Improving CUDA DNA Analysis Software with Genetic Programming

William B. Langdon, Brian Yee Hong Lam, Justyna Petke, Mark Harman
Department of Computer Science, University College London Gower Street, WC1E 6BT, UK
University of Cambridge Metabolic Research Laboratories, Addenbrooke's Hospital, Cambridge CB2 0QQ
W.Langdon@cs.ucl.ac.uk, yhbl2@cam.ac.uk

ABSTRACT

We genetically improve BarraCUDA using a BNF grammar incorporating C scoping rules with GP. Barracuda maps next generation DNA sequences to the human genome using the Burrows-Wheeler algorithm (BWA) on nVidia Tesla parallel graphics hardware (GPUs). GI using phenotypic tabu search with manually grown code can graft new features giving more than 100 fold speed up on a performance critical kernel without loss of accuracy.

Categories and Subject Descriptor I.2.8 [search]: heuristic I.2.2 [Artificial Intelligence]: Automatic Programming D.2.8 [Software Engineering]: Metrics[complexity measures, performance measures]

Keywords SBSE; GPGPU; Bioinformatics; Genetic improvement

1. INTRODUCTION

Modern Biology is an increasingly data rich science. The central dogma of Biology states that the fundamental information for all forms of life is encoded in the DNA carried by every living cell. Next generation DNA sequencing machines were crucial to the decoding of the human genome in 2000 [5] and since have been used to decode the genomes of many other species and to map the variation of individual human genomes [1]. In a single run modern NextGen scanners can give hundreds of millions of short DNA sequences.

Typically before the (noisy) data are used their DNA sequence is located in a reference genome, such as the human genome. Since each can be compared to the reference genome independently, a common strategy is to run multiple instances of the DNA look up software in parallel. Barracuda [6] takes the approach of parallel running on GPU hardware. It is essentially a port of the well established BWA tool [17] written in nVidia's CUDA dialect of the C programming language. Before genetic improvement [8], on typical 100 base pair (bp) DNA data from The 1000 Genomes Project [1], with a top end Tesla K40 GPU, Barracuda processed on average 16 000 sequences per second. Figure 1 shows more than 100 fold speedup for a performance critical component of Barracuda during genetic improvement.

GECCO'15, July 11–15, 2015, Madrid.

ACM 978-1-4503-3472-3/15/07 $15.00.
DOI: http://dx.doi.org/10.1145/2739480.2754652 .

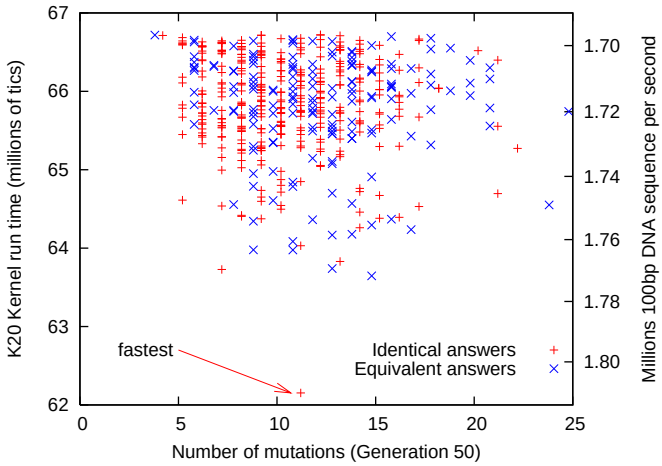

Figure 1: Distribution of speed and number of changes in top 500 correct Tesla K20 `cuda_find_exact_matches` kernels.

Barracuda, now including some of the genetic improvements described in Section 3.1, is available via Source Forge. (We evolved improvements to Barracuda release 0.7.0). Barracuda has been under development since 2009. It comprises 49 C source and include files containing more than 8000 lines of code, containing six CUDA kernels. (In [10] we evolved a complete kernel, whereas here GP [20] improves an existing kernel.) The code which aligns the sequences is largely implemented by the `cuda_split_inexact_match_caller` kernel, which makes heavy use of code in common with `cuda_find_exact_matches`. Typically 85% of NextGen DNA data generated by The 1000 Genomes Project matches the reference human genome exactly. Our approach is to concentrate upon these data and generate huge speed ups with the `cuda_find_exact_matches` code. DNA sequences which do not match exactly are passed to `cuda_split_inexact_match_caller` to be dealt with in the usual way. The complete GPU (device) alignment code is 696 lines of CUDA, whilst the GP works with 200 of these.

On a 2 496 processor Tesla K20 GPU, the original code could process on average 15 000 100bp sequences per second. (Bowtie2 typically processes more than 5000 such sequences per second per CPU core [9, Fig. 10].) The optimised `cuda_find_exact_matches` kernel processes well in

excess of a million such sequences per second. However, we must bear in mind that this speed up will be diluted by the rest of the code, nevertheless the lessons provided by GP on the exact code may possibly be applied to the whole program.

The next section describes our GP system, including the introduction of phenotypic tabu search, whilst Section 3 describes the results of applying it to Barracuda which include average speeds of 1.8–2.4 million 100bp DNA sequences per second (depending on hardware, i.e. K20 or K40). Section 4 includes discussion of our evolutionary process and the ultimate speedup we might get and Section 5 concludes.

2. GENETIC IMPROVEMENT SYSTEM

The manually produced CUDA code was automatically converted into a BNF grammar which describes each of its 200 lines, line by line [12, 19]. (Figure 2 contains fragments of the whole grammar.) The GP uses the grammar to control the mutations the GP can make. The grammar ensures that after each mutation the new code is still syntactically correct (but see Section 2.3).

2.1 Fixed Representation Parameters

The manually produced kernel has 15 configuration options which GP can selectively enable or disable by conditional compilation (see Table 2). This includes allowing evolution to set the number of threads used per DNA sequences query and the number of threads per block to all reasonable values. (Number of configuration options = $(2^{13} \times 7 \times 37 = 2\,121\,728$.) The first part of each GP individual is of fixed length and codes each of these 15 options. (Figure 3 contains part of an example GP individual.) The second part describes each code mutation the GP individual makes via the grammar. Since the second part can contain any number of mutations: the search space of code changes is effectively unlimited. Notice unlike a traditional GA, our GP does not start at random but instead starts from the default (manually written) code. This is similar to seeding the initial population [14].

2.2 Variable Code: Grammar Types

Each line of the source code is represented by one or more rules in the BNF grammar. Evolution can change the contents of the kernel but is not permitted to change its structure. So it cannot remove any of the seven functions, their arguments (except as already permitted by manually written conditional compilation), declarations or return or other control flow statements. However, it can substantially change the code inside functions and between flow control statements including whether functions are called or not. Each variable rule belongs to one of nine types. Mutations only move code between rules of the same type.

43 of the variable rules are simple lines, almost all being assignments (e.g. <_Kkernel_bnf.cu_126> in Figures 2 and 3). Subject to the scoping rules, they can be exchanged with each other, or indeed inserted before other statements.

```
... cache_threads=4 ... <_Kkernel_bnf.cu_126>
```

Figure 3: Fragments of GP individual which sets the `cache_threads` configuration option to 4 (see Table 2). The code mutation `<_Kkernel_bnf.cu_126>` means that line 126 is deleted (see Section 2.6).

There are 24 IF rules. Mutation can delete these by simply setting the condition to always false or (subject to the scoping rules) it can replace the conditional part with the conditional part of another IF rule. There are also two ELSE rules.

There are 7 variable C `for` loops. `for` loop headers are split into their three components (by the `;`). Subject to scooping rules, for1 type rules can be replaced by other for1 rules, for2 by for2 and for3 by for3. However in 4 cases the first part of the `for` loop includes a declaration (such as `for(int i = 0;...)` and so is not variable. We force loop termination via the grammar. The conditional part, `for2`, of every `for` loop includes testing that a local counter has not exceeded its maximum value. Evolution is not permitted to disrupt this in any way. The maximum value is set to twice the maximum number of iterations used in any loop in the manually written code. There is no special penalty on exceeding this limit, instead the fitness function will penalise kernels that calculate incorrect answers or run for a long time. The remaining three types are specials for CUDA.

The CUDA compiler allows the programmer to suggest (via a `#pragma`) that it unroll `for` loops. The grammar allows GP to use unroll pragma before every `for` loop and rely on the fitness function to see if the effect is beneficial. The default is not to use `#pragma`. Optionally each `#pragma unroll` statement takes an integer value from 1 to 11. Again evolution can chose to use this or not.

The compiler supports `__restrict__` on kernel and CUDA device functions. For nVidia GPUs since the K20, if all array arguments are marked `__restrict__`, the compiler may access read only arrays marked `const` via a small read-only cache. Actually this cache is shared with texture data. Since the texture cache may already be in use, it is not clear if reusing it as a read-only data cache will be beneficial. Instead the choice is left to evolution. To support this, the grammar has a single `<optrestrict` rule which evolution can enable or disable. The grammar links all array and pointer arguments so that they are all marked with `__restrict__` or not together. By default `__restrict__` is enabled.

The last CUDA special is the `<launchbounds` rules. These can appear immediately before the kernel is declared. The grammar allows evolution to choose to use it if it wants but ensures the `__launch_bounds__` syntax will be correct and relies on the fitness function to see if it has any benefit. The default is not to use `__launch_bounds`.

The 15 configuration options are implemented via a combination of C macros and conditional compilation. Evolution sets and clears these directly.

2.3 Scoping Rules

Earlier work with GP for bug fixing etc. [11, 13, 16] reported that the main reason why code does not compile is due to variables being moved out of scope. This is also true with the CUDA 6.0 C compiler. Therefore, the BNF grammar is automatically augmented with data indicating the scope limitation of every line of the grammar that can be changed.

A simple text file is used to hold, for each of the 113 variable rules, the locations the rule can be copied to. The locations are given as line numbers in the original source code. Typically these are within the function holding the line itself. Fifteen rules do not depend on any variable declared in a function and so they can be moved to any location, with a matching type, in the kernel. In all but 0.35% of cases the

Table 1: GPU Hardware. Each GPU chip contains 13 or 15 identical independent multiprocessors (MP, column 4). Each MP contains 192 stream processors. (Total given in column 6). ECC enabled.

GPU	Introduced	compute level	MP	total cores	Clock	L1 cache	L2 cache	Memory	Bandwidth
Tesla K20	2012	3.5	13	13 × 192 = 2496	0.71 GHz	48KB	1.25 MB	5 GB	140 GB/s
Tesla K40	2013	3.5	15	15 × 192 = 2880	0.88 GHz	48KB	1.50 MB	11 GB	180 GB/s

```
<Kkernel_bnf.cu_118>          ::= "unsigned int tmp = *data;\n"
                                                    ...
<Kkernel_bnf.cu_119>          ::= " if" <IF_Kkernel_bnf.cu_119> " \n"
<IF_Kkernel_bnf.cu_119>       ::= "(*lastpos!=pos_shifted)"
                                                    ...
<Kkernel_bnf.cu_126>          ::= "" <_Kkernel_bnf.cu_126> "\n"
<_Kkernel_bnf.cu_126>         ::= "*lastpos=pos_shifted;"
                                                    ...
<okdeclaration_>              ::= "#ifndef OKDEC\n" "OKDECLARATION\n" "#define OKDEC 1\n" "#endif
<okdeclaration_End>           ::= "#ifdef OKDEC\n" "#undef OKDEC\n" "#endif /*OKDEC*/\n"
<Kkernel_bnf.cu_415>          ::= <okdeclaration_> <pragma_Kkernel_bnf.cu_415> "for(" <for1_Kkernel_bnf.cu_415>
                                  ";" "OK()&&" <for2_Kkernel_bnf.cu_415> ";" <for3_Kkernel_bnf.cu_415> ")
                                  ((ldg_t*)mycache)[x/inc] = load(x);\n"
<pragma_Kkernel_bnf.cu_415> ::= ""
<pragma_K0>                   ::= "#pragma unroll \n"
<pragma_K1>                   ::= "#pragma unroll 1\n"
                                                    ...
<pragma_K11>                  ::= "#pragma unroll 11\n"
<for1_Kkernel_bnf.cu_415>     ::= ""
<for2_Kkernel_bnf.cu_415>     ::= "x<cache_loads"
<for3_Kkernel_bnf.cu_415>     ::= "x+=inc"
```

Figure 2: Five fragments of the 200 rule BNF grammar describing the human coded kernel which is evolved by GI. <Kkernel_bnf.cu_118> is fixed and cannot be evolved. It simply describes line 118. The grammar can be thought of as having an implicit "start" rule for each line of source code. The conditional part of the `if` on line 119 is given by <IF_Kkernel_bnf.cu_119>. GI mutation can replace `"(*lastpos!=pos_shifted)"` with code taken from another IF. <_Kkernel_bnf.cu_126> is variable and can be deleted, replaced by another variable rule (starting with <_) or another variable rule can be inserted before it. <okdeclaration_> onwards are rules automatically generated to support the `for` loop on line 415. The OK() macro detects and aborts excessively long loops. Evolution can insert #pragmas. The three components of the `for` loop are described by <for1_Kkernel_bnf.cu_415> etc.

scoping rules ensure mutated code compiles (see Figure 4). The only cases where mutants do not compile relate either to conditionally compiled code or to C macro definitions, which do not follow the normal nested scope rules.

2.4 Genotypic and Phenotypic Tabu Search

Initially, some mutations had absolutely no effect as they changed code which was then removed by conditional compilation. To avoid excessive silent mutations in hidden C code we introduced a two stage tabu search to GP.

The tabu list takes two forms: genetic and phenotypic. Nothing is removed from either, i.e. they grow with each generation. The genetic tabu list is simply every genotype that the GP has tried so far. Notice, even though we know there is some noise in the fitness function, we do not allow the GP to re-try complete solutions and its search is forced to diverge.

To prevent conditional compilation from hiding mutations, before GP individuals are compiled into object code the gcc -E C pre-processor is used to convert the GP modified source code into its "phenotype". We take a mutant's phenotype to be its code after conditional compilation and macro expansion. If a mutation changes only code excluded by

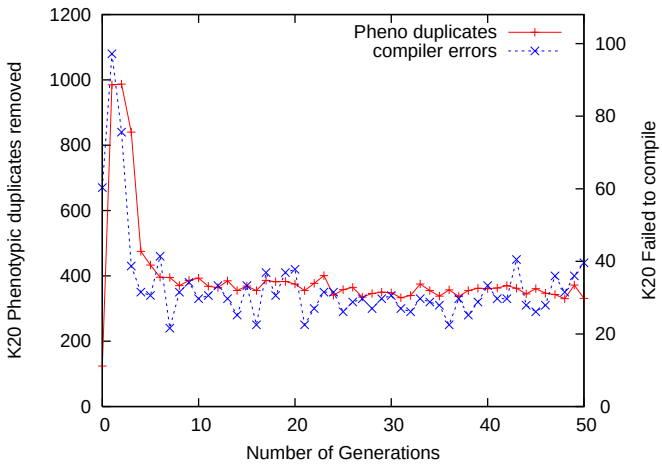

Figure 4: In generation 1, 2 and 3, many phenotypic duplicates (+, Section 2.4) are created. However this quickly falls to about 300 per generation. Although the BNF grammar and scoping rules ensure most mutant kernels compile, typically about 30 of 1000 do not (×).

Table 2: Fixed parameters for improving Barracuda. 3^{rd} column is default. 4^{th} column is number of lines of code directly affected in the default CUDA kernel.

`BLOCK_W`	int	64	all	threads per block. Must be a power of 2 (up to 32) or a multiple of 32 up to 1024.
`cache_threads`	`""` int	`""`	44	number of threads used to load the human genome cache. Must be a power of 2 (up to 32) if used or `""` if not used.
`kl_par`	binary	off	19	Calculate "k" and "l" in parallel. Doubles the number of threads per sequence.
`occ_par`	binary	off	76	Do `bwt_cuda_occ1` loop in parallel (needs `cache_threads`)
`many_blocks`	binary	off	2	Allows number of blocks to exceed 65536 at the expense of more complex index calculations
`direct_sequence`	binary	on	63	Allows `bwt_cuda_match_exact` to read the next DNA base pair directly from global memory rather than using a local array to buffer the query sequence.
`direct_index`	binary	on	6	Calculates location of queries in global memory by requiring them all to be of the same size thus avoiding indirection via `sequences_index_array`
`sequence_global`	binary	on	16	read DNA queries directly from global memory rather than via a texture.
`sequence_shift81`	binary	on	30	Use `>>` rather than `switch` to unpack DNA base pairs (off no longer supported)
`sequence_stride`	binary	on	14	For efficiency DNA queries are blocked in 32 `int` (off no longer supported)
`mycache4`	binary	on	12	Read human reference genome via cache in 4 `int` (i.e. 16 bytes) units.
`mycache2`	binary	off	11	Read human reference genome in 2 `int` units. only active if mycache4 is disabled. If neither `mycache4` nor `mycache2` are active then the cache is loaded one `int` at a time.
`direct_global_bwt`	binary	off	2	read human reference genome from global memory directly. The default is to read it via `__ldg()`.
`cache_global_bwt`	binary	on	65	Read human reference genome via cache.
`scache_global_bwt`	binary	off	35	Use up to 16 local scalars (rather than local array) to implement human reference genome cache. Ideally this means the reference genome cache is held in up to 16 registers rather than in half an L1 cache line.

conditional compilation, the post process code will be identical to the post process code of its parent and therefore it will be excluded by the tabu list. The phenotypic tabu list is simply implemented by keeping the post processed source code of every GP individual accepted into the population at each generation. At 340 MB, the tabu lists remains manageable even after 50 generations. (See also Figure 4.) Hashing these text files might be necessary with bigger populations or more long running GPs. If a mutation is discovered to have the same phenotype as another GP individual it is discarded and another is created.

The phenotypic tabu list is sufficient. However, the genotypic tabu list is also retained since it is easy to detect genetic duplicates and this avoids the overhead of generating source code and running the pre-processor.

We have treated the C source code after pre-processing as the individual's phenotype (rather than its run-time behaviour). This could be extended to later in the compiler tool chain. E.g. we could treat the assembler or even binary machine code as the phenotype. With simple compiler/assembler hierarchy this has been done [18]. However, the nVidia GPU tool stack includes both an optimising assembler and the possibility of just-in-time compilation at kernel load or launch time. The possible presence of version numbers, file names and time stamps embedded in binary files also complicates using differences later in the tool chain as proxies for difference in program behaviour.

2.5 Initial Population

The initial population of 1000 GP individuals is created using mutation. The genotypic and phenotypic tabu list are initialised with the empty (default) individual. This ensures each member of the initial population is unique.

2.6 Mutation and Crossover

Each member of the breeding pool, i.e. the top 500 individuals from the previous generation, is allocated two children. One child is created by mutating the parent. The second is created by crossover between it and another parent randomly selected from the breeding pool. If fewer than 500 individuals in the previous generation were selected, the missing children are created at random in the same way as the initial population was created (but subject to the now longer tabu lists).

Half the mutations are changes to the configuration and half are code mutations. With configuration changes, one of the fifteen parameters is chosen uniformly at random. There is a one in five chance it will be reset to its default value. For binary parameters, mutation simply flips their current value. If `cache_threads` is chosen and is currently undefined half the time it will be set to its default value (4), and half the time it will be set to one of its six legal numeric values. If it already has a numeric value half the time another value is chosen at random and half the time a neighbouring value is chosen. Mutating `BLOCK_W` follows the same strategy of making large jumps half the time and neighbouring moves the other half.

Our code mutations were inspired by those of Le Goues *et al.* [16] but act on the grammar at the level of lines of code rather than abstract syntax trees at the statement level. New code mutations are made by appending another code mutation onto the current mutation list. There are three types of code mutation: delete a line (e.g. right hand end of Figure 3), copy and replace the target line and copy and insert before the target line. E.g. `<_Kkernel_bnf.cu_948><_Kkernel_bnf.cu_927>` causes line 948 `*l0 = l;` to be replaced with line 927 `((int*)k0)[1] = __shfl(((int*)&k)[1], 0, threads_per_sequence);` Insert

mutations are denoted by a + between the rule names. For example, `<_Kkernel_bnf.cu_852>+<_Kkernel_bnf.cu_922>`, which causes line 922 `*k0 = k;` to be inserted before line 852 `*k0 = 0;`

Crossover acts on the genetic representation. It creates a new child with the fixed parameters and a variable length list of code changes obtained from its two parents. The fixed part is obtained by performing uniform crossover [22] on the 15 compulsory parameters, whilst the list of code changes is given by two point crossover, like Koza's sub-tree crossover [7]. Notice crossover only selects genes from its two parents, it does not create new ones. If crossover is unable to create a new unique offspring after a small number of tries, the child is created by mutating the parent instead.

2.7 Fitness Function

It is well known that compiling the population in one go is typically more efficient than compiling each member of the population individually [4]. Therefore, all 1000 GP individuals are converted into a single file that is compiled and linked with the rest of Barracuda to give a single image that runs the 1000 kernels one at a time on about 16 megabytes of the same randomly chosen part of a randomly chosen example file downloaded from The 1000 Genomes Project's FTP site (see Section 3.3).

To allow even loading of the GPU, the number of DNA query sequences run in parallel is chosen to be a multiple of the number of stream processing cores in the GPU, see Table 1. Thus fitness testing on the K20 processes 159744 queries in parallel, while on the K40 161280 are used.

In order to be able to compare the genetically improved code with the original, the first kernel compiled and run is always a copy of the default (manually written) code. I.e. 1001 kernels are compiled and run. Running the compiler typically takes 71% of the total run time. (Mutation and crossover, including checking the tabu lists, takes about 9% and running the kernels and checking their answers takes about 18%. The rest is housekeeping such as monitoring device temperatures.)

We did not investigate parallel compilation, but given that the GPU's are hosted by powerful multicore cluster servers, it would be possible to split the population and compile it in a number of parallel streams [2].

Although our scoping rules (Section 2.3) eliminate almost all compilation errors, there are on average 51 errors per generation (see Figure 4). When a compilation fails, the kernels responsible are eliminated from the population and the new population is compiled. Typically only 1 or 2 attempts at compilation are needed. Notice that the compiler detects and reports errors in a fraction of the time it needs to generate optimised machine code, so running the first part of the compilation process multiple times is a small overhead.

2.7.1 Fitness testing: Burrows-Wheeler algorithm

There are two components to the fitness function: the quality of the answers returned and the speed at which the kernel executes. Quality is measured by comparing all of each kernel's answers one at a time with those given by the reference kernel. Thus each kernel is tested more than 100 000 times. Speed is measured using the GPU's own clocks.

The Burrows-Wheeler algorithm uses two pointers: "k" and "l", which not only indicate where the sequence occurs in the (compressed) human genome but also they indicate the range of possible locations it may match within the reference genome. Also the difference between them is the number of potential matches. k and l are initialised to span the whole of the human genome. As the search proceeds k and l should rapidly approach each other. If the query sequence is not present this is indicated by k exceeding l. If it occurs more than once then k and l will always be a certain distance a part. (This distance is actually the number of repeats.) Barracuda does not deal with ambiguous DNA base pairs in the query sequence, instead it stops the search immediately and reports this by setting k to zero. When the unmodified kernel is run, the k,l pairs it calculates for each query DNA sequence are stored. The fitness of each genetic variant is calculating by comparing the k,l pairs it calculates against those of the reference code. For each test case:

$k = 0$ (i.e. unknown base N in query sequence). Fitness penalty of 100 unless evolved kernel also sets its k to zero. (k' and l' refer to the k and l values calculated by the mutated kernel.) If the evolved kernel says the sequence does match (i.e. $k' \leq l'$) then also get another penalty of 100 (Notice otherwise l' is not checked.)

$k > l$ (i.e. query DNA sequence is not in human reference genome). if $k' \leq l'$ (provided $k' \neq 0$) then get a penalty of 100.

$k = l$ (i.e. exact unique match)

- $k' > l'$ or $k' = 0$ then penalty of 100
- $k' < l'$ then penalty is the number of matches the GP kernel predicts more than it should have, up to a maximum of 100.
- else (i.e. the GP also says they match) the penalty is the distance between k and k', up to a maximum of 10 and the same for the distance between l and l'.

$k < l$ (i.e. more than one match)

- $k' > l'$ or $k' = 0$ then the penalty is 100
- $k' \leq l'$ i.e. GP also says more than one match) The penalty is the difference between the number of matches calculated by the GP and the true number, subject to not exceeding 50.
- else (i.e. GP reports there is a unique match) the penalty is the distance between k and k', up to a maximum of 10 and the same for the distance between l and l'.

2.7.2 Elapse time measurement and variation

Generally, runtimes on GPUs are quite stable. Typically the measured standard deviation is only 0.2% of run time.

2.8 Selection

Like fitness, selection has two parts. Firstly, to be selected a GP individual must get no fitness penalties. Notice there are some cases (actually where the query sequence itself was ambiguous) where a program may get the right answer even though its answer differs in detail from that given by the manually written code. Secondly, it must be faster than the reference code. To allow for noise in the timing, by faster we

Table 3: GP to improve Barracuda

Representation:
Fix 15 discrete parameters (Table 2) plus a variable list of replacements, deletions and insertions into BNF grammar
Fitness:
Compile modified code. K20 run on 159744 (2496×64) (K40 161280 $= 2880 \times 56$) 100bp sequences selected at random from recent NextGen DNA scans generated by The 1000 Genomes Project, Section 3.3. Compare its answers and run time with that of original kernel on the same hardware. See Sections 2.7 and 2.8.
Population:
Panmictic, non-elitist, generational. 1000 members.
Parameters:
Initial population of random single mutants. 50% truncation selection. 50% two point crossover, 50% mutation. No size limit. Stop after 50 generations.

mean the number of GPU ticks it took must be more than one million tics less than the human written code.

At the end of each generation the kernels whose answers attracted no fitness penalties and were faster than the human written code are sorted by their runtime and the first 500 are selected to be parents of the next generation.

3. RESULTS

The GP was run with a population of 1000 for fifty generations (see Table 3). Differences between the K20 and K40 are expected since they have different numbers of processors, clock speeds and memory bandwidths (see Table 1). Thus possibly evolution may find different trade offs between compute and memory operations in the two cases. Also evolution is open to stochastic variations, inherent in using real online timings (Section 2.7.2).

3.1 Choosing the Winner: Tesla K20

In one K20 run the final population contained 703 kernels better than the default starting program, 456 produced identical answers (see Figure 1).

The fastest correct program (11 mutations in Figure 1) contains three configuration changes and eight code changes (see Figure 5). The first configuration change (`scache_global_bwt`) switches from using a local memory array to using 4 `uint4` held in registers for the reference genome cache. The second (`cache_threads=2`) dedicates two threads per query. These are used for loading the reference human genome into this cache. The third (`BLOCK_W=128`) doubles the number of threads per block from 64 to 128.

The unroll pragma (see Section 2.2) is added to two `for` loops by changes 1 and 5. The first is in conditional code which is removed during pre-processing but, as we can see from Figure 5, the second (change 5) does indeed appear to help speed up the code. Changes 2 and 3 are both IF which have no effect. In the first, two conditions which are always true are swapped. Change 3 `<IF_Kkernel_bnf.cu_578>` disables the if statement `if(k == bwt_cuda.seq_len)`. Since k is always less than `bwt_cuda.seq_len` the condition could never be true so removing it does not cause the kernel to give wrong answers. Change 4 has no effect since the variable it changes, `l0`, is immediately updated by next line.

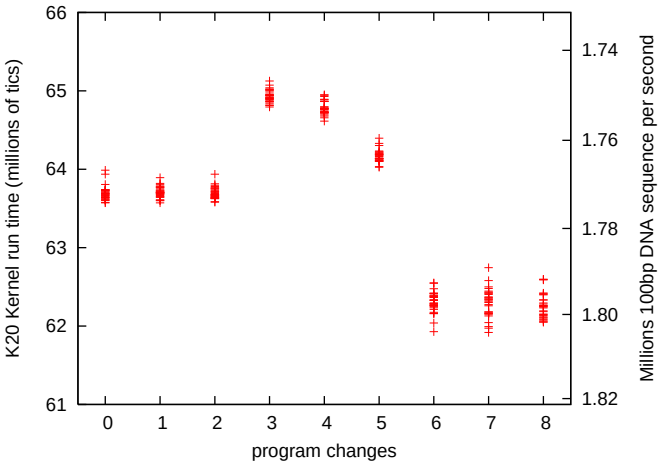

Figure 5: Performance of the fastest correct K20 kernel in last generation as more of its program changes are included.

Change 6, `<_Kkernel_bnf.cu_126>`, deletes line 126. Despite in-depth investigation, it is unclear why this is effective but it surely is. By deleting line 126, GP disables the small (8 base pair) cache used when reading DNA sequences from global memory. Ideally, CUDA will have ensured that these sequences are accessed via the GPU's read-only cache but even so, it would be expected that reading the cache 8 times instead of once would be more costly. Analysis of the CUDA compiler output and using performance tools reveals that deleting line 126 saves a single register but since the number of registers used does not appear to be near a critical number, it is not clear how this would increase performance. It may simply be that by reading the data more frequently they, and more particularly their neighbouring data, are kept in the hardware cache. Nevertheless, despite noise in the fitness function, GP has found this improvement, which is encouraging, since human programmers would be unlikely to discover it without the aid of GP.

Changes 7 and 8 also have no effect since they assign values to variables that are then immediately overwritten or they write a value to a variable which already has this value. In both cases we would expect the optimising compiler to be able to remove the inserted but non-harmful code.

3.2 Evolved Programs: Tesla K40

The fastest program in the last generation of a K40 run contains 9 changes. Although we shall describe the five coding changes in the next paragraph, they have little effect. We start by describing the four parameter changes: 1) instead of each thread processing one DNA sequence query, it uses four `cache_threads=4`. This allows the whole cache (16 words) to be loaded simultaneously. However, GP has not enabled other parallel processing options so each of the 4 threads remain synchronised and calculate the same answer four times. Nonetheless it appears that `cuda_find_exact_matches` is so I/O bound that effectively wasting three computational threads has little down side. 2) Correspondingly evolution increased the number of threads per block from 64 to 128 `BLOCK_W=128`. (This corresponds to 32 DNA queries per block.) 3) This parameter change also relates to the software cache. Setting `direct_global_bwt=1` means the human genome is read directly from global memory rather

than via the hardware read only cache shared with the textures. 4) Again (see Section 3.1 above) `sequence_global` is disabled so that the query sequences are accessed via a texture.

There are five code changes. Two (`<_Kkernel_bnf.cu_948>` and `<_Kkernel_bnf.cu_947>`) delete lines 947 and 948 (`*k0 = k;` and `*l0 = l;`) The GP has "spotted" that both k0 and l0 will have already been set to these values, so deleting these lines does not effect the kernel's answer and may speed it up. Another mutation sets the IF on line 446 to always false. This is fine as line 446 supports configuration `occ_par` which is disabled and so line 446 can never be executed. Disabling `occ_par` also means inserting `#pragma unroll 4` before the `for` loop on line 443 also has no effect. Mutation `<_Kkernel_bnf.cu_852>+<_Kkernel_bnf.cu_853>` has the effect of duplicating line 853, which does not have any harmful effects and the optimising compiler may be able to spot this and remove the duplication.

The speed up derives from using four threads to read data from the reference genome and grouping 128 threads together in a block. (Disabling `sequence_global` and `direct_global_bwt` have only a little effect on the K40). With just the thread changes the K40 processes 2.4 million short DNA sequences per second, a speed up of 54%.

On the K20, disabling `sequence_global` so that the DNA query sequences are read via a texture does have a noticeable effect. Together with `BLOCK_W=128` and `cache_threads=4`, it gives on the K20 a speed up of 28.8%, corresponding to processing 1.8 million DNA sequences per second.

3.3 Performance on Hold Out Data

The training and holdout data were randomly selected from recent popular nextGen DNA sequence data sets from The 1000 Genomes Project. (I.e. normal data, since 2 April 2012, paired-end, non-color space, 100 base-pair per end and not since withdrawn.) None of the hold out data had been used to train the GP. Over ten holdout sets the median speedup was 29.2% (49.1% on K40). The best evolved K40 program gave 53.4% (28.3% on K20) median speedup. This is on average 1 840 000 DNA sequences per second. (2 330 000 on K40) and the evolved K40 gives 2 410 000 (1 850 000 on K20). Remember `cuda_find_exact_matches` only completely deals with a fraction of the sequences. Nevertheless this is a substantial improvement over 15 000 (16 000 on K40) given by the original code averaged over all sequences.

4. DISCUSSION

4.1 Impact of Tabu Search

The impact of Tabu search is most noticeable in the initial random population and early generations (i.e. generations 1, 2 and 3) where almost as many programs are generated and discarded as are needed in the whole population (see Figure 4). As the population evolves the genetic operations find it easier to create phenotypically distinct programs. While in generation 1, only half of unique genotypes are also unique phenotypes, after generation 4, it is 82% on average.

4.2 Effectiveness of Grammar

The BNF grammar and new scoping rules have proved highly effective. The grammar ensures all offspring are syntactically correct CUDA kernel programs. After the first few generations, the population evolves with the scoping rules to ensure on average 96.7% of new kernels compile. Enforce loop termination and array index bound protection ensures all launched GP kernels, run, terminate cleanly and can be assigned a fitness value. (0.06% of configurations were illegal, and so the kernel was not launched.)

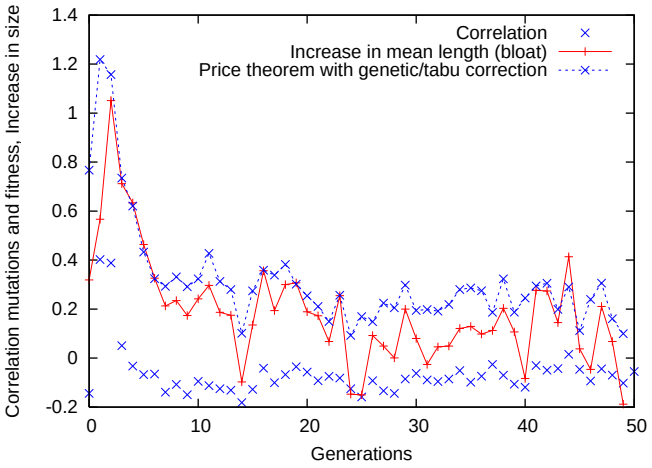

Figure 6: Actual bloat (solid line) v. Price's theorem plus mutation correction (dashed line).

4.3 Price's Theorem: Bloat Explained

Price's covariance theorem [21] applies to the average behaviour of populations and assumes mutation and crossover do not impose a bias on the population property of interest. The theorem says in large populations, the correlation in the current population between any property of interest (e.g. a gene) and the number of children will be equal to the change in the mean frequency of that gene between the current population and the next. Since in GP program size is inheritable, Price's theorem is readily applied to programs' lengths and so bloat is given by a positive correlation between length and fitness [15].

Notice half the time our mutation operator adds another mutation. I.e. it does not strictly follow Price's criterion. However, the dashed-× line in Figure 6 adds a correction for size changes directly caused by mutation and crossover and the tabu requirement to avoid programs which have already been tried. The actual increase is shown with the solid-+ line. In the initial generations the population is skewed by randomly introduced individuals. (Which are new and hence do not inherit from the previous generation). In later generations agreement is better.

4.4 Evolution of Fitness Diversity

Figure 7 shows that those kernels doing better than the manual implementation tend to have more mutations with time (bloat). (The median grows from 2 mutations in the initial random population to 11 by generation 50.) The number of kernels doing better than the manual implementation, and so eligible to be a parent, rises steadily from 136 in the initial population to on average 700 (over generations 10-50).

In the random population 46% of kernels compile and run but do not exceed the manual code. This falls to 11% (averaged over generations 10-50).

During evolution 96% of mutants compiled and 79% ran and returned all correct answers and 66% had no errors and were faster than the original code.

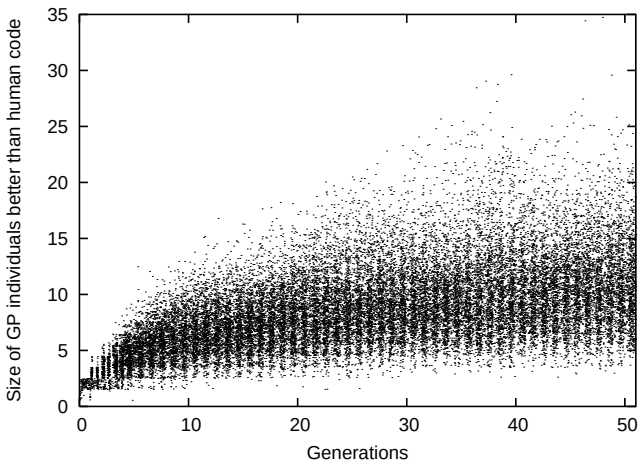

Figure 7: Evolution of number of changes in beneficial mutants (K20). Population size = 1000.

4.5 Performance Limits

The `cuda_find_exact_matches` kernel, indeed the Burrows-Wheeler algorithm itself, is fundamentally limited by the speed at which the human reference genome can be read. This is because modern GPU's have copious compute power for the BWA algorithm itself and the DNA query sequences are small and can easily be well structured for parallel operation. In contrast the BWA algorithm essentially requires random access to the reference genome for each k or l look up.

The genome data is structured so that all data for each k or l lie close together. In about 70% of cases genome data for k and l pairs are close to each other. (Both reasons make the cache of human reference genome data more effective.) Nonetheless for each base-pair in each query sequence approximately 10 integer values must be read. The software cache always reads 16 int, even if only 5 or 6 might be needed. With a K40 kernel processing 230 million base pairs per second (remember 100 base-pairs per query) it will be reading approximately 18GB per second. This is only 22% of the available bandwidth (cf. Table 1). Suggesting that a further factor of four is available.

5. CONCLUSIONS

Barracuda was developed from the widely used BWA tool (according to Google Scholar [17] has been cited more than 1000 times). Barracuda's authors [6] include both DNA sequencing and GPGPU experts. It has been under human development since 2009. Nevertheless genetic programming using phenotypic tabu search in combination with manually grown code can graft [3] speed up changes into the CUDA source. On large real world datasets, typical of current high-throughput DNA analysis, the improvement in a performance critical component is more than 100 times without loss of accuracy.

Over the complete mapping process (i.e. including all the non-CUDA code as well) on fourteen million pairs of DNA sequences with 76 bases each (note shorter sequences than the GP was trained on) barracuda_0.7.105 is 62% faster than Barracuda 0.6.2. Our version of BarraCUDA has been adopted into the core development (as barracuda_0.7.105). In the first month after it was made publicly available it was downloaded 146 times.

5.1 Acknowledgements

I am grateful for the assistance of njuffa, cbuchner1, CudaaduC, txbob, allanmac, Kevin Kang, and the many anonymous reviewers.

Tesla donated by nVidia. Funded by EPSRC grant EP/I033688/1.

GI code available via anonymous FTP and `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/barracuda_gp.tar.gz`

6. REFERENCES

[1] Durbin, R. M., et al. A map of human genome variation from population-scale sequencing. *Nature 467*

[2] Harding, S. L., et al. Distributed GP on GPUs using CUDA. In *Par. Arch. & Bioinspired Alg.*, 2009.

[3] Harman, M., Jia, Y., and Langdon, W. B. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In *SSBSE 2014*, *LNCS 8636*, pp. 247–252.

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

[5] Initial sequencing and analysis of the human genome. *Nature 409*, 6822, (15 Feb 2001), 860–921.

[6] Klus, P., et al. BarraCUDA. *BMC Res. Notes 5*, 27

[7] Koza, J. R. *Genetic Programming*. MIT press, 1992.

[8] Langdon, W. B. Genetically improved software. In *Handbook of Genetic Programming Applications*, A. H. Gandomi et al., Eds. Springer.

[9] Langdon, W. B. Mycoplasma contamination in the 1000 genomes project. *BioData Mining 7*, 3 (2014).

[10] Langdon, W. B., and Harman, M. Evolving a CUDA kernel from an nVidia template. In *WCCI 2010*

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

[12] Langdon, W. B., and Harman, M. Optimising existing software with genetic programming. *IEEE Trans. on Evo. Comp. 19*, 1 (2015), 118–135.

[13] Langdon, W. B., et al. Improving 3D medical image registration CUDA software with genetic programming. In *GECCO 2014*, ACM, pp. 951–958.

[14] Langdon, W. B., and Nordin, J. P. Seeding GP populations. In *EuroGP'2000* pp. 304–315.

[15] Langdon, W. B., and Poli, R. Fitness causes bloat: Mutation. In *EuroGP 1998*, *LNCS 1391*, pp. 37–48.

[16] Le Goues, C., et al. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering 38*, 1 (2012), 54–72.

[17] Li, H., and Durbin, R. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics 26*, 5 (2010), 589–595.

[18] Papadakis, M., Jia, Y., Harman, M., and Le Traon, Y. Trivial compiler equivalence. In *ICSE 2015*

[19] Petke, J., et al. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *EuroGP 2014*, pp. 137–149.

[20] Poli, R., et al. *A field guide to genetic programming*. `http://www.gp-field-guide.org.uk`, 2008.

[21] Price, G. R. Selection and covariance. *Nature 227* (1 August 1970), 520–521.

[22] Syswerda, G. Uniform crossover in genetic algorithms. In *FOGA 1989*, pp. 2–9.