Evolving a CUDA Kernel from an nVidia Template

[W. B. Langdon]

CREST lab,

Department of Computer Science

Introduction

- Using genetic programming to create C source code
  - How? Why?
- Proof of concept: gzip on graphics card
  - Template based on nVidia kernel
  - BNF grammar
  - Fitness
- Lessons (it can be done!)
- Future? GP to optimise kernel?

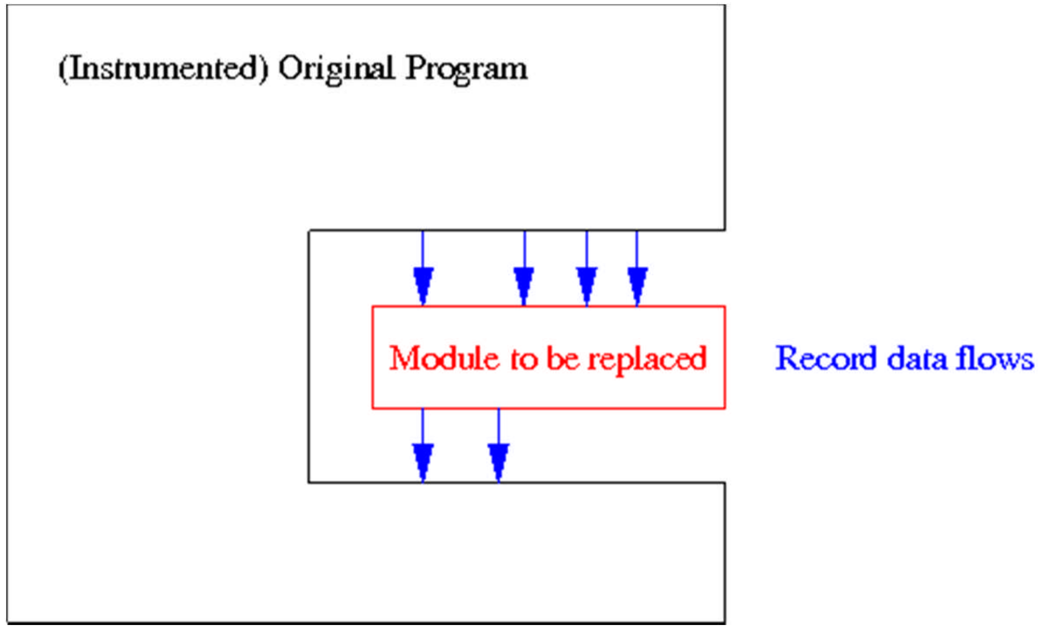
GP to write source code

- When to use GP to create source code
  - Small. E.g. glue between systems.
  - Hard problems. Many skills needed.
  - Multiple conflicting ill specified non-functional requirements
- GP as tool. GP tries many possible options. Leave software designer to choose between best.

GP Automatic Coding

- Target small unit.
- Use existing system as environment holding evolving code.
- Use existing test suite to exercise existing system but record data crossing interface.
- Use inputs & *answer* (Oracle) to train GP.
- How to guide GP initially?
- Clean up/validate new code

GP Automatic Coding



- Actual data into and out of module act as *de facto* specification.
- Evolved code tested to ensure it responds like original code to inputs.
- Recorded data flows becomes test Oracle.

Proof of Concept: gzip

- Example: compute intensive part of gzip
- Recode as parallel CUDA kernel
- Use nVidia's examples as starting point.
- BNF grammar keeps GP code legal, compliable, executable and terminates.
- Use training data gathered from original gzip to test evolved kernels.
- Why gzip
  - Well known. Open source (C code). SIR test suite. Critical component isolated. Reversible.

CUDA 2.3 Template

- nVidia supplied 67 working examples.
- Choose simplest, that does a data scan. (We know gzip scans data).
- Naive template too simple to give speed up, but shows plausibility of approach.
- NB template knows nothing of gzip functionality. Search guided only by fitness function.

scan_naive_kernel.cu

```
//WBL 30 Dec 2009 $Revision: 1.11 $ Remove comments, blank lines. int g_odata, uch g_idata. Add
strstart1 strstart2, const.
 move offset and n, rename n as num_elements
WBL 14 r1.11 Remove crosstalk between threads threadIdx.x, temp -> g_idata[strstart1/strstart2]
__device__ void scan_naive(int *g_odata, const uch *g_idata, const int strstart1, const int strstart2)
{
   //extern  __shared__  uch temp[];
   int thid = 0; //threadIdx.x;
   int pout = 0;
   int pin = 1;
   int offset = 0;
   int num_elements = 258;
   <3var> /*temp[pout*num_elements+thid]*/ = (thid > 0) ? g_idata[thid-1] : 0;
   for (offset = 1; offset < num_elements; offset *= 2)
   {
      pout = 1 - pout;
      pin  = 1 - pout;
      //__syncthreads();
      //temp[pout*num_elements+thid] = temp[pin*num_elements+thid];
      <3var> =  g_idata[strstart+pin*num_elements+thid];
      if (thid >= offset)
      <3var> += g_idata[strstart+pin*num_elements+thid - offset];
   }
   //__syncthreads();
   g_odata[threadIdx.x] = <3var>
}
```

BNF grammar

scan_naive_kernel.cu converted into grammar (169 rules) which generalises code.

```
<line10-18>      ::=   "" | <line10-18a>
<line10-18a>     ::=   <line10e> <line11> <forbody> <line18>
<line11>         ::=   "{\n" "if(!ok()) break;\n"
<line18>         ::=   "}\n"
<line10e>        ::=   <line10> | <line10e1>
<line10e1>       ::=   "for (offset =" <line10.1> ";" <line10e.2> ";offset" <line10.4> ")\n"
<line10.1>       ::=   <line10.1.1> | <intexpr>
<line10.1.1>     ::=   "1" | <intconst>

<line10e.2>      ::=   <line10e.2.1> | <forcompexpr>
<line10e.2.1>    ::=   "offset" <line10.2> <line10.3>
<line10.2>       ::=   "<" | <compare>
<line10.3>       ::=   <line10.3.1> | <intexpr>
<line10.3.1>     ::=   "num_elements" | <intconst>

<line10.4>       ::=   "*= 2" | <intmod>

<intmod>         ::=   "++" | <intmod2>
<intmod2>        ::=   "*=" <intconst>
```

Fragment of
4 page grammar

gzip

- gzip scans input file looking for strings that occur more than once. Repeated sequences of bytes are replaced by short codes.
- n^2 reduced by hashing etc. but gzip still does 42 million searches (sequentially).
- Demo: convert CPU hungry code to parallel GPU graphics card kernel code.

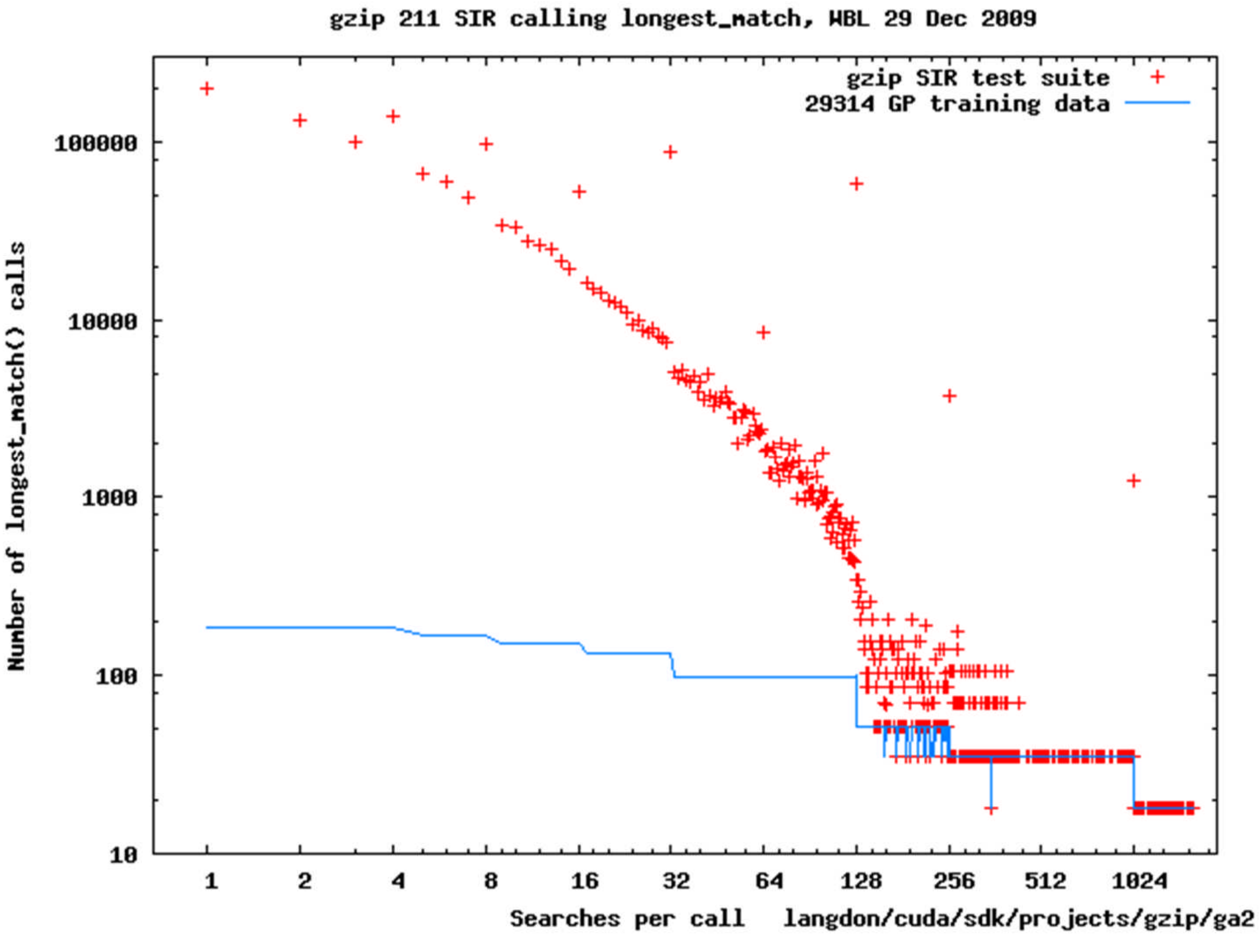
gzip longest_match()

```c
/* ===========================================================================
 * Set match_start to the longest match starting at the given string and
 * return its length. Matches shorter or equal to prev_length are discarded,
 * in which case the result is equal to prev_length and match_start is
 * garbage.
 * IN assertions: cur_match is the head of the hash chain for the current
 *   string (strstart) and its distance is <= MAX_DIST, and prev_length >= 1
 */
#ifndef ASMV
/* For MSDOS, OS/2 and 386 Unix, an optimized version is in match.asm or
 * match.s. The code is functionally equivalent, so you can use the C version
 * if desired.
 */
int longest_match(cur_match)
    IPos cur_match;                             /* current match */
{
    unsigned chain_length = max_chain_length;   /* max hash chain length */
    register uch *scan = window + strstart;     /* current string */
    register uch *match;                        /* matched string */
    register int len;                           /* length of current match */
    int best_len = prev_length;                 /* best match length so far */
    IPos limit = strstart > (IPos)MAX_DIST ? strstart - (IPos)MAX_DIST : NIL;
    /* Stop when cur_match becomes <= limit. To simplify the code,
     * we prevent matches with the string of window index 0.
     */

/* The code is optimized for HASH_BITS >= 8 and MAX_MATCH-2 multiple of 16.
 * It is easy to get rid of this optimization if necessary.
 */
#if HASH_BITS < 8 || MAX_MATCH != 258
   error: Code too clever
#endif

#ifdef UNALIGNED_OK
    /* Compare two bytes at a time. Note: this is not always beneficial.
     * Try with and without -DUNALIGNED_OK to check.
     */
    register uch *strend = window + strstart + MAX_MATCH - 1;
    register ush scan_start = *(ush*)scan;
    register ush scan_end   = *(ush*)(scan+best_len-1);
#else
    register uch *strend = window + strstart + MAX_MATCH;
    register uch scan_end1  = scan[best_len-1];
    register uch scan_end   = scan[best_len];
#endif

    /* Do not waste too much time if we already have a good match: */
    if (prev_length >= good_match) {
        chain_length >>= 2;
    }
    Assert(strstart <= window_size-MIN_LOOKAHEAD, "insufficient lookahead");

    do {
        Assert(cur_match < strstart, "no future");
        match = window + cur_match;

        /* Skip to next match if the match length cannot increase
         * or if the match length is less than 2:
         */
#if (defined(UNALIGNED_OK) && MAX_MATCH == 258)
        /* This code assumes sizeof(unsigned short) == 2. Do not use
         * UNALIGNED_OK if your compiler uses a different size.
         */
        if (*(ush*)(match+best_len-1) != scan_end ||
            *(ush*)match != scan_start) continue;

        /* It is not necessary to compare scan[2] and match[2] since they are
         * always equal when the other bytes match, given that the hash keys
         * are equal and that HASH_BITS >= 8. Compare 2 bytes at a time at
         * strstart+3, +5, ... up to strstart+257. We check for insufficient
         * lookahead only every 4th comparison; the 128th check will be made
         * at strstart+257. If MAX_MATCH-2 is not a multiple of 8, it is
         * necessary to put more guard bytes at the end of the window, or
         * to check more often for insufficient lookahead.
         */
        scan++, match++;
        do {
        } while (*(ush*)(scan+=2) == *(ush*)(match+=2) &&
                 *(ush*)(scan+=2) == *(ush*)(match+=2) &&
                 *(ush*)(scan+=2) == *(ush*)(match+=2) &&
                 *(ush*)(scan+=2) == *(ush*)(match+=2) &&
                 scan < strend);
        /* The funny "do {}" generates better code on most compilers */

        /* Here, scan <= window+strstart+257 */
        Assert(scan <= window+(unsigned)(window_size-1), "wild scan");
        if (*scan == *match) scan++;

        len = (MAX_MATCH - 1) - (int)(strend-scan);
        scan = strend - (MAX_MATCH-1);

#else /* UNALIGNED_OK */

        if (match[best_len]   != scan_end  ||
            match[best_len-1] != scan_end1 ||
            *match            != *scan     ||
            *++match          != scan[1])      continue;

        /* The check at best_len-1 can be removed because it will be made
         * again later. (This heuristic is not always a win.)
         * It is not necessary to compare scan[2] and match[2] since they
         * are always equal when the other bytes match, given that
         * the hash keys are equal and that HASH_BITS >= 8.
         */
        scan += 2, match++;

        /* We check for insufficient lookahead only every 8th comparison;
         * the 256th check will be made at strstart+258.
         */
        do {
        } while (*++scan == *++match && *++scan == *++match &&
                 *++scan == *++match && *++scan == *++match &&
                 *++scan == *++match && *++scan == *++match &&
                 *++scan == *++match && *++scan == *++match &&
                 scan < strend);

        len = MAX_MATCH - (int)(strend - scan);
        scan = strend - MAX_MATCH;

#endif /* UNALIGNED_OK */

        if (len > best_len) {
            match_start = cur_match;
            best_len = len;
            if (len >= nice_match) break;
#ifdef UNALIGNED_OK
            scan_end = *(ush*)(scan+best_len-1);
#else
            scan_end1  = scan[best_len-1];
            scan_end   = scan[best_len];
#endif
        }
    } while ((cur_match = prev[cur_match & WMASK]) > limit
             && --chain_length != 0);

    return best_len;
```

Fitness

- Instrument gzip.
- Run gzip on SIR test suite. Log all inputs to longest_match(). 1,599,028 records.
- Select 29,315 for training GP.
- Each generation uses 100 of these.

Number of Strings to Check

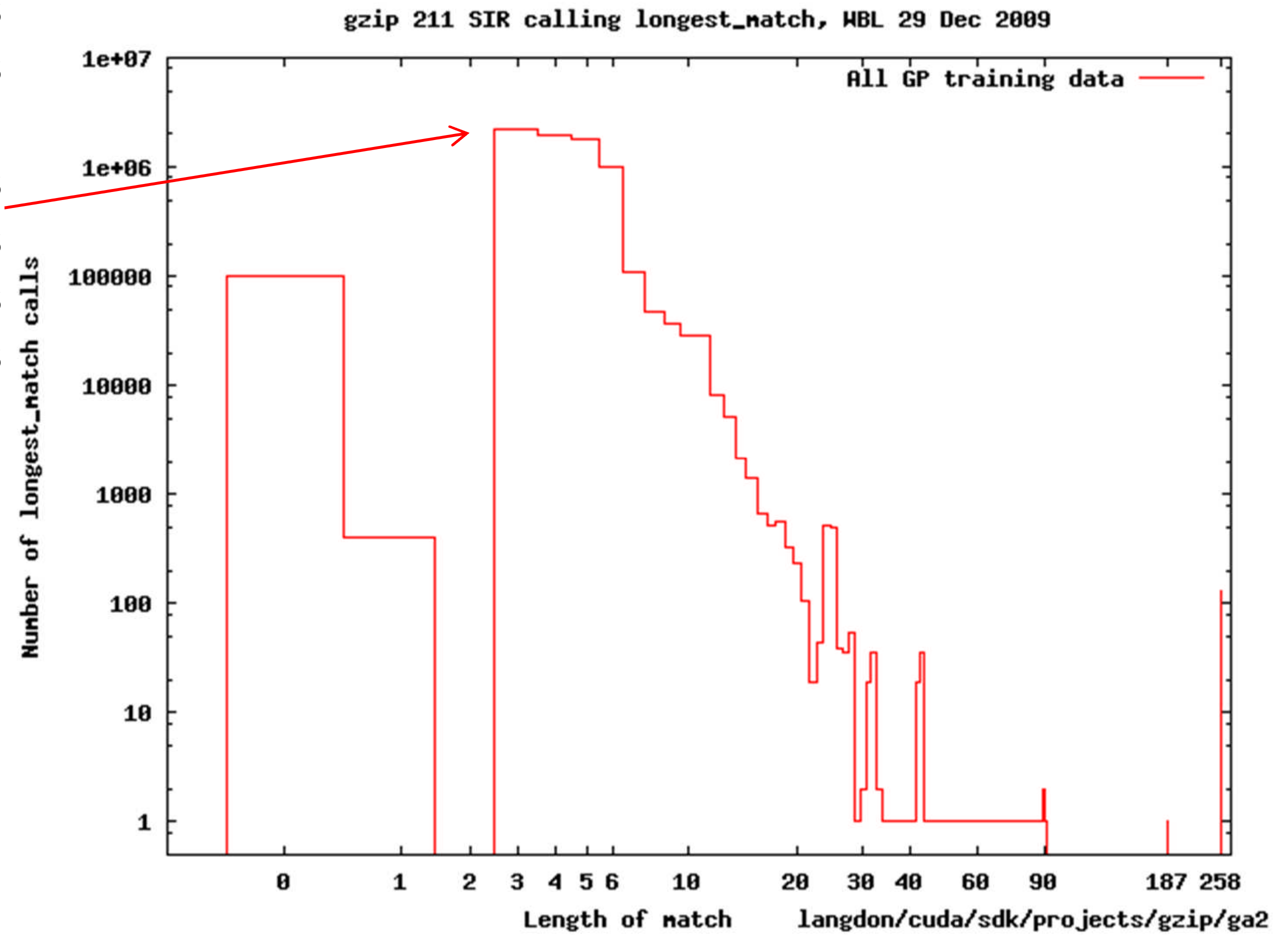


Log scales

gzip hash means mostly longest_match() has few strings to check.
Training data more evenly spread.

Length of Strings to Check

1% 0 bytes
0% 1 bytes
0 2 bytes
30% 3 bytes
26% 4 bytes
25% 5 bytes
14% 6 bytes



gzip heuristics limit search ≤ 258

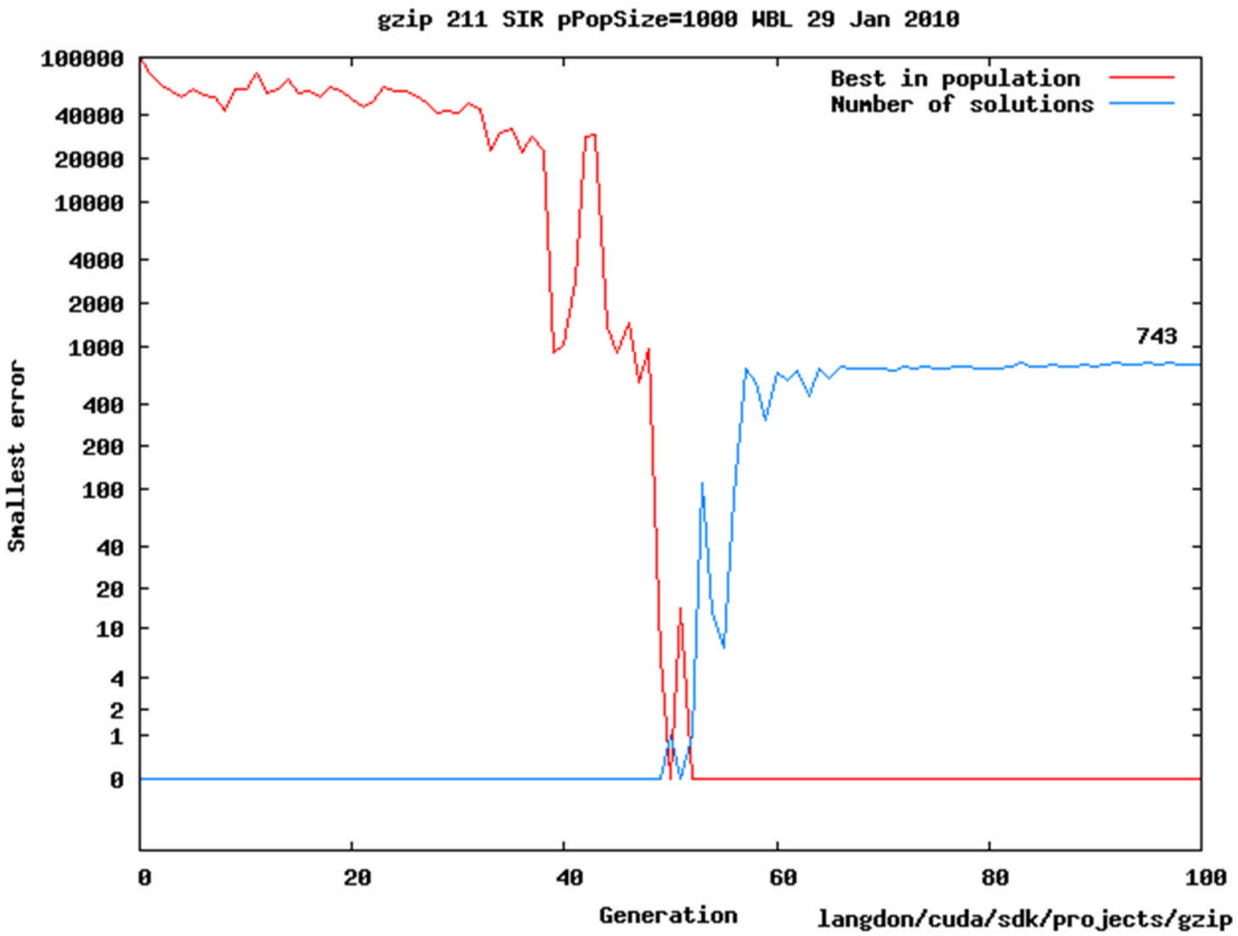
Fitness

- Pop=1000. 100 kernels compiled together.
  - Compilation time = 7×run time.
- Fitness testing
  - first test's data up loaded to GPU 295 GTX.
  - 1000 CUDA kernels run on first test.
  - Each kernel in own block. 1000−1.6 10^6 thread
  - Loop until all 100 tests run.
- Answers compared with gzip's answer.
- performance = Σ|error| + penalty
  - kernels which return 0 get high penalty.

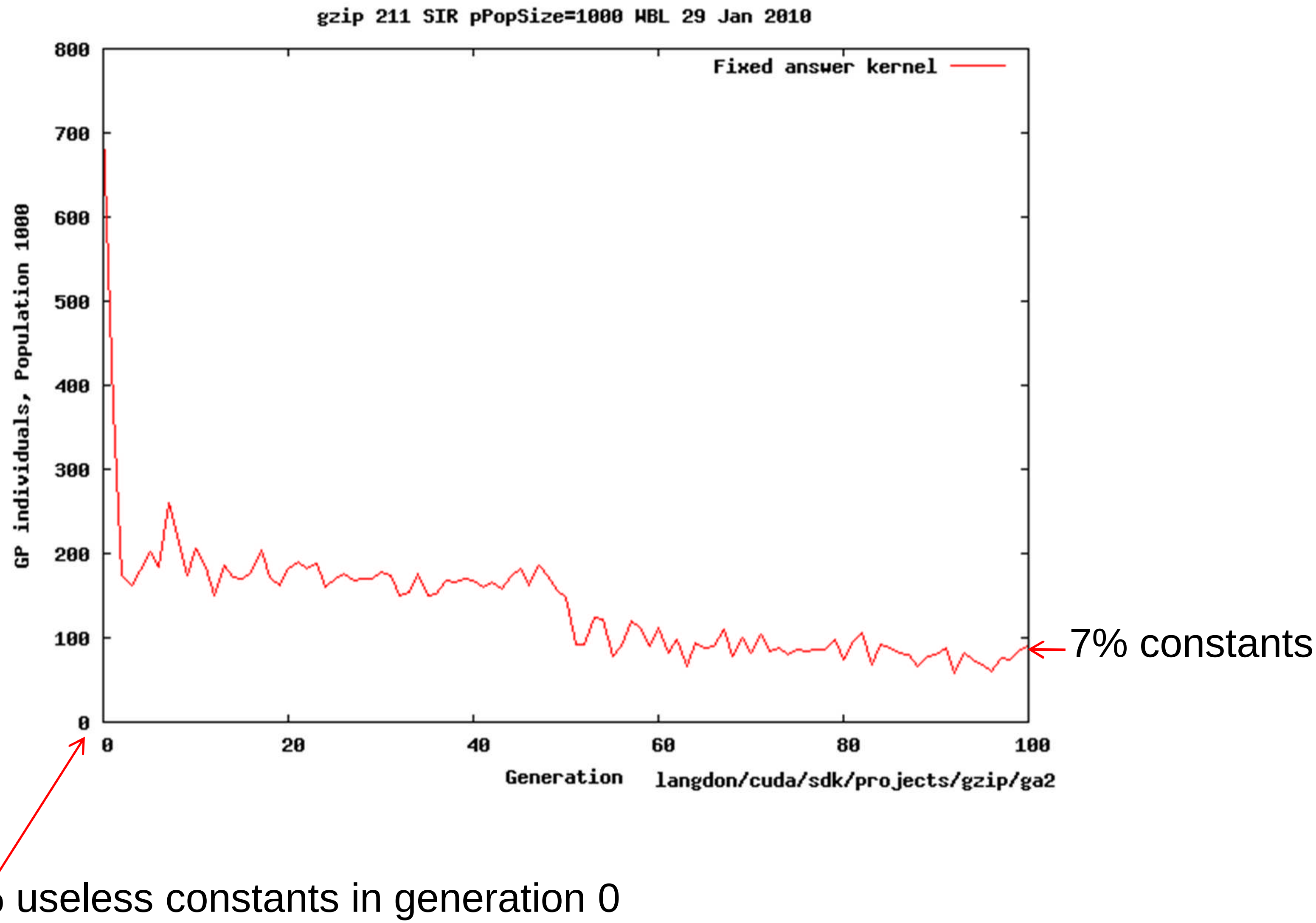
Debug

- Debugging hard
- Eventually replaced last member of evolved population with dummy
- Dummy reflects back input to host PC.
- Enables host to check:
  - Training data has reached GPU
  - Kernel has been run
  - Kernel has read its inputs
  - Kernel's answer has been returned to host PC.

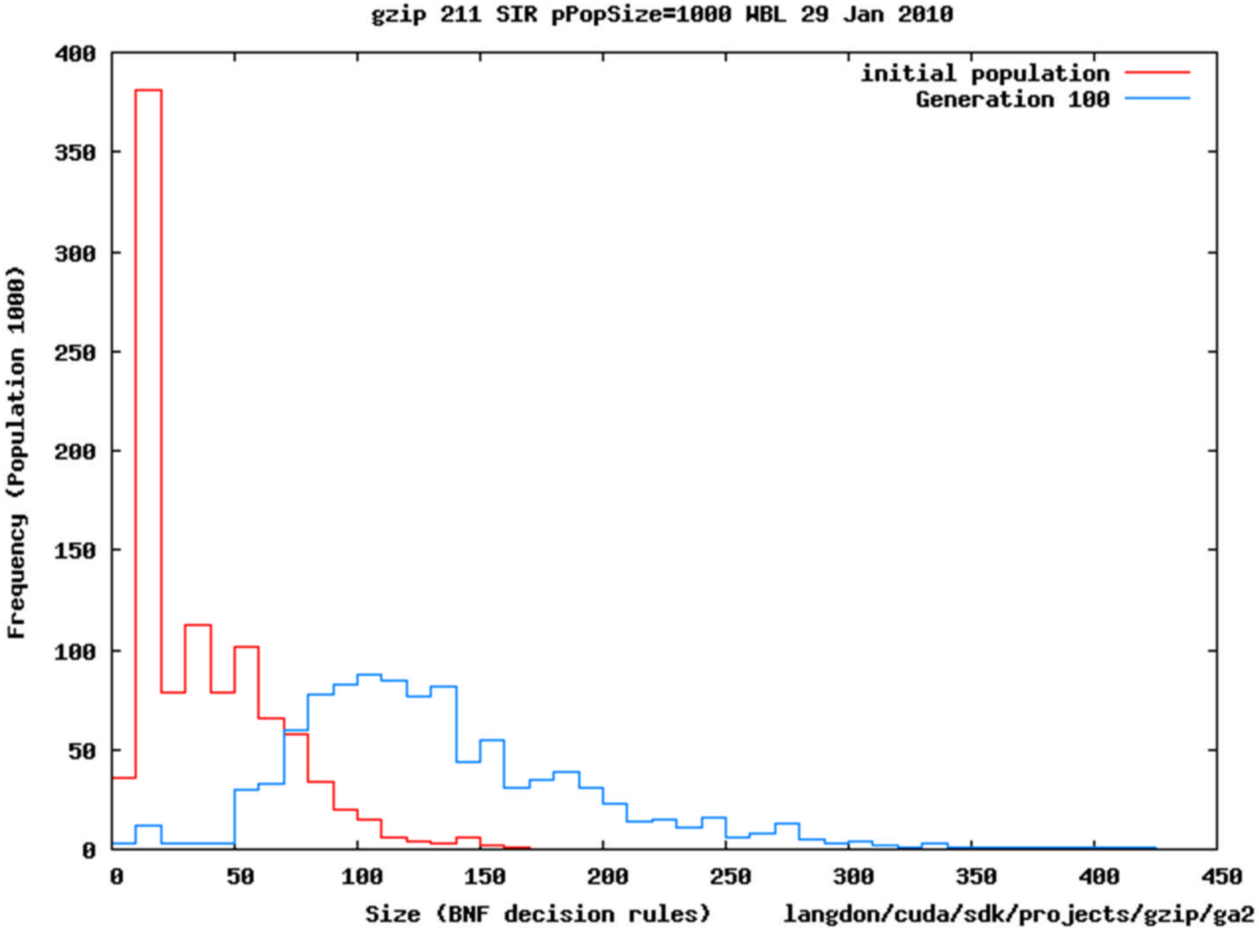
Performance of Evolving Code



Fall in number of poor programs

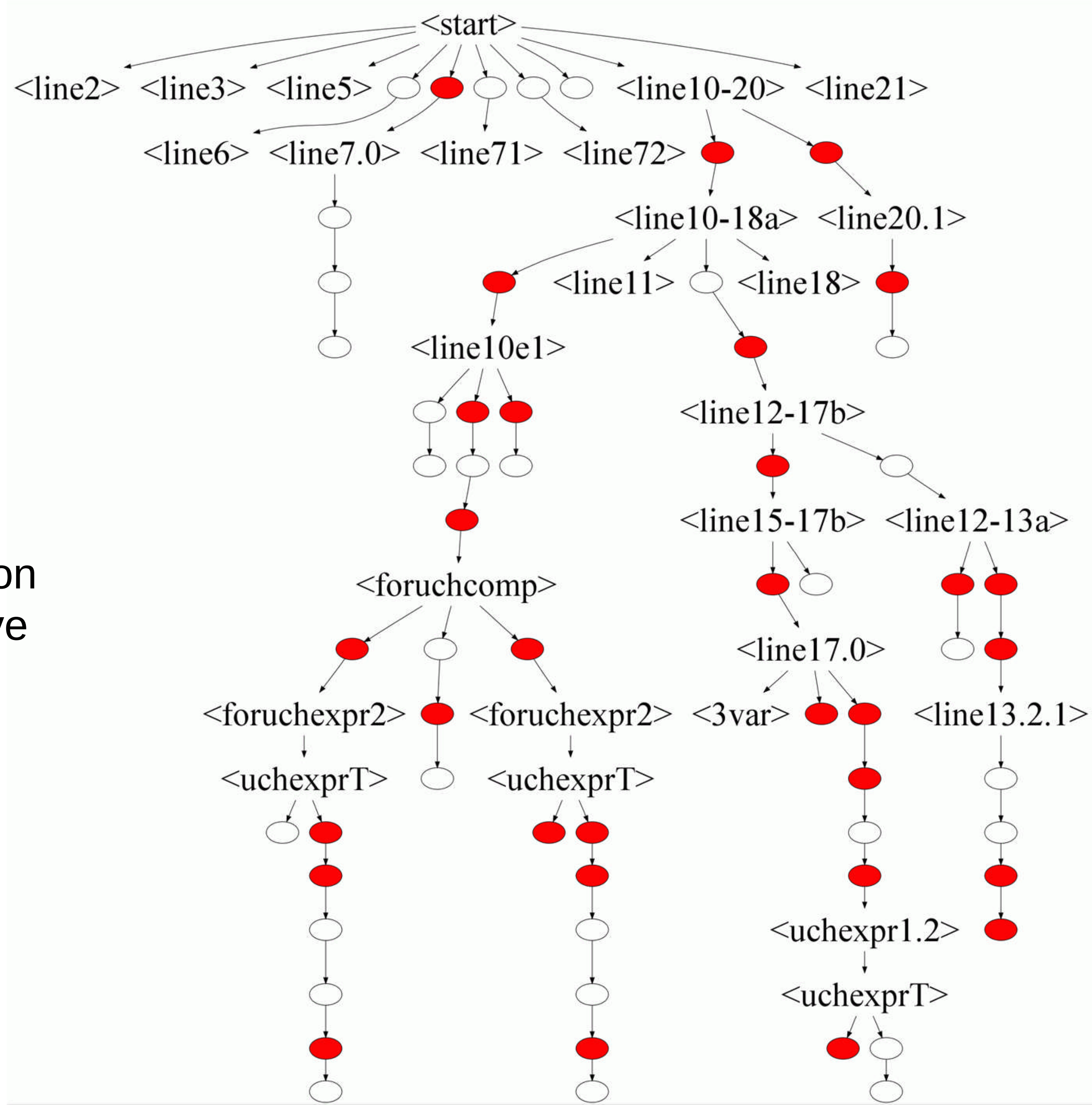


Evolution of program complexity



Evolved gzip matches kernel

Parse tree of solution evolved in gen 55. Ovals are binary decision rules. Red 2nd alternative used.



Evolved gzip matches kernel

```
__device__ int kernel978(const uch *g_idata, const int strstart1, const int strstart2)
{
int thid = 0;
int pout = 0;
int pin = 0 ;
int offset = 0;
int num_elements = 258;
 for (offset = 1 ; G_idata( strstart1+ pin ) == G_idata( strstart2+ pin ) ;offset ++ )
{
if(!ok()) break;
thid = G_idata( strstart2+ thid ) ;
  pin = offset ;
}
return pin ;
}
```

Blue - fixed by template. Red - evolved

Black - default Grey – evolved but no impact.

Conclusions

- Have shown possibility of using genetic programming to automatically re-engineer source code
- Problems:
  - Will users accept code without formal guarantees?
  - Evolved code passes millions of tests.
  - How many tests are enough?
- First time code has been automatically ported to parallel CUDA kernel by an AI technique.

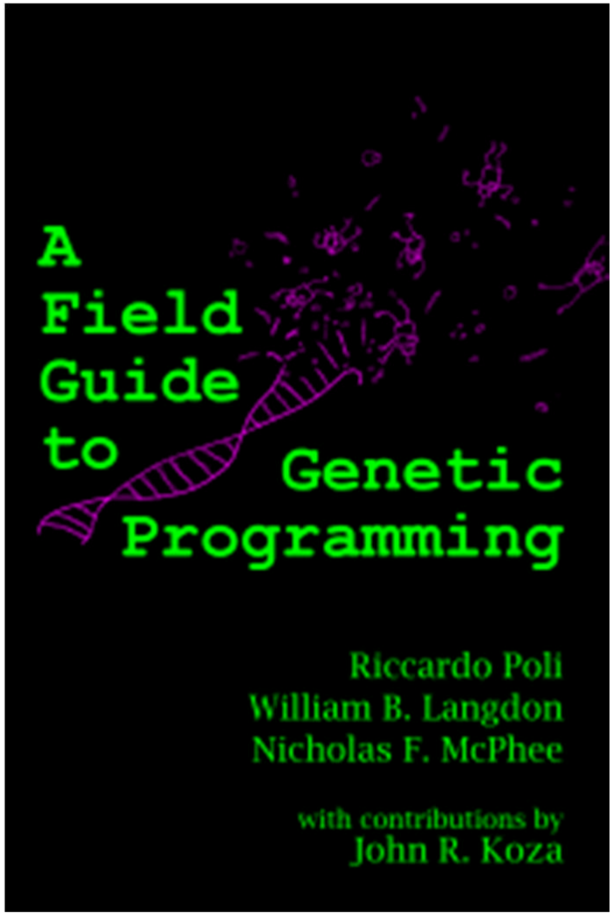
END

http://www.epsrc.ac.uk/

A Field Guide To

Genetic Programming

http://www.gp-field-guide.org.uk/



Free

PDF

The Genetic Programming Bibliography

The largest, most complete, collection of GP papers.
http://www.cs.bham.ac.uk/~wbl/biblio/

Contact W.Langdon to get your GP papers included

href link to list of your GP publications. For example mine is
http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/WilliamBLangdon.html

bibliography

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