compiling 300 kernels together is 19.3 times faster than running the compiler once for each.

Note Log x scale
Pre and Post Evolution Tuning

1. number parallel threads per block
2. compiler –arch code generation

1. CUDA Block_size parallel thread per block
During development  32
*tune* → 64 or 128
*After GP tune* → 128/512

2. Compiler code -arch sm_10
*After GP tune* → sm_10, sm_11 or sm_13
GP Evolving Patches to CUDA

Original code

BNF Grammar

Test cases

Select

Fitness

Improved system

Population of modifications

Mutation and Crossover

Modified kernel

Population of modifications

W. B. Langdon, UCL
BNF Grammar for code changes

```plaintext
if(tid<c_ActiveVoxelNumber) {

Line 167 kernel.cu

<Kkernel.cu_167> ::= " if" <IF_Kkernel.cu_167> "{"\n<IF_Kkernel.cu_167> ::= "(tid<c_ActiveVoxelNumber)"

//Set answer in global memory
positionField[tid2]=displacement;

Line 298 kernel.cu

<Kkernel.cu_298> ::= ""<_Kkernel.cu_298> "\n<_Kkernel.cu_298> ::= "positionField[tid2]=displacement;"
```

Two Grammar Fragments (Total 254 rules)
Replace variable `c_UseBSpline` with constant

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

In original kernel variable can be either true or false. However it is always true in case of interest. Using constant rather than variable avoids passing it from host PC to GPU storing on GPU and allows compiler to optimise statements like if(1)…
Grammar Rule Types

• Type indicated by rule name
• Replace rule only by another of same type
• 25 statement (eg assignment, Not declaration)
• 4 IF
• No for, but 14 #pragma unroll
• 8 CUDA types, 6 parameter macro #define
Representation

• variable length list of grammar patches.
  • no size limit, so search space is infinite
• tree like 2pt crossover.
• mutation adds one randomly chosen grammar change
• 3 possible grammar changes:
  • Delete line of source code (or replace by "", 0)
  • Replace with line of GPU code (same type)
  • Insert a copy of another line of kernel code
• Mutation movements controlled so no variable moved out of scope. All kernels compile.
• No changes to for loops. All loops terminate
Example Mutating Grammar

<IF_Kkernel.cu_167> ::= "(tid<c_ActiveVoxelNumber)"
<IF_Kkernel.cu_245> ::= "((threadIdx.x & 31) < 16)"

2 lines from grammar

<IF_Kkernel.cu_245><IF_Kkernel.cu_167>

Fragment of list of mutations
Says replace line 245 by line 167

Original code caused ½ threads to stop. New condition known always to be true. All threads execute. Avoids divergence and pairs of threads each produce identical answer. Final write discards one answer from each pair.
Fitness

• Run patched Kernel on 1 example image (≈1.6million random test cases)
  • All compile, run and terminate
  • Compare results with original answer
  • Sort population by
    – Error (actually only selected zero error)
    – Kernel GPU clock ticks (minimise)
• Select top half of population.
• Mutate, crossover to give 2 children per parent.
• Repeat 50 generations
• Remove bloat
• Automatic tune again
Bloat Removal

Fitness effect of each gene evolved by GP tested one at a time. Only important genes kept.
Results

• Optimised code run on 16,816,875 test cases. Error essentially only floating point noise. I.e. error always < 0.000107
• New kernels work for all. **Always** faster.
• Speed up depends on GPU
Evolution of kernel population

Gen 0 ½ random kernels produce incorrect answers.

Fraction of incorrect kernels falls to about 1/3

Gen 0 ½ population are error free and within 10%

After gen7 ≥1/3 pop are faster

End or run ≥½ pop speedup ≥28%
Compile and run GP kernel with all credible block_size and chose fastest
NiftyReg Results

Speedup of CUDA kernel after optimisation by GP, bloat removal and with optimal threads per block and -arch compared to hand written kernel with default block size (192) and no -arch. Unseen data.
Tesla K20c  
NiftyReg Code changes

<table>
<thead>
<tr>
<th>Remove CUDA code</th>
<th>New CUDA code</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define directxBasis 1</td>
<td></td>
</tr>
<tr>
<td>if((threadIdx.x &amp; 31) &lt; 16)</td>
<td></td>
</tr>
<tr>
<td>if(1)</td>
<td></td>
</tr>
</tbody>
</table>
| displacement=make float4(
  0.0f,0.0f,0.0f,0.0f);
| displacement.y +=
  tempDisplacement(c,b).y * basis;
| nodeAnte.z =
  (int)floorf((float)z/gridVoxelSpacing.z); |

`directxBasis` means pre-calculated X-spline co-efficients are read from texture memory not calculated.

16 idle threads exactly duplicate 16 others.

Two genes `<288><232> <288>+<293>` safe but rely on optimising compiler to remove unneeded code.
GP can Improve Software

• Existing code provides
  1. It is its own defacto specification
  2. High quality starting code
  3. Framework for both:
     – Functional fitness: does evolve code give right answers? (unlimited number of test cases)
     – Performance: how fast, how much power, how reliable,…

• Evolution has tuned code for six very different graphics hardware.

W. B. Langdon, UCL
Where Next

• gzip [WCCI2010] GP evolves CUDA kernel
• Bowtie2 50000 lines C++ [HOT paper Wednesday 11:55] 70x improvement
• StereoCamera auto-port 7x improvement
  GP does everything [EuroGP-2014]
• Babel Pigin 230k line GP and programmer working together [SSBSE 2014 challenge]
• NiftyReg GP clean but working on top of manual improvements. Up to 2234×CPU

W. B. Langdon, UCL
END
Discussion Points

- Where next?
  - 3D images for more types Brain NMR
  - Port/improve other UCL CMIC software
- Code is not so fragile
- Build from existing code (source, assembler, binary)
- fitness: compare patched code v. original
  - Gives same or better answers?
  - Runs faster? Uses less power? More reliable?
Typical Active Part of Image

Typical training data 1,861,050 active Voxels, WBL 15 May 2014

W. B. Langdon, UCL
Voxels processed in x-order so caches may reload at end of line

On average 97 voxels processed per line

1,718,861 active Voxels
To reduce clutter only one 1 in 400 plotted
Improved kernel

Typical training data 1,861,050 active Voxels, WBL 16 May 2014

On average 2481 voxels processed per line (before cache refresh)

1,861,050 active Voxels
To reduce clutter only one 1 in 400 plotted

wlangdon/nifty_reg_gp
Manual code changes

• Specialise to fixed (5) control point spacing
• Package coefficient `__device__ function()` so GP can use or replace by storing pre-calculated values.
• Expose kernel launch parameters for auto-tuner.
• grammar automatically created except for variable scope limits
CUDA Grammar Types

• `#pragma unroll`
• `__restrict__`
• `__launch_bounds__`

• `c_USEBSpline`  
  true
• `c_controlPointVoxelSpacing`  
  5
• `constantBasis`
• `BasisA`
• `directxBasis`
• `RemX`

- Pre-calculate
- Array index order
- Pre-calculate x
- Save x%5
GP Evolution Parameters

- Pop 300, 50 generations
- 50% 2pt crossover
- 50% mutation (3 types delete, replace, insert)
- Truncation selection
- 1 test example, reselected every generation
- 1.5 hours
- Unique initial population (≈hence 300)
<table>
<thead>
<tr>
<th>Remove CUDA code</th>
<th>New CUDA code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * <strong>restrict</strong> disparityMinSSD,</td>
<td></td>
</tr>
<tr>
<td>volatile extern <strong>attribute</strong>((shared)) int col_ssd[];</td>
<td>extern <strong>attribute</strong>((shared)) int col_ssd[];</td>
</tr>
<tr>
<td>volatile int* const reduce_ssd = &amp;col_ssd[(64)*2 -64];</td>
<td>int* const reduce_ssd = &amp;col_ssd[(64)*2 -64];</td>
</tr>
<tr>
<td>if(X &lt; width &amp;&amp; Y &lt; height)</td>
<td>if(dblockIdx==0)</td>
</tr>
<tr>
<td>__syncthreads();</td>
<td>#pragma unroll 3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parameter disparityMinSSD no longer needed as made shared (ie not global)
All volatile removed
Two #pragma inserted
if() replaced
__syncthreads() removed
GP and Software

• Genetic programming can automatically re-engineer source code. E.g.
  – hash algorithm
  – Random numbers which take less power, etc.
  – mini-SAT
• fix bugs (5 \(10^6\) lines of code, 16 programs)
• create new code in a new environment (GPU) for existing program, gzip
• 70 speed up 50000 lines of code
• 7 times speed up for stereoKernel GPU
  
3D NMR Brain scans  
IEEE TEC  
GECCO 2014  
EuroGP 2014  
WCCI 2010
GP Automatic Coding

• Show a machine optimising existing human written code to trade-off functional and non-functional properties.
  – E.g. performance versus:
    Speed or memory or battery life.
• Trade off may be specific to particular use. For another use case re-optimise
• Use existing code as test “Oracle”. (Program is its own functional specification)

W. B. Langdon, UCL
When to Automatically Improve Software

• Genetic programming as tool. GP tries many possible options. Leave software designer to choose between best.

• Port and optimise to new environment, eg desktop→phone (3D stereovision)
What’s my favourite number?
# Bowtie2 Patch

<table>
<thead>
<tr>
<th>Weight</th>
<th>Mutation</th>
<th>Source File</th>
<th>Line</th>
<th>Type</th>
<th>Original Code</th>
<th>New Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>999</td>
<td>replaced</td>
<td>bt2_io.cpp</td>
<td>622</td>
<td>for2</td>
<td>i &lt; offsLenSampled</td>
<td>i &lt; this-&gt;_nPat</td>
</tr>
<tr>
<td>1000</td>
<td>replaced</td>
<td>sa_rescomb.cpp</td>
<td>50</td>
<td>for2</td>
<td>i &lt; satup_-&gt;offs.size()</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>disabled</td>
<td></td>
<td>69</td>
<td>for2</td>
<td>j &lt; satup_-&gt;offs.size()</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>replaced</td>
<td></td>
<td>707</td>
<td></td>
<td>vh = __mm_max_epu8(vh, vf); vmax = vlo;</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>deleted</td>
<td>aligner_sws_se_e_u8.cpp</td>
<td>766</td>
<td></td>
<td>pvFStore += 4;</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>replaced</td>
<td></td>
<td>772</td>
<td></td>
<td>__mm_store_s128(pvHStore, vh); vh = __mm_max_epu8(vh, vf);</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>deleted</td>
<td></td>
<td>778</td>
<td></td>
<td>ve = __mm_max_epu8(ve, vh);</td>
<td></td>
</tr>
</tbody>
</table>

- Evolved patch 39 changes in 6 .cpp files
- Cleaned up 7 changes in 3 .cpp files
- 70+ times faster

offsLenSampled=179,215,892  _nPat=84
The Genetic Programming Bibliography

http://www.cs.bham.ac.uk/~wbl/biblio/

9606 references and 8904 online publications

RSS Support available through the Collection of CS Bibliographies.

A web form for adding your entries. Co-authorship community. Downloads

A personalised list of every author’s GP publications.

blog.html

Search the GP Bibliography at
http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html