Identifying Similarities in TMBL Programs with Alignment to Quicken Their Compilation for GPUs

Computational Intelligence on Consumer Games and Graphics Hardware

Tony E Lewis
Computer Science and Information Systems
Birkbeck, University of London
London, UK
tony@dcs.bbk.ac.uk

George D Magoulas
Computer Science and Information Systems
Birkbeck, University of London
London, UK
gmagoulas@dcs.bbk.ac.uk

ABSTRACT

The most impressive accelerations of Genetic Programming (GP) using the Graphics Processing Unit (GPU) have been achieved by dynamically compiling new GPU code for each batch of individuals to be evaluated. This approach suffers an overhead in compilation time.

We aim to reduce this penalty by pre-processing the individuals to identify and draw out their similarities, hence reducing duplication in compilation work. We use this approach with Tweaking Mutation Behaviour Learning (TMBL), a form focused on long term fitness growth. For individuals of 300 instructions, the technique is found to reduce compilation time 4.817 times whilst only reducing evaluation speed by 3.656%.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Performance

Keywords

Tweaking Mutation Behaviour Learning (TMBL), Alignment, Graphics Card, Graphics Processing Unit (GPU), CUDA

1. INTRODUCTION

Graphics Processing Unit (GPU) technologies have made it possible to perform Genetic Programming (GP) evaluations at remarkable speeds. Two main techniques exploit the GPU differently: data-parallel techniques dynamically write and compile GPU code for each individual [1] [2]; population-parallel techniques use an interpreter running on the GPU to evaluate individuals by treating them as data [3] [6].

GECCO'11, July 12–16, 2011, Dublin, Ireland.

Since the code for individuals in a data-parallel system is directly compiled and does not need to perform interpretation, it typically executes much faster on the GPU than equivalent code in a population-parallel system. On the other hand, data-parallel systems suffer the extra Central Processing Unit (CPU) time spend dynamically compiling in each generation. If the data-set is vast, then each compiled kernel is evaluated over very many test-cases and evaluation takes much longer than any CPU compilation. In that case, data-parallel's higher evaluation speeds make it more suitable; for smaller data sets, population parallel may be more effective.

It would help to find a middle ground. If the compilation time of data-parallel approaches be reduced, even if at the expense of the slightly decreased evaluation speeds, then problems with normal data set sizes could benefit from huge evaluation speeds. Compilers are typically complex and highly optimised so attempting to improve them would probably be unwise. The remaining possibility is to find ways to give the compiler less work to do. This research investigated finding and exploiting patterns in the code passed to the compiler.

2. TMBL

The form of Evolutionary Computation (EC) used for this paper is Tweaking Mutation Behaviour Learning (TMBL, pronounced "tumble"), which has been proposed as a baby sister to GP [4] and is akin to linear GP. The key feature of TMBL is its focus on long term fitness growth above all else. It is built on the following hypothesis: *long term fitness growth is dependent on the ease with which mutations can affect an individual's behaviour without (necessarily) ruining its existing functionality.* Such changes are known as tweaks.

An analogy helps motivate this hypothesis. Imagine that you are given around a hundred toy blocks with patterns on their surfaces so that lining them up in one particular way makes their patterns fit together. Imagine you are asked to solve the puzzle but only using trial and error: no preplanning, no writing, just considering random changes and performing them if they improve things.

Given this challenge, you would almost certainly take the puzzle, lay it out flat and solve it without much difficulty. Imagine you are then given an equivalent set of blocks but this time you must build the blocks vertically in a tower. This would be much harder. In fact, with around a hundred blocks, you might find it almost impossible. However much

progress is made, at some point it's necessary to grab some block near the bottom and ruin the prior achievements.

The argument is that the same principles hold for a GP tree flipped upside-down: at some point changes must be made to a node near the root of the tree and that ruins all the nodes above it. The lower blocks in the puzzle support the blocks above them physically; the lower nodes in the inverted GP tree support the nodes above functionally.

What went wrong when the tower became vertical? It became difficult to make changes to parts where progress had been made without damaging what had already been achieved. This view motivates the design of a representation for TMBL that is like a form of linear GP.

TMBL has high computational requirements and will typically be applied to problems with complex problems with moderately large data sets. With currently technology, these factors make it important to find the quickest possible methods of evaluating it. TMBL is focused on the long term fitness growth through a slow process of tweaks on previous successful solutions. The consequence of this is that its populations typically contain individuals that are highly similar to each other.

3. APPROACH

To exploit a GPU, one must write and compile a function for it. This is known as a kernel. The data-parallel approach involves writing such kernels each time a batch of individuals is to be evaluated. These kernels are then used for evaluation on the GPU. Several individuals may be grouped together into one kernel. The problem of this approach is the time spent compiling.

This research aimed to reduce the time spent dynamically compiling code for the GPU. The chosen approach targets the redundancy of compiling blocks of similar code by exploiting their similarities. Rather than sending code like that on the left of Table 1 to the compiler, an aligning algorithm is first used to identify the similarities and pull them together to form code like that on the right of Table 1.

The biggest danger with this approach is that the execution speed is reduced because the execution path through the kernel for each individual is more convoluted. Can the faster compilation outweigh the slower evaluation? If so, under what circumstances? The investigation sought to tackle these questions.

How should the individuals be aligned? It might be possible to identify these similarities by keeping track of the mutation and crossover operations performed on each individual. However that would be a rather complicated and brittle approach which would need extending with each new genetic operator. A more robust approach is to align individuals when they are to be evaluated.

3.1 Alignment

How can lists of instructions be aligned? There is a standard approach to aligning lists of items but it was not used in this context. To explain why, it is important to describe the standard approach first.

The principles underlying alignment algorithms are not dependent on the type of thing being aligned, so this paper talks about aligning lists made up of instructions, letters, items, amino acids and coloured shapes. The simplicity of letters makes them suitable for outlining the principles.

Consider two lists of letters, `DVSGGWIVHGVRGS` and `SGGWVHGRKGSA`. An alignment of these two lists involves laying them out such that some items from one list might tally with some items from the other. Hence an alignment is a list of alignment positions, each containing the next entry from one or more of the lists. For example, one possible alignment is as follows:

```
DVSGGWIVHGVRG.....S
  |  |  |||||
..S..G..GWVHGRKGSA
```

This is a poor quality alignment for two reasons. First, relatively few pairs of letters have been aligned. Second, many of the aligned pairs do not contain matching letters. This is permitted in other contexts but for this application, instructions may only be aligned with each other if they are identical. Under these criteria, the following alignment is an improvement:

```
DVSGGWIVHGVR.GS
  |||| ||| | ||
..SGGW.VHG.RKGSA
```

These alignments involve a single pair of lists. Later, it will be necessary to create multiple alignments with more lists. That algorithm will be described after the single alignment algorithm, upon which it is built.

How many possible ways are there to align a list containing m items with a list containing n items? That depends on whether different alignments that tally the same pairs of items should only be counted once. If not, the number of ways of performing the alignment may be calculated iteratively. If either of the lists is empty, there is only one possible way since there are no choices that need to be made. Otherwise, there are three recipes for aligning the lists:

Table 1: Reducing work for the compiler through alignment. On the left, each program's code is in a separate block. Since the compiler doesn't know these blocks are highly similar, it repeats work by compiling each separately. On the right, the similarities have been identified first so the common instructions are pulled together and need only be compiled once.

Unaligned

```
if (prog == 0) {
  ...
  slot1 += slot3;
  slot2 = 3.1096370;
  ...
}
else if (prog == 1) {
  ...
  slot1 += slot3;
  slot2 = 3.1096370;
  ...
}
else if (prog == 2) {
  ...
  slot1 += slot3;
  slot2 *= slot1;
  slot2 = 3.1096370;
  ...
}
```

Aligned

```
...
slot1 += slot3;
if (prog == 2) {
  slot2 *= slot1;
}
slot2 = 3.1096370;
...
```

- take the first item off the *first* list, add it to the end of the alignment and then align the remaining items in any possible way,
- take the first item off the *second* list, add it to the end of the alignment and then align the remaining items in any possible way or
- take the first item off *both* lists, tally them at the end of the alignment and then align the remaining items in any possible way.

The total number of ways is the sum of these possibilities and so can be calculated with the following iterative formula: $F(m,n) = F(m-1,n-1)+F(m,n-1)+F(m-1,n)$, where $F(0,i) = 1$ and $F(i,0) = 1$ for any i. This formula generates the result that there are around $2.054 * 10^{75}$ possible ways to align two lists of 100 items. Fortunately things are much better than this calculation suggests since the problem exhibits optimal substructure, ie optimal solutions can be calculated from optimal solutions to subproblems. This makes the problem amenable to standard dynamic programming techniques.

3.2 The Needleman and Wunsch Algorithm

Bioinformatics involves aligning amino acid sequences as a core technique and so has developed the area considerably. The basic algorithm used for this is the Needleman-Wunsch (NW) algorithm [5]. At the time of writing, the website of the Journal of Molecular Biology, which published the original NW paper [5], states it has received 2531 citations; Google Scholar estimates 5612. This is the standard alignment algorithm, against which others may be contrasted.

A scoring system is used to guide the algorithm and the standard scheme awards an alignment one point for tallying each pair of identical items. The NW algorithm is an application of dynamic programming to alignment. The principle of dynamic programming is to find the optimal solution to each subproblem and then reuse these results. This can be applied to alignment by using the following subproblem: align the two query sequences but with one or more items taken off the front of either or both. The optimal solution will contain the optimal solution to one of these subproblems. This observation can be applied iteratively to build up the optimal solution in simple steps.

For two sequences of lengths m and n, this process can be represented elegantly in a matrix of dimensions $m \times n$. Using conventional matrix indexing, the sequences both start in the top-left and finish in the bottom-right. Each possible alignment can be represented as a path through the matrix between these corners. Figures 1(a) and 1(b) can be viewed as such matrices.

3.3 The Need for a New Alignment Algorithm

What is the computational complexity of the NW algorithm with respect to the sizes of the sequences? For each cell, the original NW algorithm scans for the maximum values in the strips running down and right from the below-right cell. To examine the $(i-1)+(j-1)-1$ such cells for every position (i,j) (where $i > 1$ and $j > 1$) in an $m \times n$ matrix the number of examinations is:

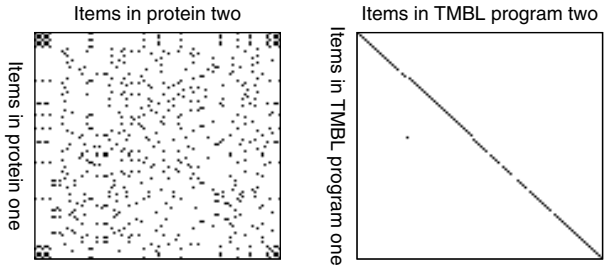
$$\sum_{i=2}^{m}\sum_{j=2}^{n}(i+j-3)$$

$$= \frac{1}{2}\left(m^2n + mn^2 - m^2 - n^2 - 4mn + 3m + 3n - 2\right)$$

This means the algorithm takes $O(n^3)$ time to align two sequences of length n and $O(n^2)$ time to compare a sequence of length n against a non-trivial, fixed-length sequence. Sankoff [7] later showed how to refine the algorithm to reuse more information and hence reduce running times to $O(n^2)$ time and $O(n)$ respectively. This is now widely referred to as the NW algorithm.

This means that NW is slower than would ideally be required here. Consideration of NW's requirements will show that they are much more stringent than those required here. NW alignments are used to provide a consistent measure of the level of sequence similarity between proteins and to identify stretches of sequences that are most similar. NW often faces very difficult problems and is relied upon to perform as good an alignment as possible.

In contrast, this problem only requires an alignment algorithm to face simple problems and to do a fairly good job quickly. Figure 1 helps to highlight the difference between the sorts of problems that might be faced.



(a) Aligning two protein sequences of length 100

(b) Aligning two TMBL programs of length 100

Figure 1: Aligning TMBL programs is very different to aligning protein sequences so NW may not be appropriate. A black pixel indicates a match between the two corresponding items such that they could be aligned. Subfigure 1(a) shows the problem of aligning two sequences of length 100 **from the proteins 2yuv and 2yuz. Subfigure 1(b) shows the problem of aligning two TMBL programs with** 100 **instructions. The two programs share the same single parent.**

Figure 1(a) shows the problem of aligning two protein sequences from the Protein Data Bank (PDB) files 2yuv and 2yuz. These sequences score 35% sequence identity over 97% overlap (using a gap penalty of 3), which provides good evidence of relatedness and suggests that this is a relatively easy alignment problem. A faint line can be discerned running from top left to bottom right, a likely rough path for the optimal alignment. However the signal for this path is weak and is difficult to discern due to the substantial noise of chance matches. Since each item is one of only 20 amino acids, we would expect such "false positive" matches in 5% of cases.

Contrast this with Figure 1(b) which shows the problem of aligning two 100 instruction TMBL programs. Here, there are very few items that match by chance since there are very many possible instructions. Furthermore the signal is

very strong because they have received few mutations since copying their shared parent's genome.

Note that there is another important difference: in this application two instructions may only be aligned together if they match whereas bioinformatics sequence alignments may include many aligned pairs that do not.

All of this motivates the design of a new, rough alignment algorithm.

3.4 A Rough Alignment Algorithm

The proposed algorithm's core idea is to just keep looking for the next matching pair. The algorithm is only ever interested in the next best step: it does not look far around for better, less direct routes and it never turns back from its current path. This narrowly focused approach is in contrast to the global approach of NW. For problems such as the one shown in Figure 1(a), the false positives could easily lead this algorithm astray through poor alignment routes. However for problems such as the one shown in Figure 1(b), the algorithm should rarely wander off track and should quickly find its way back to the main path if it does.

```
Stage One (neighbouring pair alignments)
 * Align each neighbouring list pair as follows
 * Start at the top-left corner of the matrix
 * Work diagonally down-right through matches
 * If the next down-right item is not a match,
search progressively further down and right from
the previous match
 * In searching, prefer:
    1st:  smaller maximum increments
    2nd:  more diagonal jumps
    3rd:  smaller first-list increments
 * Continue from the next match found
 * Repeat until there are no more matches to the
bottom right of the previous match
 * Glue the resulting alignments together

Stage Two (build extra connections)
 * Consider unconnected pairs where both items:
    - immediately precede two connected items,
    - immediately follow two connected items,
    - start their respective lists or
    - end their respective lists
 * Reject the pair if its items mismatch
 * Reject if connecting it would connect items
within a list
 * Reject if connecting it would form a cross
 * If the pair has not been rejected, connect it
 * Repeat until no more connections can be made
```

Figure 2: A summary of the two stages involved in the alignment algorithm.

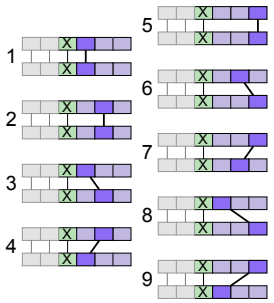
In more detail, the algorithm starts from just outside the top-left of the alignment matrix. It repeatedly looks for the next matching position, and moves there. When the algorithm can find no more matching pairs to the bottom-right of its current position, it stops. Non-matching items are never aligned and so can be ignored by the algorithm.

How does the algorithm choose the next matching position based on its current location? In short, it sweeps out and selects the first match it finds. The sweep is summarised as "Stage One" in Figure 2 and is depicted in Table 3(a).

The worst case for the algorithm is aligning two sequences with no matching items. This would require the described algorithm to search the entire matrix for the first match before giving up. Hence the algorithm aligns two sequences of length n in amortized $O(n^2)$ time, like the the NW algorithm. However the best case is aligning two identical sequences and the described would perform this in linear time whereas the NW algorithm would still take $O(n^2)$. Furthermore, this speed should degrade gracefully so that adding a few, small mutations should add little time to the alignment.

X					
	1	4	9	16	25
	3	2	7	14	23
	8	6	5	12	21
	15	13	11	10	19
	24	22	20	18	17

(a) First 17 search positions



(b) First nine as diagrams

Figure 3: The order in which the proposed alignment algorithm sweeps to find the next match. The X denotes the current location and the numbers indicate the sequence of the sweep.

3.5 A Rough Multiple Alignment Algorithm

This mechanism provides the means to align pairs of lists of items as illustrated in Figure 4(a) but this is only part of the problem of forming a multiple alignment. How should the code deal with aligning more than two lists? A globally optimum NW-based algorithm can align k sequences of length n using a k-dimensional matrix. That algorithm runs in $O(n^k)$ time (with respect to both n and k) which is unacceptable for all but the smallest of cases. The compromise often adopted in bioinformatics methods is to perform all-versus-all pairwise comparisons (typically alignments) and then use the results iteratively to identify the next most similar list and add it into a core (by performing further alignments). This has much better running time since it requires $\frac{k(k-1)}{2}$ alignments and so — when used in conjunction with NW— runs in $O(n^2k^2)$ time (with respect to n and k).

Even so, this problem demands a quicker, rougher approach. Rather than aligning all $\frac{k(k-1)}{2}$ pairs, the new algorithm aligns each of the $k-1$ pairs of neighbouring lists and then glues the resulting alignments together. As shown in Figure 4(b), this may leave some easy connections missed out. For instance, if one individual with one mutated item is placed in the middle of many otherwise identical lists, the mutated item will break the connection between identical items on either side.

Figure 5 shows an example of individuals being aligned after stage one and after stage two. Notice situations such as the one at the top of Figure 5(a): the first mutated instruction has broken the connection between its neighbours. In Figure 5(b), these two groups have been connected.

The proposed algorithm's second stage takes the aligned-

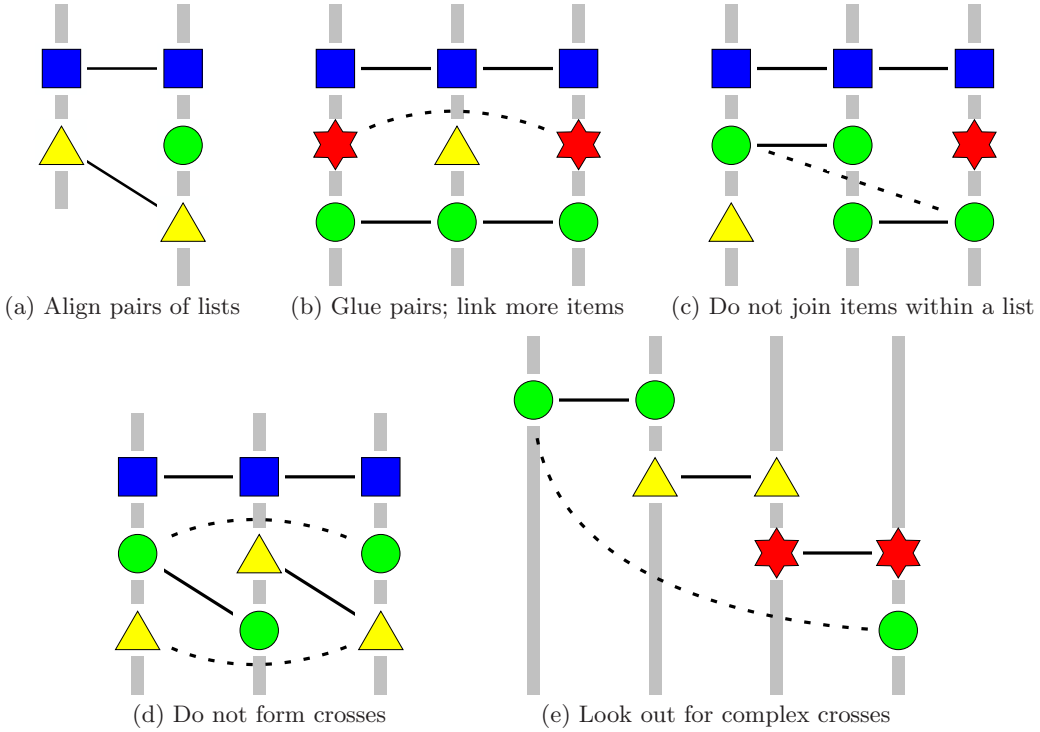


Figure 4: Grey strips represent lists of items, coloured shapes represent items being aligned, solid lines represent alignment links and dashed lines represent possible new links. 4(a) First, items are aligned within pairs of neighbouring lists. 4(b) Then these alignments are glued together and extra connections (eg the dashed line) may be formed. These links must not be formed if they connect items within a list or form crosses. 4(c) Joining items within a list (eg by forming the dashed line) breaks the alignment since it means one of those duplicates will be absent from the resulting code. 4(d) Forming a cross (eg by forming either of the dashed lines) breaks the alignment since then one joined group (eg joined circles) comes both before and after another (eg joined triangles). 4(e) Identifying some crosses (such as the one created by joining the third circle to the other two) may involve tracing back through lists that do not include any of the items to be joined (eg the list containing one triangle and one star).

and-glued neighbour-pair-lists from the first stage and scans for these extra connections. This avoids spending time on another alignment as is usual in bioinformatics multiple alignments. Since these connections are not formed in an alignment process, they are vulnerable to two new dangers, which are worth highlighting.

The first danger is connecting items such that two identical items within the same list are placed in the same group of equivalents. Figure 4(c) shows an example in which the proposed dashed link would result in the two green circles in the middle list being placed in the same group. Of course, this problem only occurs when a list contains two identical items. This is rare with this TMBL representation but must still be guarded against.

The second danger is forming a cross between two groups of equivalents so that each group contains some items before the other group and some items after it. Figure 4(d) shows an example in which either of the proposed dashed links would create a group that has items before another group and items after it. This would make the alignment invalid and so must be avoided. Figure 4(e) shows a more complicated cross example in which the proposed dashed link would create a group that has items before the triangles and the stars and an item after them. This example illustrates that some crosses may only be identified by tracing through relationships and through lists that do not include any of the items to be joined.

The scan for these extra connections proceeds by searching through the individuals' instructions. For each, a list of candidates instructions for connection is drawn up. To be a candidate for connection, a pair must be matching, not yet connected and must either both start or both precede their respective lists or either both immediately precede or both immediately follow two connected items.

Such items are checked for conflicts and if there would not be any, the connection is added. When any connection has been made between two items, the algorithm follows back up the pairs that immediately precede them in case they can be sequentially connected like two sides of a zip.

The algorithm checks for the two types of conflict described earlier: overlaps and crosses. The checks for cross are the most involved so these are only initiated when all other tests have been passed. The cross-checking subroutine checks for crosses from above the first item to below the second item. It is called with both orderings of the item so that it checks for crosses in either direction.

The cross-checker scans up the equivalences from the first item and searches for any routes that lead to something be-

(a) Stage one

(b) Stage two

Figure 5: Aligning 20 individuals, each with 80 instructions. Columns represent individuals, black marks represent individuals' instructions and rows represent aligned instructions. After stage one there are 183 positions; after stage two more connections have been identified and there are only 137 positions.

low the second item. It is important to ensure that any possible cross will be found but it is also important that no more time is spent checking for crosses than is necessary. To this end, the code works on the assumption that there are no pre-existing crosses (or other conflicts) in the alignment. This allows the code to terminate searches whenever it hits a "stop": an instruction beyond which a cross cannot exist if there are no pre-existing crosses. Instructions are stops if they are equivalent to an item preceding an equivalent of the second item or if they precede another stop. A simplified version of this is summarised as "Stage Two" in Figure 2.

4. EXPERIMENTS

The Compute Unified Device Architecture (CUDA) platform was used for the experiments. The techniqe is note dependent on the platform and might be applied in other GP compilation scenarios such as compiling code for execution on a multi-core CPU. The C++ alignment code was written as a template so it can align numbers, strings or TMBL instructions.

The experiments were concerned with the effects of the alignment on compilation time and evaluation speed. The results were verified against results from non-aligned code are not otherwise of interest.

The individuals in the experiments were generated as children of a single seed parent using a low mutation rate. Two points should be noted here. First, this creates groups of similar individuals which will favour the alignment technique. The aim was to provide a realistic environment. However in other systems, such as standard GP systems, the diversity may be much greater and this may make alignment technique worthless.

Second, the seed individual is evolved and so is likely to use most of its instructions (as has been found with TMBL) so the compiler will not optimise away many of the instructions. This might be very different if the seed individual were randomly initialised. To generate individuals with fewer instructions, the instructions were removed from the start of the seed individual. This may change the individual so that more of the instructions may be optimised away. Hence the recorded evaluation rates for low numbers of instructions may be an overestimate.

Operating system	Ubuntu Linux 10.10
Linux Kernel	2.6.35-28-generic-pae
GPU Device	nVidia GeForce GTX 260 [Core 216] (216 cores, core clock speed: 590MHz shader clock speed: 1296MHz)
CUDA toolkit	v3.2
Device driver	260.19.44
NVCC	v0.2.1221

Table 2: Details of the system

Number of CPU threads	1
Individuals per kernel	4
Number of individuals	120
Number of instructions	200
Number of evaluation repeats	8
Number of test-cases per evaluation	65536
Number of iterations per evaluation	50

Table 3: Default parameters for the runs

The system configuration is provided in Table 2 and the default parameters are provided in Table 3. The last three entries refer to how the evaluation speeds were assessed: this involved timing eight consecutive launches of kernels, each executing all individuals for 50 iterations over 65536 test-cases. For a standard population of 120 individuals, each with 200 TMBL instructions, this means executing 0.6291456×10^{12} TMBL instructions.

The mutation rate was set such that 95% of individuals have at least one mutation. For individuals with 200 instructions, this translates to a rate of 1.487% by instruction.

Each of the results in the experiments is averaged over five runs. Each line in the graphs in Section 5 has a background bar that indicates the mean value plus and minus one estimated standard error. In many cases, these bars are so thin that they cannot be seen. This suggests that the relatively small number of repetitions has been adequate to give good estimates of the means.

5. RESULTS

Figure 6 shows the time per individual to align and generate the CUDA C source. This indicates that the time spent on these tasks is very small and that the alignment actually reduces this time. This is presumably because it reduces the amount of code that must be output. This might also suggest that the code-outputting code would benefit from some

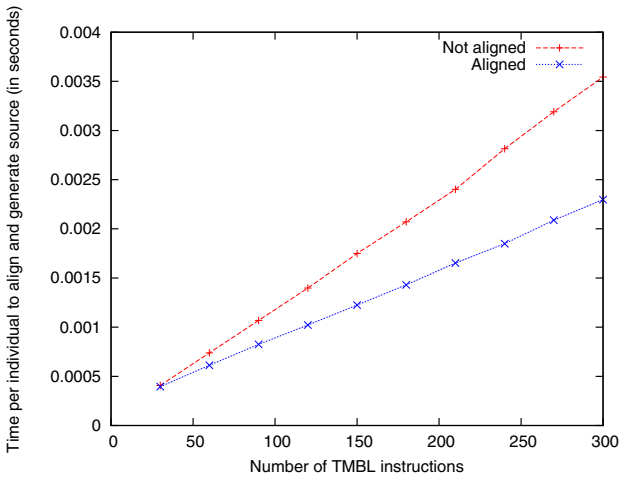


Figure 6: The time per individual to align and generate source over varying numbers of TMBL instructions

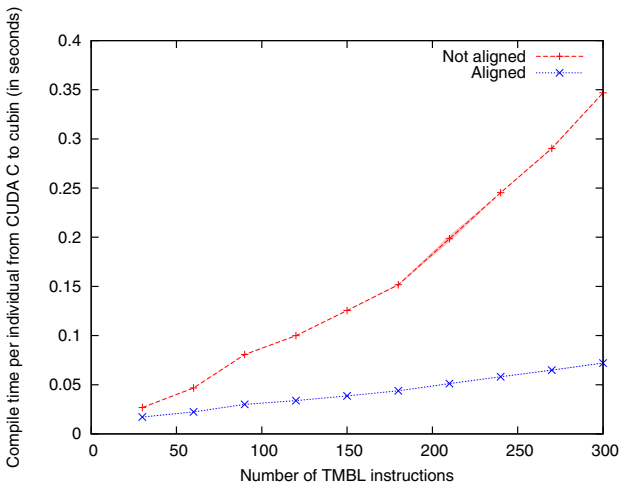


Figure 7: The time per individual to compile from CUDA C to cubin over varying numbers of TMBL instructions

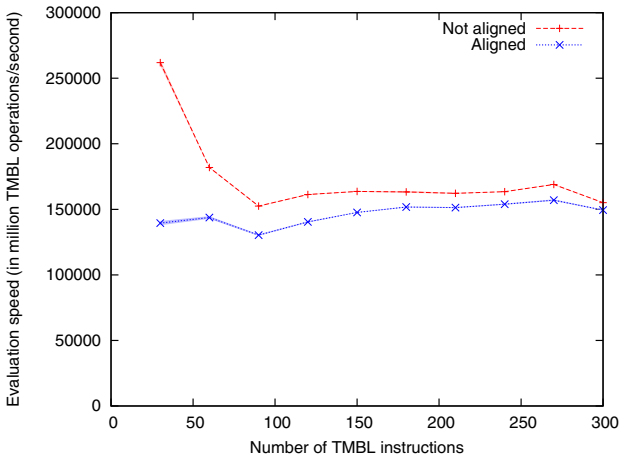


Figure 8: The evaluation speed over varying numbers of TMBL instructions

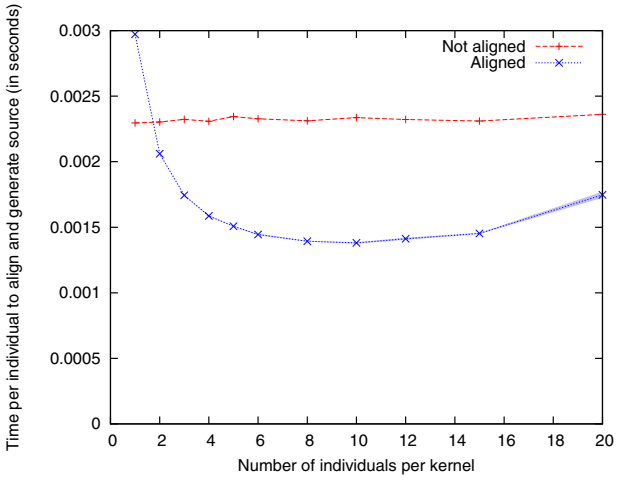


Figure 9: The time per individual to align and generate source over varying numbers of individuals per kernel

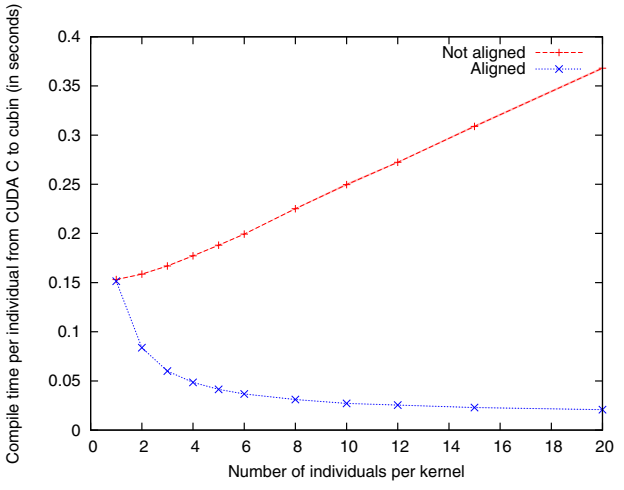


Figure 10: The time per individual to compile from CUDA C to cubin over varying numbers of individuals per kernel

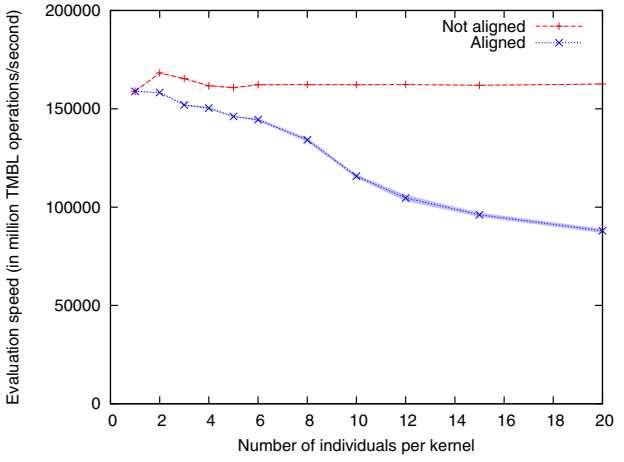


Figure 11: The evaluation speed over varying numbers of individuals per kernel

optimisation. Encouragingly, the graph seems to suggest that this time increases linearly with the number of TMBL instructions.

Figure 7 shows the time taken per individual to compile from CUDA C to a cubin binary file, ready to be loaded onto the GPU. These durations are much longer than those in Figure 6. The reduction in compile time achieved by alignment is more pronounced as the number of TMBL instructions increases. At 300 instructions, the compile time per individual is 0.347 seconds without alignment and 0.072 seconds with, a reduction of 79.238%.

Figure 8 shows the evaluation speeds and it shows the reduction in evaluation speed caused by alignment. The slowdown reduces as the number of TMBL instructions increases and is relatively small from 60 instructions. At 300 instructions the evaluation speed is 155052.633 million operations per second without alignment and 149383.227 million operations per second with, a decrease of only 3.656%.

Figures 9, 10 and 11 show these same properties over varying numbers of individuals in each kernel. Figure 9 shows that the alignment time remains small across these values.

Figure 10 shows that the reduction in compile time from alignment gets much larger as the number of individuals per kernel increases. Figure 11 shows that increasing the number of individuals per kernel also increases the reduction in evaluation speed.

Note that the lines meet at one individual per kernel in Figures 10 and 11 because at this point, there is no alignment work to be done so the input to the compiler remains the same.

6. CONCLUSION

Code was written to identify the similarities in TMBL kernels and unite them. The aim was to reduce the compilation time whilst keeping the evaluation speed comparable. Implementing this involves aligning the individuals against each other. The standard NW algorithm was examined but was found to be a more thorough algorithm than was required so a new algorithm was proposed that is rough but fast. This was extended with another rough but fast algorithm for forming multiple alignments.

Experiments showed that, at 300 instructions, the method reduced compilation time 4.817 times whilst only reducing evaluation speed by 3.656%. This satisfies the aim of making data-parallel evaluation speeds accessible for moderately sized data-sets. The amount of time spent aligning the individuals and generating the source code was relatively negligible and was even quicker when no alignment was being used. Increasing the number of instructions increased the reduction in compile time and decreased the reduction in evaluation speed. Increasing the number of programs per kernel increased the reduction in compile time but also increased the reduction in evaluation speed. These results suggests the following guidelines: use as many instructions as can be benefited from and then tune the number of individuals per kernel to fully load both the GPU and CPU.

The program conditions in the aligned source code only used if statements and only tested the individual's index with equality tests. Future work could tackle the evaluation speed reduction by adding else-if and else statements and by using greater-than and less-than tests.

The technique was applied to a TMBL representation but it could equally be applied to other forms of GP. There are two issues that need to be considered in judging the applicability: the representation of the individuals and the nature the population.

The representation is important in that it must allow similarities to be exploited to reduce duplication in the compiler's workload. In practice this should be possible for most forms, for example GP trees can be flattened into linear lists of instructions and these can then be aligned. This issue might not be the problem it initially appears.

More important is the nature of the population. The work here exploits the fact that in most TMBL generations, most individuals are mostly similar to each other. This suggests that the technique may be better suited to forms in which populations tend to contain many highly similar individuals. Where this is not true the technique is likely to be worthless.

7. REFERENCES

[1] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 7-11 July 2007. ACM Press.

[2] S. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on gpus. In *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[3] W. B. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 Mar. 2008. Springer.

[4] T. E. Lewis and G. D. Magoulas. Tweaking a tower of blocks leads to a TMBL: Pursuing long term fitness growth in program evolution. In *IEEE Congress on Evolutionary Computation (CEC 2010)*, pages 4465–4472, Barcelona, Spain, 18-23 July 2010. IEEE Press.

[5] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[6] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel GP on the G80 GPU. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 98–109, Naples, 26-28 Mar. 2008. Springer.

[7] D. Sankoff. Matching sequences under deletion-insertion constraints. *Proceedings of the Natural Academy of Sciences of the U.S.A.*, 69:4–6, 1972.