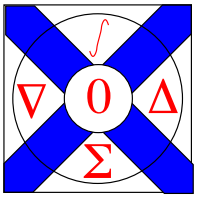

Computational Science Technical Note **CSTN-171**

Genetic Programming using the Karva Gene Expression Language on Graphical Processing Units

Alwyn V. Husselmann and K. A. Hawick

2013

Genetic Programming (GP) has been employed in many problem domains, and as a result, it has been the subject of much scientific inquiry. The extensive literature body of GP has reported applications in algorithm discovery, image enhancement and cooperative multi-agent systems, as well as many other areas and disciplines, such as agent-based modelling in Geography and Social Science. As models become more complex, further research toward higher efficiency have been warranted. We discuss solutions to large-scale systems which require automatic programming, and present results of a modified data-parallel implementation of GP based on Gene-expression Programming for Graphical Processing Units (GPUs), as well as a modified Santa Fe Ant Trail problem to measure the efficacy of this algorithm. We present results on algorithm convergence as well as timing performance on both GPU and CPU implementations.

Keywords: karva language; CUDA; genetic programming; gpu; parallel; optimisation

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-171,
        author = {Alwyn V. Husselmann and K. A. Hawick},
        title = {Genetic Programming using the Karva Gene Expression Language on Graphical
                Processing Units},
        booktitle = {Proc. 10th International Conference on Genetic and Evolutionary Methods
                (GEM'13)},
        year = {2013},
        number = {CSTN-171},
        pages = {GEM2456},
        month = {22-25 July},
        publisher = {WorldComp},
        institution = {Computer Science, Massey University, Auckland, New Zealand},
        keywords = {karva language; CUDA; genetic programming; gpu; parallel; optimisation},
        owner = {kahawick},
        timestamp = {2013.03.19}
}
```

Technical Report CSTN-171

Genetic Programming using the Karva Gene Expression Language on Graphical Processing Units

A.V. Husselmann and K.A. Hawick
Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
email: {a.v.husselmann, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

May 21, 2013

Abstract

Genetic Programming (GP) has been employed in many problem domains, and as a result, it has been the subject of much scientific inquiry. The extensive literature body of GP has reported applications in algorithm discovery, image enhancement and cooperative multi-agent systems, as well as many other areas and disciplines, such as agent-based modelling in Geography and Social Science. As models become more complex, further research toward higher efficiency have been warranted. We discuss solutions to large-scale systems which require automatic programming, and present results of a modified data-parallel implementation of GP based on Gene-expression Programming for Graphical Processing Units (GPUs), as well as a modified Santa Fe Ant Trail problem to measure the efficacy of this algorithm. We present results on algorithm convergence as well as timing performance on both GPU and CPU implementations.

Keywords: karva language; CUDA; genetic programming; gpu; parallel; optimisation.

1 Introduction

Genetic Programming (GP) was the combinatorial optimiser specially adapted for evolving programs by way of natural selection and evolutionary processes [35]. It was the result of John Koza's paradigm altering work of 1995 [20], the same year that saw parametric optimisation take a great leap forward with the advent of the particle swarm optimiser, due to Kennedy and Eberhart [18]. Combinatorial optimisation had been a problem under careful research for many years, with classic problems such as the Traveling Salesman Problem [38], Prisoner's Dilemma [1] and the Knapsack problem [10]. Many of these problems are representative of real-world applications, and advancements in solutions to these are beneficial to finding effective solutions for other problems. GP has been applied to intrusion detection [5], soccer playing "softbots" [24,25], models of land change in Mexico [27], team learning [26], algorithm discovery [2] and even image enhancement [36]. Cooperative multi-agent systems also benefit from the use of genetic programming [34], as well as classification for data mining [40].

Since the introduction of Genetic Programming in 1995, the canonical algorithm has been the subject of many enhancements and alterations, as is the case with related parametric optimisers [15, 17]. Some of these modifications include Linear GP [3] and Cartesian GP [30]. Some modifications are very extensive, such as more restrictive versions of GP [4] and also complete overhauls of the original representation of programs, such as that in Ferreira's Gene Expression Programming (GEP) [7]. Many of these modifications were intended to combat common problems in GP, which have precluded more extensive use. O'Neill et al describe these problems in detail [33]. The most prominent of which is code size; in which there exists a drift in the population towards larger and more redundant programs [37].

GP is a population-based optimiser, in which a set of programs are improved upon successively, by replacing it with a set of new programs, with genetic operators applied in a bid to drift away from bad solutions. A set of programs in this context is usually referred to as a generation. Each program represents a specific candidate solution to the problem at hand. Typically the most computationally expensive aspect of genetic algorithms in general, is the fitness evaluation of each of these candidate solutions. A fitness metric is necessary to guide the genetic operators in a way that is analogous to the concepts of natural selection and "survival of the fittest" in biology. The most common genetic operators considered are known as selection, mutation and crossover, where mutation perturbs a candidate solution in the search space, and crossover combines two candidate solutions to hopefully obtain a new single candidate with a slight improvement. Selection is simply a mechanism to obtain inputs for the crossover operator.

Gene Expression Programming (GEP) [7, 8] is one algorithm which departs from the traditional representation of programs in GP. It replaces the tree-based representation

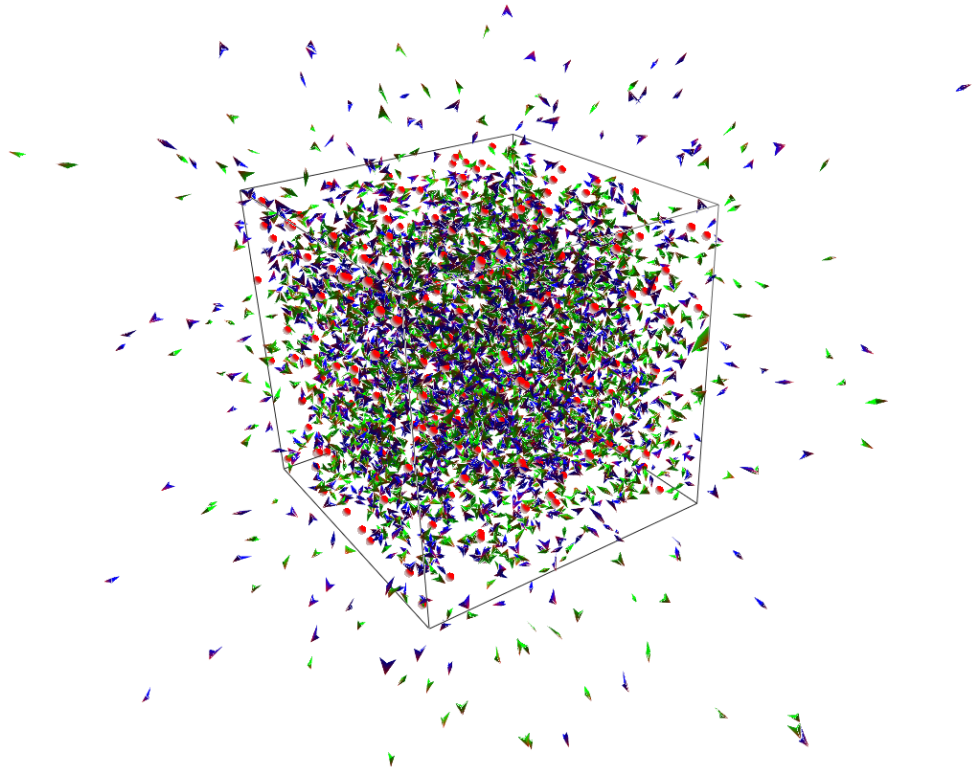

Figure 1: A 3D rendered version of the classic genetic programming test environment known as the Santa Fe Ant Trail. This visualisation [16] is a frame taken after random initialisation of the 256 agents. This representation casts light on the computationally expensive process of evaluating program individuals.

with a linear one, which still encodes an abstract syntax tree (AST). Perhaps the greatest advantage to GEP is its inherent support for introns: non-coding sections of the genotype. When the linear representation is interpreted to obtain the phenotype, certain introns may not make it into the tree. As well as being more representative of actual biological systems [30], this also brings a great simplification to mutation and crossover.

In this article we present a modified GP using the *k-expression* program representation from GEP, and accelerate this using Graphical Processing Units (GPUs). Our rationale for using GPUs are two-fold: commodity pricing on GPUs is very favourable for the computing power that can be exerted on these platforms, and second, the inherent parallelism in population-based optimisers such as the GP allow not only parallel fitness evaluation, but also parallel selection, crossover and mutation. Most other data-parallel implementations of GP focus on accelerating fitness evaluations [21]. We make a calculated effort to accelerate the genetic operators in our implementation. Other GPU-based GPs include efforts in acceleration on GPU clusters [12] and different representations [22]. Interestingly, there have also been parallelisation research inspired by quantum computing, most notably the evolution of CUDA (Compute Unified Device Architecture) PTX programs by a quantum-inspired linear GP [6].

Our article is structured as follows: In Section 2 we present some background on GEP, its operators, and various specifics on the algorithm. We also introduce the Compute Unified Device Architecture (CUDA) and the idiosyncrasies of efficient data-parallel simulations designed for this. In Section 3 we describe our modifications to GP and also the method by which we measure the efficacy of this algorithm. Following this, we use Section 4 to present our results and then discuss these in Section 5, then finally we conclude and present some possible future work in Section 6.

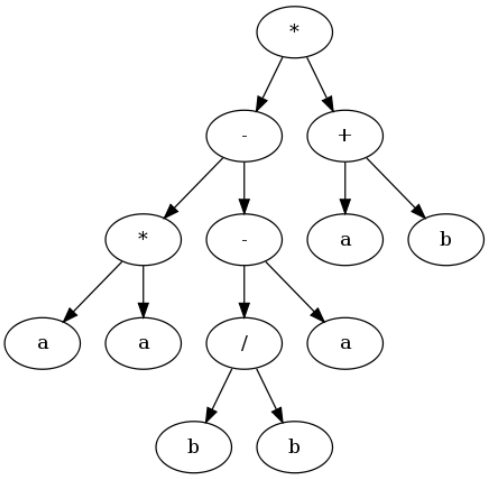

Figure 2: Phenotypic AST built from genotype represented by the *karva*-expression *-+*-abaa/abbacda.

2 Gene Expression Programming

We provide some background to gene-expression programming (GEP) and mechanisms for implementing it. Perhaps the most advantageous feature of Gene Expression Programming (GEP) is its representation of candidate programs [8]. Ferreira [7] designed a language named *Karva*, and candidate programs are represented in the form of *karva*-expressions, or sometimes shortened to *k*-expressions. These expressions take the following form:

```
01234567890123456
*-+*-abaa/abbacda
```

The first line is simply used as an indexing convenience. The second line is the genotype of the candidate solution to a particular problem. The distinction between *genotype* and *phenotypes* in GP in general is simply the interpretation of a representation into an executable program. This is not always necessary, especially in Linear GP, where candidates are stored in the same form as they are executed (namely, sequences of instructions executed successively).

The symbols used in the *k*-expressions are either terminals or non-terminals; and in this case, the terminals are `a,b,c` and `d`. In this example, all the non-terminals are self-explanatory and of arity 2, except `Q`, which is the square root function, of arity 1. GEP has support for any set of function terminals of any arity, provided that the expression length is long enough to give each function its required arguments. This will be made clear in how the genotype is interpreted into a phenotype. This *k*-expressionis shown in its phenotypic form in Figure 2.

The tree is built by reading the *k*-expressionfrom left to right and filling the arguments of the non-terminals in the tree level by level. Upon careful inspection, it is noteworthy that the terminal `d` is an intron, and hence not in the phenotype. Ferreira proposes several genetic operators and leaves the selection of these to the user and their specific application.

For ease of reference, we provide here a brief overview of the mutation and crossover operators. Our selection operator is tournament selection; as it most closely aligns with our needs in data-parallel computation. Details on this is provided in Section 3.

Crossover of two *k*-expressions is simple. We elect to use a one-point recombination (for its simplicity), which involves selecting a crossover point at random, and then creating two new candidates from the recombination about that point. Consider the *k*-expressionshown in 2. If it were recombined with another expression at index 5, such as:

```
01234567890123456
*-++baa/bbabbacda
```

Then the results would be the following:

```
01234567890123456
```

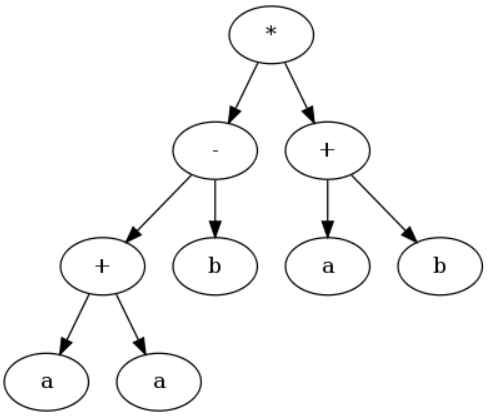

Figure 3: The AST built from the genotype represented by the *karva*-expression *-++babaa/abbacda.

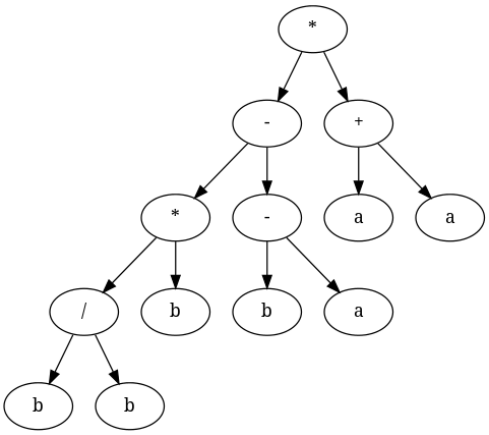

Figure 4: The phenotypic AST built from the genotype represented by the *karva*-expression *-+*-aa/bbabbacda.

```
*-++babaa/abbacda
*-+*-aa/bbabbacda
```

Which, when interpreted, are shown in Figs. 3 and 4.

Mutation of a *k*-expressionis simple. By using a uniform random number, a random symbol in the expression is chosen, and that symbol switched for another terminal or non-terminal.

The most popular method of selecting pairs of candidates to pass through the crossover operator is Roulette Selection [11]. Candidates are chosen at random throughout the population, but the probability of selecting each candidate is proportional to their fitness. This is usually accomplished by choosing a random number in $(0, 1)$ and mapping this to a hypothetical "roulette" wheel where the range of $(0, 1)$ is divided up in such a way that the probabilities of choosing each candidate are correct relatively.

The Roulette selection method does not parallelise extremely well. Most parallel GP algorithms make use of a selection method named "Tournament Selection" [29]. In this scheme, every candidate is compared against another uniform-randomly selected candidate, and the best candidate (by fitness) is chosen as one of a crossover pair. Once this process is complete, two candidates at a time

are recombined and then the mutation operator is called.

Genetic Programming responds very well to parallelisation. We use NVidia's Compute Unified Device Architecture (CUDA) platform [23, 32] to accelerate our algorithm. The CUDA platform arose from a very effective arrangement of MIMD and SIMD processors, which were intended for processing large numbers of pixel data as fast as possible. General-Purpose Graphical Processing Units (GPGPU) has gained much interest since the advent of CUDA, particularly in light of the fact that using pixel and fragment shaders for simulation is an arcane and difficult affair. CUDA makes this process much more accessible and purpose-built [31].

The CUDA-enabled GPU consists of several Streaming Multi-processors (SMs) which have a certain number of "CUDA cores". These SMs process work units known as "blocks", which represent a 1D, 2D or 3D grid of threads. These blocks are sized by the user, and typically coincide with simulation-specific requirements. An SM computes a block until completion, and then, if available, carries on to the next block. During execution, threads are divided into groups of 16, known as "warps". Warps are the smallest unit of execution in CUDA. Warps are executed in a SIMD fashion on the CUDA cores on each SM (sometimes known as SIMT). The combination of all SMs are therefore MIMD.

CUDA-enabled GPUs have some idiosyncratic behaviour including memory access penalties and scoping among others. These can sometimes be problematic when not given careful due consideration. CUDA provides a variety of memories to the user, each of which has a different access penalty and scope. For brevity, we omit a thorough discussion of these. The process of executing simulations with CUDA involves copying data across the PCI bus to the GPU's global memory, where it is then manipulated by device-specific code. Once this computation is complete on the GPU, the program would copy the modified data back again. Depending on applicability in the application, one can make use of host page-locked memory which reduce expensive memory copies.

GPU-specific instructions are created by making use of special syntax added to the C language, which is compiled by the `nvcc` compiler. Once this code has been compiled, the rest of the program is passed to the system C compiler for normal compiling. This allows the CUDA device drivers to copy device specific instructions to the GPU for computing.

3 CUDA Implementation

Apart from our use of The *Karva* language for expressing candidate solutions, we also make special effort in order to accelerate the process by which a new generation is computed. It should be noted that sometimes the fitness function can greatly outweigh this computation.

It appears then that k-expressions are naturally well-suited to being used in CUDA. This is because they can be stored as sequences of (independent) characters or integers. Furthermore, crossover and mutation are almost trivially easy, bearing in mind the head and tail requirements. Our method for combining the traditional GP algorithm and GEP-style k-expressions is shown in Alg. 1.

Algorithm 1 The parallel implementation of GP on GEP k-expressions.

```
allocate & initialise space for n candidate programs

allocate space for random deviates
while termination criteria not met do
  call CURAND to fill the random number array with
  uniform deviates in the range [0,1)

  copy candidates and candidate bests to device
  CUDA: compute_argument_maps()
  CUDA: interpret/execute programs
  CUDA: update food locations/fitness
  copy back to host

  if end-of-generation then then
    CUDA: apply genetic operators to programs
    replace old programs with new ones
  end if
  visualise the result
end while
```

In this algorithm, we parallelise the majority of computations. A disadvantage to using k-expressions is that interpretation of these programs are not straight-forward. We compute so-called "argument maps" in order to allow the CUDA-based interpreter we designed to directly fill the instructions with their corresponding arguments without using recursion.

Precisely the method by which we apply the genetic operators is strongly dependent on the selection method. As we have mentioned before, Tournament selection is the method of choice for most parallel implementations. The algorithm we use for accomplishing this in CUDA is shown in Alg. 2.

Parameters for the CUDA-based algorithm are $P(\text{mutate}) = 0.1$ and $P(\text{crossover}) = 0.8$.

To facilitate comparison, we have implemented a single-threaded CPU-based GP optimiser, with the exact same objective function, but computed in a serial fashion. Parameters are similar to the CUDA-based modified GP, with $P(\text{crossover}) = 0.8$ and $P(\text{mutation}) = 0.01$. The CPU-based GP makes use of the canonical tree-based representation with a depth restriction of 4. The same number of agents were used (1024). Tournament selection is also used, and initialisation/point mutation is done by the Full method. Crossover is implemented as a subtree swap. To avoid program bloat, we prune the trees following the genetic operators to a maximum depth of 4, where leaves are replaced by random terminal symbols.

Algorithm 2 Parallel Tournament Selection

launch a CUDA kernel with $n/2$ threads
assign thread candidate $x/2+1$ and $x/2$
set a to random index
if candidate a beats candidate $x/2+1$ **then**
  replace candidate $x/2+1$ with a
end if

set b to random index
if candidate b beats candidate $x/2$ **then**
  replace candidate $x/2$ with b
end if

recombine candidates $x/2$ and $x/2+1$
mutate the two resultant candidates
save results over the original two candidates

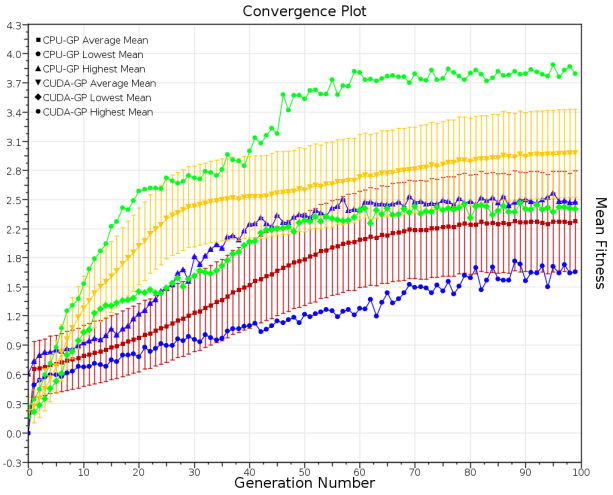

Figure 5: Convergence results for the CUDA-based GP with k-expressions and the CPU-based GP with the canonical tree-based representation. The graph shows the average mean value of each generation, from 100 independent runs. The error bars represent the average standard deviation of the 100 runs in each generation. Lowest and highest population means are also shown.

4 Convergence & Performance Results

We present some convergence data showing how the various algorithmic implementations behave as well as some timing performance data for a GPU/CUDA implementation compared with a conventional serial CPU implementation.

Convergence results for the CUDA-based GP (based on k-expressions) as well as the CPU-based GP with canonical representation is shown in Fig. 5. The plot shows the average mean values of each generation for both algorithms. Each of these data points has been averaged across 100 independent runs. The error bars on the average mean line represents the average standard deviation of the population fitness across all 100 separate runs.

	Frame Time (μ s)	Gen. Time (μ s)
CUDA-GP (k-exp)	1160 ± 40	423.2 ± 0.5
CPU-GP	48000 ± 8300	2600 ± 300

Table 1: Performance data for the CPU-based canonical GP and the k-expression GPU-based GP algorithm.

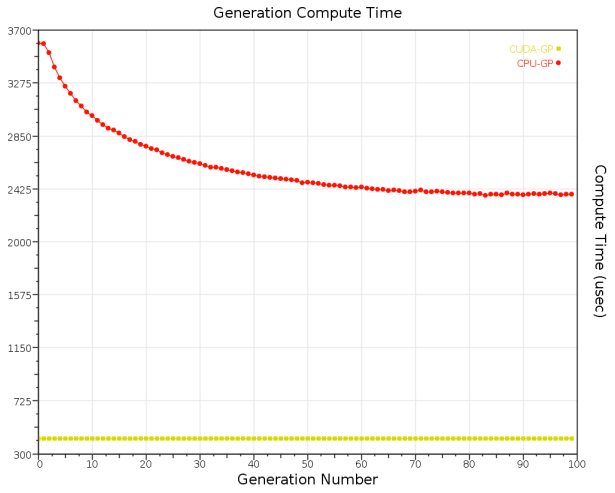

Figure 6: Wall-clock performance of the CUDA and CPU-based algorithms by generation.

It is clear from this graph that the CUDA-based GP with k-expressions clearly outperform the standard GP. At around generation 20, the CUDA-GP seems to have a larger spread in the average mean. We believe that this may be indicative of a local minimum, analogous to the same phenomenon in parametric optimisation. It is interesting to note that, because of the difference in representation between the CUDA-GP and the standard GP, at generation 20, the standard GP shows the same increase in spread of mean, albeit, less pronounced.

The spread of the mean towards the end of the simulation is smaller for the standard GP than for the CUDA-GP. Although a smaller spread is much more desirable, the highest mean of the standard GP only barely surpasses the score of the lowest mean in the CUDA-based GP.

Performance data for each of these algorithms were also collected. These are shown in Tab. 1. We averaged the frame compute time across the 300 frames in each generation, and then across all 100 independent runs. The generation compute time represents the time it took the algorithms to compute a new population only. This was also averaged over the 100 separate runs. The data is of the form mean $\pm$ std. dev.

From this data, the CUDA-based algorithm achieves a speedup of 6 times over the CPU algorithm for computing new populations, and 41 times over the CPU algorithm for computing a single frame of the simulation.

Fig. 6 shows the time taken by each algorithm for computing a new population for fitness evaluation. This process is mostly just the genetic operators; selection, crossover

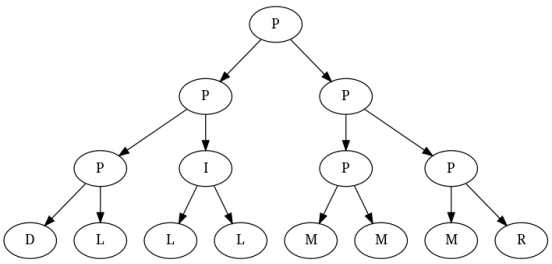

Figure 7: A typical individual agent generated by the Full method in our simulation. LISP-style code for this tree is (P(P(P(D)(L))(I(L)(L)))(P(P(M)(M))(P(M)(R)))).

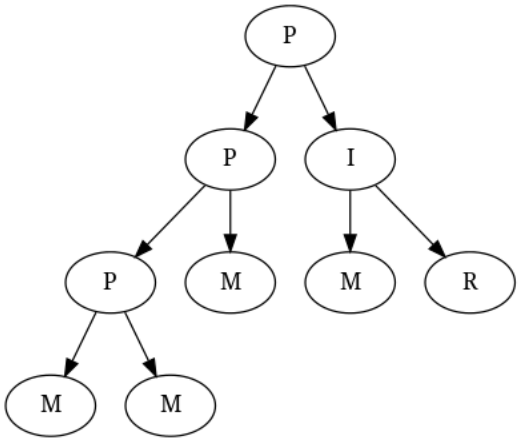

Figure 8: A highly effective generated agent. The LISP-style code for this is (P(P(P(M)(M))(M))(I(M)(R))).

and mutation. It is interesting to note that the compute time for the CPU-based standard GP is nonlinear, while the CUDA-based GP (with GEP-style k-expressions) is practically linear. This is not an artifact of the plot itself. The mean generation compute time for the CUDA algorithm has a standard deviation of just 40μ sec, whereas the CPU algorithm has a much larger 8300μ sec, even taking into account the fact that its frame time is 41 times larger.

We believe that this may be due to the initialisation method (Full) of the GP algorithm. Fig. 7 shows what form a typical initialised agent would take. This is in sharp contrast with Fig. 8 which depicts a much more effective solution. As can be seen from the initial program, they can potentially contain more `IfFoodAhead` functions, which are far more computationally expensive. This explains why initialised programs are often more expensive to evaluate, and hence increase the overall generation compute time for early generations.

From observation, it seems that highly effective programs generally take a certain form. An `IfFoodAhead` function is used to cause movement in a direction other than straight, and the rest of the program is simply `Move` terminals. The reason why more `Move` terminals are beneficial is because it allows the agent to move faster, and hence potentially reach more food.

5 Discussion

By using the *Karva* language from within Ferreira's Gene Expression Programming algorithm, we believe we may have attained better convergence for a particular reason; crossover with k-expressions is a lossless matter. During mutation in the standard GP, code bloat occurs, which is generally remedied by using a pruning method, which forces trees to be of a certain maximum depth. This is accomplished by simply replacing functions with terminals which are just above the maximum depth. The trees are also pruned after crossover, since subtrees are chosen at random, and destination nodes are also chosen at random. In other words, one tree can be virtually appended to another, causing a program that is twice the size of the original. Every time pruning is applied, potentially valuable information is lost. By using k-expressions, we avoid this problem.

It is worth noting that, as with most metaheuristics, the parameter tuning effort is vitally important. Algorithms such as these are very sensitive to their parameters. Our tuning effort was carried out by hand, and as a result, it could be improved somewhat. From the results we obtained however, we believe we could not improve the standard GP even to the point where the result would at all be inconclusive.

The speedup obtained from a data-parallel implementation is also very promising. There are some optimisations we have not included in our CUDA-based algorithm, such as interaction redundancy elimination using spatial partitioning, as well as hiding memory latency with tiling. Even without these techniques, it is clear that a GPU implementation presents a substantial improvement over CPU-based GP, which will also generally extend to Evolutionary Algorithms (EAs).

The notion of evolving a grammar is a powerful one providing there is sufficient expressivity in the grammar itself to accommodate appropriate features. Programming grammars and formats such as Extended Backus-Naur Form (EBNF) and Backus Naur Form (BNF) and their various derivatives [19, 39] may offer some interesting possibilities if appropriate genetic operators can be applied to language evolution. Although there is work reported in the literature on parsers that learn a language based on examples [28] in (E)BNF the notion of evolving languages themselves based on such a representation does not yet seem to have been explored. Given the recent interest in the literature on domain-specific languages (DSLs) [9, 13, 14] and their use for reducing code complexity in a wide range of applications, there is scope for applying the techniques we have discussed to DSLs expressed in an appropriate grammar that can be subsequently evolved and investigated for fitness metrics such as compactness.

6 Conclusions and Future Work

We have presented a CUDA-based Genetic Programming algorithm using the *Karva* language from Gene Expression Programming for program representation. We have characterised and compared this algorithm against the canonical CPU-based Genetic Programming algorithm both with the same modified Santa Fe Ant Trail objective function.

Our results suggest that using *Karva* provides a great benefit towards convergence aspects of the algorithm, as well as towards improving the wall-clock performance. We have also discussed our method of selection, as well as crossover and mutation in the context of k-expressions. The resultant programs we obtained from both the CPU and CUDA-based were competitive with hand-tuned programs for the given restrictions. The best wall-clock performance was achieved by the CUDA-based algorithm, and we also discussed some irregularities in the CPU performance data for population computations.

There is scope for further work towards extending or improving Grammatical Evolution in a similar manner, especially taking into account its highly desirable attribute in making use of BNF grammars. We also belive it possible to develop tools that will support visualisation of this sort of program space in a more appropriate fashion.

References

[1] Axelrod, R.: The emergence of cooperation among egoists. The American Political Science Review 75, 306–318 (1981)

[2] van Berkel, S.: Automatic Discovery of Distributed Algorithms for Large-Scale Systems. Master's thesis, Delft University of Technology (2012)

[3] Brameier, M.: On Linear Genetic Programming. Ph.D. thesis, University of Dortmund (2004)

[4] Castle, T., Johnson, C.G.: Evolving high-level imperative program trees with strongly formed genetic programming. In: Proceedings of the 15th European Conference on Genetic Programming, EuroGP. vol. 7244, pp. 1–12. Springer (April 2012)

[5] Crosbie, M., Spafford, E.H.: Applying genetic programming to intrusion detection. Tech. rep., Department of Computer Sciences, Purdue University, West Lafayette (1995), aAAI Technical Report FS-95-01

[6] Cupertino, L., Bentes, C.: Evolving cuda ptx programs by quantum inspired linear genetic programming. In: Proceedings of GECCO'11 (2011)

[7] Ferreira, C.: Gene expression programming: A new adaptive algorithm for solving problems. Complex Systems 13(2), 87–129 (2001)

[8] Ferreira, C.: Gene Expression Programming - Mathematical Modeling by an Artificial Intelligence. Springer, 2nd edn. (2006), iSBN 3540327976

[9] Fowler, M.: Domain-Specific Languages. No. ISBN 0-321-71294-3, Addison Wesley (2011)

[10] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman (1979)

[11] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989), iSBN 0201157675

[12] Harding, S.L., Banzhaf, W.: Distributed genetic programming on gpus using cuda, submitted to Genetic Programming and Evolvable Machines, 2009

[13] Hawick, K.A.: Engineering domain-specific languages for computational simulations of complex systems. In: Proc. Int. Conf. on Software Engineering and Applications (SEA2011). pp. 222–229. No. CSTN-123, IASTED, Dallas, USA (14-16 December 2011)

[14] Hawick, K.A.: Fluent interfaces and domain-specific languages for graph generation and network analysis calculations. In: Proc. Int. Conf. on Software Engineering (SE'13). IASTED, Innsbruck, Austria (11-13 February 2013)

[15] Husselmann, A.V., Hawick, K.A.: Levy flights for particle swarm optimisation algorithms on graphical processing units. Parallel and Cloud Computing 2(2), 32–40 (2013), `http://pcc.vkingpub.com/Download.aspx?ID=24`, submitted to J. Parallel and Cloud Computing

[16] Husselmann, A.V., Hawick, K.: 3d vector-field data processing and visualisation on graphical processing units. In: Proc. Int. Conf. Signal and Image Processing (SIP 2012). pp. 92–98. No. CSTN-140, IASTED, Honolulu, USA (20-22 August 2012)

[17] Husselmann, A.V., Hawick, K.A.: Parallel parametric optimisation with firefly algorithms on graphical processing units. In: Proc. Int. Conf. on Genetic and Evolutionary Methods (GEM'12). pp. 77–83. No. CSTN-141, CSREA, Las Vegas, USA (16-19 July 2012)

[18] Kennedy, Eberhart: Particle swarm optimization. Proc. IEEE Int. Conf. on Neural Networks 4, 1942–1948 (1995)

[19] Knuth, D.E.: Backus normal form vs. Backus Naur form. Commun. ACM 7(12), 735–736 (Dec 1964), `http://doi.acm.org/10.1145/355588.365140`

[20] Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Statistics and Computing 4(2), 87–112 (June 1994)

[21] Langdon, W.B.: A many-threaded cuda interpreter for genetic programming. In: Esparcia-Alcazar, A.I., Ekart, A., Silva, S., Dignum, S., Uyar, A.S. (eds.) Proceedings of the 13th European Conference on Genetic Programming, EuroGP. pp. 146–158. Springer (April 2010)

[22] Langdon, W.B., Banzhaf, W.: A simd interpreter for genetic programming on gpu graphics cards. In: O'Neill, M., Vanneschi, L., Esparcia, A., Gustafson, S. (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP (March 2008)

[23] Leist, A., Playne, D.P., Hawick, K.A.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience 21(18), 2400–2437 (25 December 2009), CSTN-065

[24] Luke, S.: Genetic programming produced competitive soccer softbot teams for robocup97. In: Koza, J.R., Banzhaf, W., Chellapilla, K., Kumar, D., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Goldberg, D., Iba, H., Riolo, R. (eds.) Genetic Programming 1998: Proceedings of the 3rd annual conference. pp. 214–222. Morgan Kaufmann, San Mateo, California (1998)

[25] Luke, S., Hohn, C., Farris, J., Jackson, G., Hendler, J.: Co-evolving soccer softbot team coordination with genetic programming. Robocup-97: Robot soccer world cup I 1,

398–411 (1998)
[26] Luke, S., Spector, L.: Evolving teamwork and coordination with genetic programming. In: Proceedings of the First Annual Conference on Genetic Programming. pp. 150–156. MIT Press (1996)
[27] Manson, S.M.: Agent-based modeling and genetic programming for modeling land change in the southern yucatán peninsular region of mexico. Agriculture Ecosystems & Environment 111, 47–62 (2005)
[28] Mernik, M., Gerlic, G., Zumer, V., Bryant, B.R.: Can a parser be generated from examples? In: Proceedings of the 2003 ACM symposium on Applied computing. pp. 1063–1067. SAC '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/952532.952740
[29] Miller, B.L., Miller, B.L., Goldberg, D.E., Goldberg, D.E.: Genetic algorithms, tournament selection, and the effects of noise. Complex Systems 9, 193–212 (1995)
[30] Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation 10(2), 167–174 (2006)
[31] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue 6(2), 40–53 (March/April 2008)
[32] NVIDIA® Corporation: NVIDIA CUDA C Programming Guide Version 4.1 (2011), http://www.nvidia.com/ (last accessed April 2012)
[33] O'Neill, M., Vanneschi, L., Gustafson, S., Banzhaf, W.: Open issues in genetic programming. Genetic Programming and Evolvable Machines 11, 339–363 (2010)
[34] Panait, Luke: Cooperative multi-agent learning: The state of the art. Autonomous Agents and Multi-Agent Systems 11, 387–434 (2005)
[35] Poli, R., Langdon, W., McPhee, N.: A field guide to genetic programming. lulu.com (2008), http://www.gp-field-guide.org.uk
[36] Poli, R., Cagnoni, S.: Genetic programming with user-driven selection: Experiments on the evolution of algorithms for image enhancement. In: Genetic Programming 1997: Proceedings of the 2nd Annual Conference. pp. 269–277. Morgan Kaufmann (1997)
[37] Soule, T., Foster, J.A.: Code size and depth flows in genetic programming. In: Koza, J., Deb, K., , Dorigo, M., Fogel, D., Garzon, M., Iba, H. (eds.) Proceedings of the 2nd Annual Conference on Genetic Programming. pp. 313–320. MIT Press (1997)
[38] Wilson, G.V., Pawley, G.S.: On the stability of the travelling salesman problem algorithm of Hopfield and Tank. Biol. Cybern. 58, 63–70 (1988)
[39] Wirth, N.: What can we do about the unnecessary diversity of notation for syntatic definitions. Communications of the ACM 20(11), 822–823 (November 1977)
[40] Zhou, C., Xiao, W., Tirpak, T.M., Nelson, P.C.: Evolving accurate and compact classification rules with gene expression programming. IEEE Transactions on Evolutionary Computation 7, 519–531 (December 2003)