Geometric Optimisation using Karva for Graphical Processing Units

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

Abstract— *Population-based evolutionary algorithms continue to play an important role in artifically intelligent systems, but can not always easily use parallel computation. We have combined a geometric (any-space) particle swarm optimisation algorithm with use of Ferreira's Karva language of gene expression programming to produce a hybrid that can accelerate the genetic operators and which can rapidly attain a good solution. We show how Graphical Processing Units (GPUs) can be exploited for this. While the geometric particle swarm optimiser is not markedly faster that genetic programming, we show it does attain good solutions faster, which is important for the problems discussed when the fitness function is inordinately expensive to compute.*

Keywords: CUDA; geometric; genetic programming; gpu; parallel; particle swarm

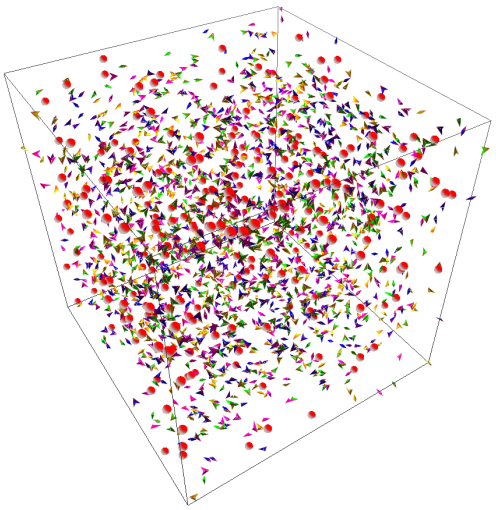

Fig. 1: A 3D version of the classic Genetic Programming test environment named the Santa Fe Ant Trail. This is a visualisation of 2048 "ants" in their competitive coevolution in gathering the red food items.

1. Introduction

The advent of combinatorial optimisers saw the birth of genetic programming (GP) [17], [30], which is the term now widely representative of all algorithms intended to optimise in the search space of programs. John Koza first invented GP in 1995 alongside the pioneering work of Kennedy and Eberhart towards the Particle Swarm Optimiser (PSO) [15], [3], [16]. Combinatorial and parametric optimisers have largely evolved in tandem over time, and a great deal of research effort has been expended in improving them. These have resulted in a great many varieties of parametric optimisers, including the Cooperative PSO of van den Bergh [38], space exploration advancements [13] and notably, data-parallel optimisers [9]. Combinatorial optimisation in the space of programs have also gained Cartesian GP [23], Gene Expression Programming (GEP) [4] and a handful of others. The intentions behind these algorithms are usually to improve upon one or more aspects of the class of optimiser in question. GP has been applied to a variety of problem domains including intrusion detection [2], robotics [21], [20], geography [22], algorithm discovery [37], image enhancement [31], as well as data mining [40] and cooperative mult-agent systems [29].

Genetic Programming (GP) is an optimiser intended to successively evolve a generation of candidate solution programs to solve a particular problem. John Koza's work toward GP [17] was arguably inspired by the pioneering work of John Holland [6] earlier in the 20th centuary on the canonical genetic algorithm. Most variants of the original algorithm today make use of various representations, operators and other techniques such as elitism and alternative selection mechanisms. GP maintains a population of candidate solutions and computes a new population of individuals after some genetic operators have been executed based on relative fitness evaluations of these individuals. These operators are most commonly crossover, mutation and selection. These are analogous to biological processes, which was the primary source of inspiration behind *them*.

One modification worthy of note is Ferreira's Gene Expression Programming (GEP) [4] algorithm. A common problem with GP-based algorithms is in choosing a suitable representation for candidate solutions. The original made use primarily of abstract syntax trees (ASTs) in pointer-tree storage and directly for evolution. Linear GP algorithms [1] store their programs as linear strings of instructions, executed one after the other. This unfortunately suffers from

the Halting Problem [28]. GEP represents individuals as a string of symbols (codons) but this string is known as a genotype, since it must be interpreted to obtain a tree-based phenotype. This has a number of advantages, which are discussed in more detail in Section 2.

Data-parallelism on commodity-priced hardware such as Graphical Processing Units (GPUs) have gained much interest in the past few years for their relatively inexpensive and formidable processing power [19]. The concept of executing Evolutionary Algorithms (EAs) on GPUs is not new however; many modified algorithms (particularly EAs) have been proposed for a great variety of problems, but most of these focus on parallelising the fitness evaluation process [18]. This is commonly the most computationally expensive and time consuming aspect of EAs.

Geometrically unified EAs have recently become a field of interest due to Moraglio and colleagues [24], [27]. It has resulted in the generalisation of a handful of EAs to arbitrary search spaces. The only caveat of these *geometric* algorithms is that the user must be able to provide geometric operators suitable for the search space under consideration. Geometric Differential Evolution [26], [27] and Geometric Particle Swarm Optimisation [25] have already been proposed with varying success. While generalisation of these excellent parametric optimisers incite keen interest, their efficacy in comparison to Genetic Programming and other combinatorial optimisers have not been accurately determined yet [33].

In this article we seek to explore the dynamics of the Geometric Particle Swarm Optimiser (GPSO) using a modified representation named *k*-expressions (short for *karva*-expressions) from Ferreira's Gene Expression Programming (GEP) algorithm with modified genetic operators to suit this representation. We then accelerate this algorithm using NVidia CUDA-enabled GPUs. NVidia's Compute Unified Device Architecture (CUDA) allows access to an effective and highly efficient means of utilising data-parallelism to many algorithms including agent-based modelling [8], [14] and other situated agent parallelism [11] problems as well as optimisation [13].

Section 2 contains more details on the GEP *k*-expression representation, including the relevant genetic operators introduced by Ferreira, as well as a brief introduction on CUDA and GPU-based EAs. We also present a brief summary of the GPSO. Following this, in Section 3 we describe our algorithm and the combination of GEP and GPSO as well as the modified genetic operators we use. We then present some performance and convergence results in Section 5 and compare our results with a Genetic Programming algorithm. Finally, we discuss our results and conclude in Sections 5 and 6 respectively.

2. Background

As mentioned before, the canonical GP maintains a population of candidate solutions. In this work, we have elected to use the *k*-expression representation for the elegant simplicity it affords to genetic operators, and its inherent support for introns, or non-coding sections. The linear nature of this representation is also very desirable for parallelisation, even taking into account that an interpreter would still be necessary.

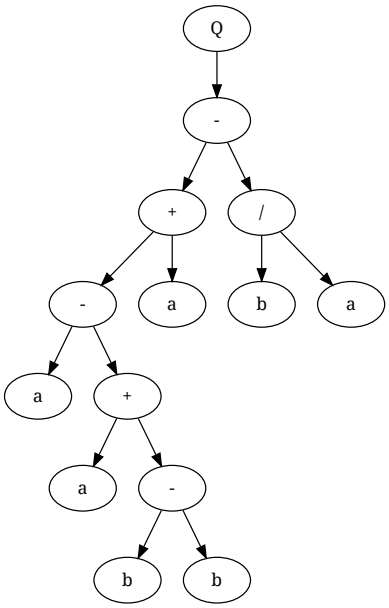

Fig. 2: The phenotypic AST built from the genotype represented by the *karva*-expression Q-+/-abaa+a-bbacda.

K-expressions are arranged in a string of fixed length, divided into a head section and a tail section. Function symbols can be of any arity, but can only appear in the head section of the expression. This serves the purpose of ensuring that the genotype is always interpreted into a syntactically correct phenotype tree. Terminal symbols may appear in both the head or tail sections. An example *k*-expression is shown below. When interpreted, this expression results in the tree shown in Fig. 2. The tree is built by reading the expression from left-to-right and filling in the arguments of each codon, level by level.

```
01234567890123456 7
Q-+/-abaa+a-bbacda
```

It is noteworthy that the terminal symbol `d` does not appear in the phenotype. This is the mechanism by which introns are supported. Once all arguments have been supplied in the tree, the rest of the expression is simply ignored, but not discarded [4]. A mutation in the tail section which swaps a terminal for a non-terminal could easily reactivate these ignored codons.

We now provide a brief overview of the original PSO by Kennedy and Eberhart. The PSO is characteristically known for maintaining a constantly updated global best solution, as well as a constantly updated personal best

solution for each candidate. The behaviour of the algorithm in geometrically moving a candidate through space, biased toward the personal best or global best is reminiscent of crossover and mutation behaviour, as it also contains a stochastic component. This random aspect of the algorithm has also been subject to improvements, as it is the main space exploration mechanism; without which, there could only be genetic drift [12]. Eqns. 1 and 2 show the recurrence relations which underlie the inertial PSO [35].

The stochastic component is introduced by the ephemeral random constants r_p and r_g (typically uniform random deviates in (0,1)), and user-defined constants ω, ϕ_p and ϕ_g. ω is known as the inertial constant, whereas the latter two constants determine a constructive bias towards either the personal best $\mathbf{p_i}$ or global best $\mathbf{g}$. Together, these equations attempt to create convergence, and diffusive space exploration similar to that of Brownian motion [5]. Recent work has led this toward more computationally expensive Lévy flights [34], [12], [7] for their improved convergence qualities.

$$\mathbf{v_{i+1}} = \omega\mathbf{v_i} + \phi_p r_p(\mathbf{p_i} - \mathbf{x_i}) + \phi_g r_g(\mathbf{g} - \mathbf{x_i}) \quad (1)$$

$$\mathbf{x_{i+1}} = \mathbf{x_i} + \mathbf{v_i} \quad (2)$$

It is prudent to also discuss the Geometric PSO (GPSO) and the implications that it puts forward. In the simplest implementation, this algorithm requires that a new mutation operator be defined, and a new crossover operator be defined for multiple parents; so as to mimic geometric movement toward a personal best solution and/or the global best solution. Considering that the original algorithm shown in Eqns. 1 and 2 requires geometric movement and specifically bias, toward either the global or personal best, it becomes clear that this new crossover operator must be able to bias towards one parent or the other, hence being weighted in some fashion.

Crossover and mutation in the context of linear genetic programs was investigated by Togelius and others [36] and several possible operators were proposed. The authors concede that significant research still remains in finding the most appropriate operators, but some effective ones presented include weighted subtree swap, weighted homologous crossover and weighted one-point crossover. Homologous crossover ensures that the common region between two candidates are kept intact [32]. Togelius and colleagues reports that common regions can sometimes be too small for this operator to be constructive [36]. The other two operators are more self-explanatory.

Finally, we now provide a brief summary of GPU-based simulation, especially using NVidia's (Compute Unified Device Architecture) CUDA platform [19]. The CUDA architecture arose from a potent arrangement of MIMD and SIMD computing, initially intended for processing large quantities of pixel data in parallel. Many researchers have spent years using "ping-pong" buffering with pixel and fragment shaders to modify textures in order to accomplish General-purpose GPU simulation (GPGPU). CUDA makes this process far more accessible.

Essentially, the GPU is divided into several Streaming Multi-processors (SMs) with a certain number of "CUDA cores" which process work units known as "blocks". Each of these blocks would be sized by the user, up to a maximum (at the time of writing) of 1024 threads. When an SM executes a block, the threads are divided into groups of 16, which are named "warps". These warps are the smallest unit of execution in CUDA. They are subsequently executed in a SIMD fashion on the CUDA cores in each SM. This arrangement is sometimes known as Single-instruction Multiple-thread (SIMT).

GPUs generally have some idiosyncratic behaviour regarding memory access coalescence, scoping and penalties, among other aspects which usually require special consideration. The CUDA memory hierarchy provides a range of memories with varying access times and scope restrictions, but we omit an extensive discussion on this for brevity. The process of executing simulations while taking advantage of the vast computing power of CUDA usually involves copying data to the GPU global memory from the host, then performing the GPU-specific code ("CUDA kernel"), and finally copying the modified data back. There are more efficient ways of utilising CUDA-enabled GPUs, such as host page-locked memory, which remove the need for expensive memory copies between host and device.

GPU-specific code is written by using special syntax which NVidia released as an addition to the C language. This syntax is parsed and compiled by the NVidia compiler, and then the rest of the code is passed to the system C/C++ compiler for normal processing. The result is a C or C++ program with additional non-C syntax which is effectively removed by the `nvcc` compiler, and the rest is compiled as a regular program.

3. Method

Our method for combining CUDA, GEP and GPSO is summarised in Alg. 1. We modify the weighted crossover of The Particle Swarm Programming algorithm of Togelius and colleagues [36] to operate on *k*-expressions by following the multi-parent crossover scheme the authors proposed. This is taken from [36] and shown in Eq. 3.

$$\Delta GX((a,w_a),(b,w_b),(c,w_c)) = GX((GX((a,\frac{w_a}{w_a+w_b}),(b,\frac{w_b}{w_a+w_b})),w_a+w_b),(c,w_c)) \quad (3)$$

GX is the crossover operator, and ΔGX is the multi-parent crossover operator. It is assumed that w_a, w_b and w_c are all positive and sum to 1. Essentially this equation defines the weighted, multi-parent crossover as two crossovers, the

Algorithm 1 The parallel implementation of the GPSO on GEP *k*-expressions .

```
allocate and initialise enough 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: update candidate bests
    CUDA: recombine and mutate programs
    replace old programs with new ones
  end if
  visualise the result
end while
```

first being between a and b, where weights are re-normalised to sum to 1, and the second is a crossover with c. Togelius, De Nardi and Moraglio provide more details on this using convex set theory [36]. See also [27], [24], [25], [36] for more details on the rationale and mathematical aspect of this procedure.

As can be seen in Alg. 1 we parallelise the majority of computations. In order to interpret the programs, we also need to compute an "argument map" so as to allow the interpreter to determine which arguments belong to which functions in program strings.

Now we have determined the mechanism by which we will ensure that crossover still maintains its geometric properties as much as possible. What remains to be determined is precisely how this will be done on the linear *k*-expressions . Ferreira [4] defines one-point crossover as choosing a crossover site or "pivot", and then exchanging symbols about this point to obtain two new candidates. In order to ensure that this crossover is geometric in the sense that we can compute a multi-parent one-point crossover and still be able to bias the result towards one parent candidate or the other, we must ensure that it is *weighted*.

Our method for accomplishing this recombination is by using the ω, ϕ_g and ϕ_p parameters as the weights (w_a, w_b and w_c) in Eq. 3. We further define the candidate a as the current candidate under consideration, b as the corresponding personal best of a, and c as the global best candidate discovered so far. The fitness values of these are not used in the crossover process. Notice also that unlike GP, we do not require selection, other than simply $P(crossover)$, a probability defined by the user, as in GP. GEP crossover defines a "donor" and a "recipient" tree, which are chosen randomly also.

Mutation is simple in comparison. Traditionally, mutation is derived from initialisation methods such as [39]:

1) Grow method
2) Full method
3) Ramped half-and-half

Typically, mutation is simply a replacement of a subtree by regrowing it using one of these methods. Point mutation is not the only space exploration operator, but it is the one given the most consideration [39] since the work of Koza [17].

Point mutation is simple for *k*-expressions , apart from the only restriction being that a tail-section symbol may not be swapped for a function symbol. This ensures that the result of the mutation operator is always a valid candidate. It is worth noting that a symbol in the head section may be changed into any other function symbol, regardless of arity [4]. The size of the head and tail are left as a configuration parameter, but can be computed to ensure the head is maximised. From Ferreira's work [4] the head and tail section sizes must satisfy the equation shown in Eq. 4:

$$t = h(n-1) + 1 \tag{4}$$

The symbols in the equation represent the tail length t, head length h and the maximum arity possible in the function set n.

Having described our algorithm we now turn our attention towards a suitable test platform. The Santa Fe Ant Trail is a classic problem used for evaluating genetic programming-based algorithms. Essentially the problem demands an appropriate combination of two function symbols (`IfFoodAhead` arity 2, `ProgN2` arity 2) and three terminal symbols (`Move`, `Right` and `Left`) for pursuing food particles in a spatial 2D environment. The `IfFoodAhead` function executes its first argument if there is food straight ahead of the particle, and the second argument if not. The `ProgN2` function simply executes both of its arguments in order. In our case, we have elected to use a 3D version of the problem. Apart from having `Left` and `Right` terminals, we introduce `Up` and `Down` terminals.

To the best of our knowledge, there has been no previous effort to parallelise the GEP (or indeed a variant of this) algorithm on GPUs. However, to assist in comparison, we have compared results from this algorithm against an implementation of Genetic Programming (GP) with tournament selection and *karva*-expressions as program representation.

Our analysis of this algorithm involves two aspects. Firstly, its ability to converge upon a good solution (preferably the global optimum), and secondly its ability to utilise the parallel architecture of a GPU; hence its *wall-clock* efficiency.

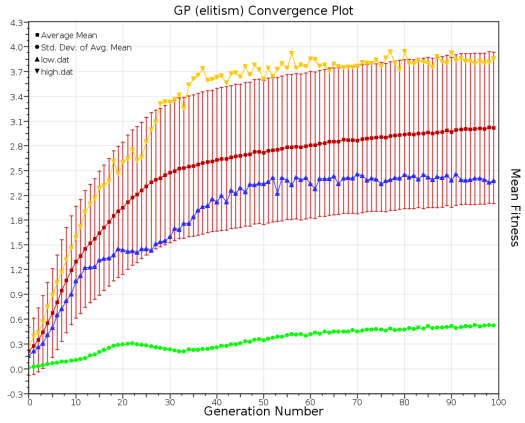

Fig. 3: Convergence results for the GP, with elitism. The graph shows the average mean value of each generation, from 100 independent runs. The error bars represent the average standard deviation of the 100 runs in each generation.

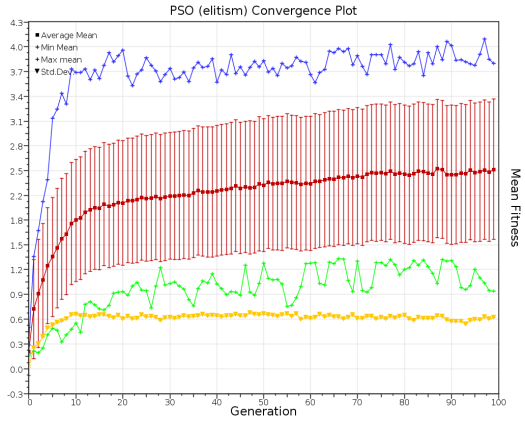

Fig. 4: Convergence results for the GPSO with elitism. Each data point has been averaged 100 times in independent runs, and the error bars represent the average standard deviation of each generation.

In our experiments we compare against the aforementioned GP implementation with *k*-expressions in terms of convergence and speed. The parameters we used for the GP were: P(Crossover) $= 0.8$, P(Mutate) $= 0.1$. We use the same crossover and mutation rates for the modified GPSO, and for the PSO-specific settings, we used: $\omega = 0.1$, $\phi_p = 0.6$, $\phi_g = 0.3$. As for the simulation itself, we restrict angular velocities to 0.1 units, and initial velocities are initialised to between -0.16 and 0.16. In order to use a higher mutation rate, Togelius et al recommend using Elitism, whereby the best candidate is replicated verbatim into the new population following the genetic operators. This is a common technique used in EAs to bias the population in a particular direction. We make use of elitism in both the GP and the GPSO.

4. Algorithm Convergence Results

Fig. 5 shows the convergence results for the modified GPSO and the GP. Each data point in all the plots shown have been averaged 100 times in independent runs. It is therefore conclusive that the PSO is indeed more able to find a good solution faster, but if computing fitness for more than about 23 frames is viable, then the GP is more appropriate.

Figs. 3 and 4 show the convergence results for the GP and GPSO respectively. We experimented with elitism, where the best individual is copied verbatim into the next generation, Fig. 5 shows conclusively that elitism allows the algorithms to perform better, albeit marginally.

Each data point of the Average Mean has been represents an average of 100 means from the same generation number. From these plots, it is clear that the GPSO has more spread per generation than the GP, which is not very desirable. The minimum and maximum values are also shown.

Finally, Fig. 6 shows the average compute time, by generation, for both the GP and the GPSO. The fitness evaluation consisted of computing 300 frames of the candidate programs and gathering fitness results from this. Therefore, each data point represents the average frame compute time across each of the 300 frames, and then averaged 100 times by independent runs. The generation compute times are also shown, although they are somewhat hidden. While the first observation seems that the GPSO is faster than the GP, this is somewhat misleading. Essentially, the plots in Fig. 6 would be completely linear, if all the terminal and function symbols were of the same complexity.

The average new-generation population compute time for the GP was $420\mu sec$, and for the GPSO it was $440\mu sec$. Even though this is not comparable to the fitness evaluation ($340,000\mu sec$), it was still worth the effort, as this must happen in serial following the fitness evaluation phase. The function `IfFoodAhead` has a rough complexity of $\mathcal{O}(f)$, where f is the number of food particles, which would approach $\mathcal{O}(fN)$, should all candidates have one of these symbols in its program. Of course, the worst case here is that every candidate consists only of these functions and enough terminals to satisfy the k-expression's head and tail sections. Hypothetically, given a maximum expression length $l = 8$, and a head length $h = 3$ (hence a tail length of 5), then the maximum number of `IfFoodAhead` functions would be 3. Extrapolating from this, assume all N particles were formed like this, then evaluation would be of complexity $\mathcal{O}(3fN)$, which could very well exceed $\mathcal{O}(N^2)$.

Therefore, following from this argument, we could perhaps conjecture that at generation 20, the GP increased its use of the `IfFoodAhead` function, while the PSO had reached a steady equilibrium of a certain number of these functions. This would seem to agree with our suspicion that the GP is in fact better in preserving population diversity. In Section 5 we explore this in more depth.

Our attempts to improve the GPSO beyond the results we see here was met with disappointment. Our parameter tuning

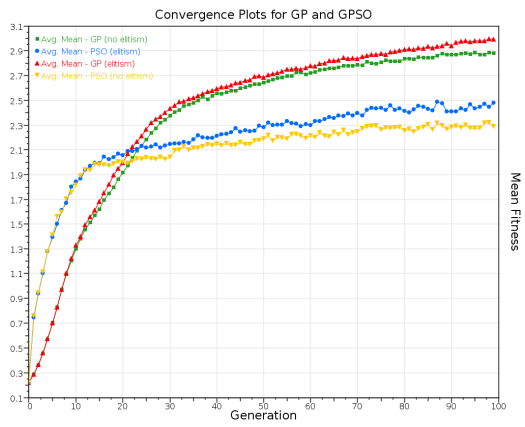

Fig. 5: Convergence results for the GP and GPSO, as well as the use of elitism for both. Each data point has been averaged across 100 independent runs to obtain meaningful statistical data.

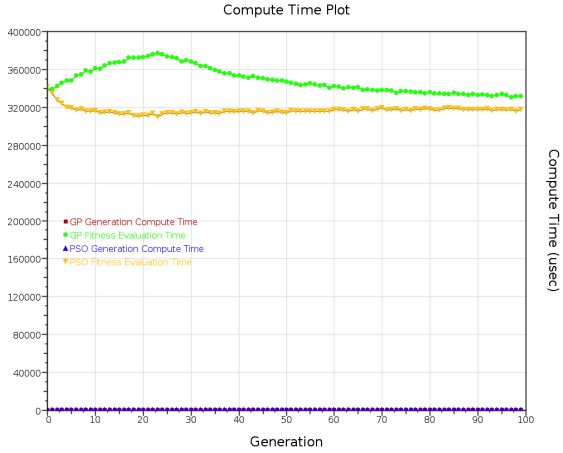

Fig. 6: Compute times for fitness evaluation, and generation compute time for GPSO and GP.

effort for phi_g, phi_p and ω included normalised combination of respective scores of particles and also normalised weighted scores, but the best parameters were simply $phi_g = 0.3, phi_p = 0.6, \omega = 0.1$.

Fine-tuning crossover and mutation probabilities had varying effects on convergence. Removing the crossover phase with the global best solution reduced mean scores to 0.2, and similar results were obtained from removing the crossover with the personal best. Randomising slightly the crossover point with hand-tuned parameters to aid in diversity did not improve scores at all.

Our results indicate that, at the very least, that the GPSO operating over *k*-expressions is appropriate for when the fitness evaluation is extremely computationally expensive. Given enough time and compute power, however, the GP operating on *k*-expressions is more suited to the problem.

5. Discussion

Evolutionary Algorithms such as the GPSO and the GP we have compared above frequently involve a very manual parameter-tuning effort in order to ensure an unbiased comparison. We describe a meta-optimiser (also based on the PSO) in [10]. We found that the PSO was suitable as a "super-optimiser" or "meta-optimiser" for fitness evaluations which are of relatively low compute expense. In this case, fitness evaluation was clearly far more expensive, and for a "meta-optimiser" to be successful in obtaining good parameters, it would need to be an optimiser which requires very few frames for a good solution. In this case, we believe that our meta-optimiser could potentially take months to obtain a result comparable to hand-tuning. Meta-optimisers, in general, are notoriously expensive to run.

From our experiments it is not immediately clear why the GPSO is not as effective as the GP over more than approximately 23 generations. We believe that this may be due to how diversity in population is managed between the GPSO and GP. Consider the following as the rationale for this: Fig. 5 shows a clear change in average mean fitness from approximately generation 10 for the GPSO. Whereas, for the GP, a very slight decrease in average mean fitness is shown. This is reminiscent of local minima in parametric optimisers. This may also be indicative of the inability of the GPSO to use extensive diversity to its advantage in escaping and reaching the global optimum. Consider also the artifact shown in Fig. 3 in the standard deviation at generation number 20. The same position in Fig. 4 is fully linear.

As for the performance data we present, it would be unwise to favour the GPSO from the observation in Fig. 6 that the frame compute time is lower. The slightly higher compute time does, after all, translate into a higher success rate as shown in Figs. 3 and 5.

6. Conclusions

We have presented a modified Geometric Particle Swarm Optimiser (GPSO) searching through the space of Ferreira's *k*-expressions. We have also compared this against a Geometric version of the canonical Genetic Programming method for evolutionary optimisation in the space of *k*-expressions. Our results show that the GPSO is not clearly superior over the GP, however, it is able to attain an acceptable solution faster, more consistently. This could be a desirable attribute, especially when the fitness function is inordinately expensive to compute.

We have also shown that Geometric algorithms such as the GPSO can be parallelised effectively in both the fitness evaluation phase, and the genetic operator phase (mutation, recombination). CUDA is particularly effective in this case, as evolutionary algorithms lend themselves well to data-parallelism.

There is scope for other Evolutionary Algorithms with geometric modification and parallelisation to be investigated using similar GPU/Hybrid techniques to those we have presented.

References

[1] Markus Brameier. *On Linear Genetic Programming*. PhD thesis, University of Dortmund, 2004.

[2] Mark Crosbie and Eugene H. Spafford. Applying genetic programming to intrusion detection. Technical report, Department of Computer Sciences, Purdue University, West Lafayette, 1995. AAAI Technical Report FS-95-01.

[3] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proc. Sixth Int. Symp. on Micromachine and Human Science*, pages 39–43, Nagoya, Japan, 1995.

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

[5] G.E.Uhlenbeck and L.S.Ornstein. On the theory of the Brownian motion. *Phys.Rev.*, 36:823–841, Sep 1930.

[6] J. H. Holland. *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press, 1975.

[7] Gang Huang, Yuanming Long, and Jinhang Li. Levy flight search patterns in particle swarm optimization. In *Seventh International Conference on Natural Computation*, 2011.

[8] A. V. Husselmann and K. A. Hawick. Spatial agent-based modelling and simulations - a review. Technical Report CSTN-153, Computer Science, Massey University, Albany, North Shore,102-904, Auckland, New Zealand, October 2011. In Proc. IIMS Postgraduate Student Conference, October 2011.

[9] A. V. Husselmann and K. A. Hawick. Levy flights for particle swarm optimisation algorithms on graphical processing units. Technical report, Computer Science, Massey University, 2012. Submitted to J. Parallel and Cloud Computing.

[10] A. V. Husselmann and K. A. Hawick. Particle swarm-based meta-optimising on graphical processing units. Technical report, Computer Science, Massey University, 2012. Submitted to AsiaMIC, Phuket, Thailand 2013.

[11] A. V. Husselmann and K. A. Hawick. Spatial data structures, sorting and gpu parallelism for situated-agent simulation and visualisation. In *Proc. Int. Conf. on Modelling, Simulation and Visualization Methods (MSV'12)*, pages 14–20, Las Vegas, USA, 16-19 July 2012. CSREA.

[12] A. V. Husselmann and K. A. Hawick. Random flights for particle swarm optimisers. In *Proc. 12th IASTED Int. Conf. on Artificial Intelligence and Applications*, Innsbruck, Austria, 11-13 February 2013. IASTED.

[13] Alwyn V. Husselmann and K. A. Hawick. Parallel parametric optimisation with firefly algorithms on graphical processing units. In *Proc. Int. Conf. on Genetic and Evolutionary Methods (GEM'12)*, number CSTN-141, pages 77–83, Las Vegas, USA, 16-19 July 2012. CSREA.

[14] A.V. Husselmann and K.A. Hawick. Simulating species interactions and complex emergence in multiple flocks of boids with gpus. In T. Gonzalez, editor, *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pages 100–107, Dallas, USA, 14-16 Dec 2011. IASTED.

[15] Kennedy and Eberhart. Particle swarm optimization. *Proc. IEEE Int. Conf. on Neural Networks*, 4:1942–1948, 1995.

[16] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proc. IEEE Int. Conf. on Neural Networks*, pages 1942–1948, Piscataway, NJ. USA, 1995.

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

[18] W. B. Langdon. A many-threaded cuda interpreter for genetic programming. In Ana Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP*, pages 146–158. Springer, April 2010.

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

[20] Sean Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In J. R. Koza, W. Banzhaf, K. Chellapilla, D. Kumar, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the 3rd annual conference*, pages 214–222. Morgan Kaufmann, San Mateo, California, 1998.

[21] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. *Robocup-97: Robot soccer world cup I*, 1:398–411, 1998.

[22] Steven M. Manson. 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.

[23] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[24] A. Moraglio. *Towards a Geometric Unification of Evolutionary Algorithms*. PhD thesis, Computer Science and Electronic Engineering, University of Essex, 2007.

[25] A. Moraglio, C. Di Chio, and R. Poli. Geometric particle swarm optimization. In M. Eber et al, editor, *Proceedings of the European conference on genetic programming (EuroGP)*, volume 4445 of *Lecture notes in computer science*, pages 125–136, Berlin, 2007. Springer.

[26] A. Moraglio and S. Silva. Geometric differential evolution on the space of genetic programs. *Genetic Programming*, 6021:171–183, 2010.

[27] A. Moraglio and J. Togelius. Geometric differential evolution. In *Proceedings of GECCO-2009*, pages 1705–1712. ACM Press, 2009.

[28] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11:339–363, 2010.

[29] Panait and Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11:387–434, 2005.

[30] R. Poli, W.B. Langdon, and N.F. McPhee. *A field guide to genetic programming*. lulu.com, 2008.

[31] Riccardo Poli and Stefano Cagnoni. 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*, pages 269–277. Morgan Kaufmann, 1997.

[32] Riccardo Poli and Nicholas F. McPhee. Exact schema theory for gp and variable-length gas with homologous crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

[33] Riccardo Poli, Leonardo Vanneschi, William B. Langdon, and Nicholas Freitag McPhee. Theoretical results in genetic programming: the next ten years? *Genetic Programming and Evolvable Machines*, 11:285–320, 2010.

[34] Toby J. Richer. The levy particle swarm. In *IEEE Congress on Evolutionary Computation*, 2006.

[35] Yuhui Shi and Russel Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation Proceedings*, 1998.

[36] Julian Togelius, Renzo De Nardi, and Alberto Moraglio. Geometric pso + gp = particle swarm programming. In *2008 IEEE Congress on Evolutionary computation (CEC 2008)*, 2008.

[37] Sjors van Berkel. Automatic discovery of distributed algorithms for large-scale systems. Master's thesis, Delft University of Technology, 2012.

[38] Frans van den Bergh. *An Analysis of Particle Swarm Optimizers*. PhD thesis, University of Pretoria, 2001.

[39] Matthew Walker. Introduction to genetic programming. Downloaded from http://www.cs.montana.edu/ bwall/cs580/introduction_to_gp.pdf February 5, 2013.

[40] Chi Zhou, Weimin Xiao, Thomas M. Tirpak, and Peter C. Nelson. Evolving accurate and compact classification rules with gene expression programming. *IEEE Transactions on Evolutionary Computation*, 7:519–531, December 2003.