Computational Science Investigations - Technical Note **CSI-0010**

Computational Science Investigations
CSI: Hull

Automatic High Performance Structural Optimisation for Agent-based Models

A. V. Husselmann and K. A. Hawick

2014

The problem of structural optimisation for agent-based models is one which holds great promise. Being able to optimise a set of behaviours has potential to improve productivity greatly, and at the very least, generate inspiration. This problem consists of three smaller problems which must be mitigated: ease of use, performance, and also the use of combinatorial optimisation. In this article, these three problems are managed by introducing a domain-specific language (DSL) operating over graphical processing units (and also single-threading) for performance, and a suitable optimiser for this architecture. We carry out a number of experiments to demonstrate and evaluate the performance and effectiveness of this approach. We conclude that such a methodology is indeed useful and performs adequately but is currently limited by the lack of debugging support and visual programming tools.

Keywords: CUDA, parallel, optimisation, domain-specific languages, agent-based models, karva

BiBTeX reference:

```
@InProceedings{CSI-0010,
       Title = {Automatic High Performance Structural Optimisation for Agent-based Models},
       Author = {A. V. Husselmann and K. A. Hawick},
       Booktitle = {Proc. 14th Int. Conf. on Software Engineering Research and Practice (SERP'14)},
       Year = {2014},
       Address = {Las Vegas, USA},
       Month = {21-24 July},
       Note = {ISBN: 1-60132-286-0},
       Pages = {346-352},
       Publisher = {WorldComp},
       Keywords = {CUDA, parallel, optimisation, domain-specific languages, agent-based models, karva},
       Owner = {kahawick},
       Timestamp = {2014.08.17},
       Url = {http://digital.hull.ac.uk/reports/0010/csi-0010.html}
}
```


Prof Ken Hawick, Computer Science, University of Hull, Cottingham Road, Hull, HU6 7RX, UK.

Automatic High Performance Structural Optimisation for Agent-based Models

A.V. Husselmann[1] **and K.A. Hawick**[2]
[1]Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
[2]Computer Science, University of Hull, Robert Blackburn, Hull, HU6 7RX, United Kingdom
email: [1]a.v.husselmann@massey.ac.nz, [2]k.a.hawick@hull.ac.uk
Tel: +64 9 414 0800 Fax: +64 9 441 8181

Abstract— *The problem of structural optimisation for agent-based models is one which holds great promise. Being able to optimise a set of behaviours has potential to improve productivity greatly, and at the very least, generate inspiration. This problem consists of three smaller problems which must be mitigated: ease of use, performance, and also the use of combinatorial optimisation.*

In this article, these three problems are managed by introducing a domain-specific language (DSL) operating over graphical processing units (and also single-threading) for performance, and a suitable optimiser for this architecture. We carry out a number of experiments to demonstrate and evaluate the performance and effectiveness of this approach. We conclude that such a methodology is indeed useful and performs adequately but is currently limited by the lack of debugging support and visual programming tools.

Keywords: CUDA, parallel, optimisation, domain-specific languages, agent-based models, karva.

1. Introduction

Agent-based Modelling (ABM) is an elegant and interdisciplinary simulation methodology. It is particularly applicable in areas where a system is comprised of individual components which are *situated* in some manner, *communicate or interact* in a restricted manner, and have some level of *autonomy* [1]. The practice of ABM has already reached into several diverse disciplines and studies including cancer immunology [2], social science [3], synthetic biology [4], land change modelling [5] and even criminology [6].

There is a lack of agreement over the precise definitions of ABM, but as noted by Macal and North, the set of the most common attributes of agent-based models in the literature are [1]:

1) Identity - Agents must be identifiable, discrete individuals.
2) Situation - Agents are situated in some fashion.
3) Goal-oriented - Agents have goals to achieve.
4) Autonomy - Agents are autonomous and may operate independently.
5) Learning - Agents could potentially learn and adapt.

A well-known agent-based model is the Predator-prey model [7], [8]. In essence, this model emulates a simple "food chain" wherein individual predators pursue prey. Prey attempt to flee, but if predators surround the prey, it is eaten. Should a predator not catch prey, it dies of starvation. Both prey and predators breed. This causes an interesting effect on population numbers. In general, this model is useful for ecological inquiry, and also gives a useful benchmarking tool for machine learning algorithms such as Genetic Programming [9]. Despite its simplicity, interesting discoveries have been made with this model. Jim and Giles showed that if predators communicate with simple evolved languages, they are able to catch prey more easily [10].

Such models normally have parameters that require calibration for the purpose of model validation and verification. Instead of laborious trial-and-error tuning, the calibration process is typically reinterpreted as an optimisation problem [11], [12], [13]. Several attempts have been documented in the past, mostly involving genetic algorithms [12], [14], and other evolutionary algorithms [13]. A limitation of such systems is that they only make provision for automatically calibrating *scalar* parameters.

These efforts can be very useful, but cannot extensively modify the structure of the model itself without considerable difficulty in representation. Should a model be structurally suboptimal, it may well be that a properly configured parameter optimisation effort could be entirely fruitless. The work of Epstein in 1999 perhaps earmarked evolutionary algorithms (EAs) for structural optimisation with agent-based models [15]. Epstein noted that, in comparison, parameter optimisation deals with a much smaller search space than that of the search for a set of behaviours, or rules.

Like parameter optimisation, there have been a number of attempts to optimise structure in agent-based models. In 2012, van Berkel proposed the use of Grammatical Evolution (GE) [16] for the purpose of generating NetLogo [17] programs based on a set of predetermined "building blocks" [18], [19]. EAs such as GE do require a cost function to indicate the relative fitness difference between individual candidates. In his work, van Berkel conceded issues regarding the process of selecting these functions. Earlier, in 2010, Learning classifier systems, Q-learning and neural networks were investigated for the same purpose [20]. Among these algorithms, none were clearly better than the others. Another attempt involved the Ant Hill problem, and was done by the evolution of finite state machines using EAs [21].

It is not generally easy to write an agent-based model in C++ and implement an evolutionary algorithm optimiser on top of it. Therefore, these previous attempts suggest that some method of improving usability is very important. Moreover, this would allow experts in agent-based modelling in other disciplines unrelated to computer science to immediately harness this technology.

In terms of performance, meta-optimisation on the particle swarm optimiser (PSO) [22] gives clear indications of what to expect when optimising any aspects of an agent-based model. At its heart, the PSO is a modified flocking model [23], [24], as it is inspired from such behaviour as seen in simulations such as Boids [25]. Van Berkel's work was distributed on a set of processors, however, experiments were deemed to still be too expensive to compute (3 or more hours) [18]. Privošnik approached the problem by having a heuristic to reduce fitness evaluations [26]. Coupled with a growing interest in larger-scale agent-based models, and recent breakthroughs in general purpose graphics processing unit usage (GPGPU), parallel ABMs has emerged [27], which is capable of mitigating performance issues.

Here, emphasis is given to performance improvements using Graphical Processing Units (GPUs) for the purpose of parallelisation. In our previous works, we have investigated the use of GPUs in the context of a domain-specific language (DSL) [28], [29], [30]. DSLs are special languages developed for the purpose of servicing a very specific problem domain [31]. The main purpose in doing so was to ensure that optimisation tasks could be specified within the same language that the model is written, and ensuring that it is easy to do so. It has been proposed that DSLs be known by having a well defined domain, a clear notation; the semantics of which is to be formally and informally equivalent [31].

We have previously developed a DSL named SOL, which is primarily intended for lattice-oriented ABM [28], [29], [30]. It includes two optimisation algorithms, one is based on Karva [32], Genetic Programming [33], and Compute Unified Device Architecture (CUDA) [34]. The other is a very simple tree-based evolutionary algorithm.

The language itself is built on Terra [35], which is a very recent multi-stage programming language [36]. It makes use of LLVM [37], which is a very mature compiler architecture. Part of the motivation behind this choice was the release of NVidia's parallel thread execution (PTX) backend compiler for LLVM (NVPTX) [38]. In this article, we aim to introduce the modifications necessary to the SOL language and demonstrate its performance characteristics. We also demonstrate the use of the SOL language to solve a well known problem used in Genetic Programming.

In Section 2 the use of optimisation in the SOL language is described. SOL is extended in Section 3 with GPGPU, and some experiments are then carried out and reported on in Section 4. Finally, we draw some conclusions in Section 5.

2. SOL Model Optimisation

In the optimisation methodology described here, instead of a modeller providing fine-grained and carefully articulated local behaviours in a custom optimiser, the modeller instead provides an objective function and some possible behaviour *within* simulation code. The difficulty in both these tasks are comparable: small variations in local behaviours as well as different choices of objective function can both lead to radically different results. However, the novelty in this approach to the problem lies in the elegance with which a model can be prototyped with all aspects including optimisation encapsulated within it, without suffering performance penalties.

In cases where an objective function is more natural to use in modelling, it can be a great advantage to have the behaviour generated from it, or at least provide a suggestion which can be used as the basis for manual experimentation. By allowing a modeller to express most of a simulation with reasonable certainty, and other parts with annotated uncertainty and clear objectives, an underlying optimiser can usefully generate some behaviours.

One of the factors which sets this new language apart from other model induction attempts is that the syntax used for optimisation is itself a constraint upon the search space. Techniques such as Genetic Programming have been used in the past for evolving entire models [19] which were provided with user-defined actions and perceptions of agents. In the case of SOL, only relevant actions and perceptions are provided, but they are given in terms of a *potential solution* given by the user. This kind of syntax serves to (1) provide a good starting solution and more importantly (2) limit the immense search space. It is reasonable to assume that the dynamics of a system is at least partially known by the expert, and therefore, it is worth optimising only the *uncertain* portions.

Given a method for expressing the objective function and for measuring it, some method of configuring the output desired from the optimiser is required. Certain constraints apart from search space define, in large part, the operation of the optimiser, or even what kind of optimiser is used. Three such configurations are implemented by specifying one of the keywords recombination, single, or permutation. These qualifiers tell the compiler which kind of optimisation is required. What is referred to by these qualifiers is some kind of reorganisation of *statements* marked for optimisation. While single and permutation refer to the selection of a single statement only, or a permutation of statements (with replacement), recombination refers to a *disassembly* of the provided code into terminals and nonterminals for recombination. The recombination qualifier is given the most attention in this article, since the other two qualifiers have been previously described [28], [30].

A complete example of this is given in Listing 1. Lines 34–56 contain what will be referred to as an uncertain construct. This term is purely related to the structure of the model, and not of code semantics. The quantity is count_sheep() and the objective is to minimise this (hence

the minimise keyword given on line 34).

In cases where the fitness of a model depends partly on stochastic behaviours, it is important to obtain statistically significant scores. This is typically done by averaging multiple evaluations of the objective function. This has certain implications on performance, which is a major factor in our decision to use parallel computing on GPUs. Though these devices are less expensive than grid computers, effort is required to ensure that a program maximises their computing power.

In this process, a user first provides a program with an uncertain code segment within a model. Upon execution of the host program which is written in C++, the Terra runtime parses the provided code immediately, transforming it into a type-checked abstract syntax tree (AST). This tree is what is modified by the optimiser in a later stage. The host Lua script then duplicates this tree several times until a population of N trees is made. At this point, the optimiser searches for an optimisation statement, of which there can be only one within a program. The optimiser then initialises each program's code segment with randomly chosen statements taken from within the provided construct. Depending on the type of construct, there will be one chosen statement (single), or a combination of statements (permutation) with replacement, or a complete recombination of code (recombination).

In summary, the overall process is detailed in Algorithm 1. This algorithm contains all the components generally expected within an EA, and more specifically one which uses genetic operators. The termination criteria shown is simply a maximum number of generations. Computing a new generation of programs is done by the process shown in Algorithm 2.

Algorithm 1 The complete simulation process eliminating uncertain constructs.

```
Terra custom parser reads agent description
Read user-provided parameters
Allocate & initialise space for n candidates
{small variations when no uncertainty is present}
Compute and compile a new generation of candidates
Zero all scores
while Termination criteria not met do
  for x frames do
    Execute model programs
    Collect new scores into running totals by model
    Visualise the result
  end for
  Compute a new generation using collected scores
end while
Output best candidate in final population.
```

3. Parallel SOL

Previously, a lattice along with a temporary write-only lattice was allocated on the host. Should CUDA be enabled in a SOL model, the data is instead allocated on the

```
1  sol
2    defvar count = 1
3    defvar pred = 1
4
5    query neighbours6
6      if neighbour == 1 then
7        count = count + 1
8      else
9        if neighbour == 2 then
10         pred = pred + 1
11       end
12     end
13   done
14
15   defvar predator = 2
16   defvar prey = 1
17
18   if me == predator then
19     defvar temp = get_closest_prey
20     move towards temp
21     if pred == 6 then die end
22     split
23   end
24
25   defvar eq = count_sheep()
26
27   if me == prey then
28     defvar closepred = get_closest_predator()
29     if (distance to (closepred)) < 2
30       then
31       die
32     else
33       defvar closeprey = get_closest_prey()
34       select permutation to minimise(eq)
35         -- flee predator (F)
36         if (distance to (closepred)) < 3
37                 then
38           move awayfrom closepred
39         end
40         -- breed (B)
41         if (distance to(closeprey)) < 2
42                 then
43           split
44         end
45         -- overcrowding
46         if (count > 7) then
47           die
48         end
49         -- move randomly
50         move random 4
51         -- seek mate (M)
52         if (distance to(closeprey)) >= 2
53           then
54           move towards closeprey
55         end
56       end
57     end
58   end
59 end
```

Listing 1: A program written in the SOL DSL for the Predator-Prey model, containing an uncertain construct (lines 42–64).

Algorithm 2 The process of generating a new generation of candidate models for evaluation.

```
Collect scores
for x candidate models do
    Optimiser performs evolutionary operators
    Pass modified typed tree through code generator
    Wrap generated function code with arguments
    Emit wrapped code
end for
Overall generated code is compiled to machine code
Fn pointer to compiled function passed to C++ via Lua
```

GPU hardware by using the CUDA API [34] at runtime. These device pointers are provided to the Terra compiled function, which SOL is compiled to in turn, instead of pointers to host memory. The compiled SOL code is therefore able to operate on the lattice, as allocated by the host on the GPU hardware. The code parser and type checker are identical, but a separate CUDA code generator is used in order to accommodate the restrictions imposed by the CUDA GPU architecture. Compiled code is then mostly PTX instructions, wrapped with the necessary host code to launch CUDA kernels with the correct thread grid and block dimensions. Once a timestep is computed, the data is copied back from the GPU to the host and then passed to the visualiser.

Compiling Terra code for CUDA is straightforward, provided that the boundaries of the device in terms of memory and thread resources are respected. The usual Terra code generated is essentially compiled into a single CUDA kernel, which is launched with a grid and block configuration, and its arguments, by a separate host Terra function. Given that an appropriate grid and block must be provided, this presents an opportunity to discuss different parallelisation techniques.

Three parallelisation strategies are implemented from which the user may freely choose. The first is a simple "one-thread, one-model" (1T1M) strategy, where a single CUDA thread is assigned a candidate model. This CUDA thread is then responsible for executing the entire model simulation once per time step. This is unsuitable most of the time, especially when one candidate model operates on a larger lattice, or the model is demanding of processing time required. The second strategy is named "one-block, one-model" (1B1M), in which an entire CUDA block is dedicated to computing a single model simulation once per timestep. While this may seem the obvious choice in nearly all circumstances, the limitations of block sizes (1024 threads maximum at the time of writing), mean that the lattice sizes have a limit. A great many candidate model simulations can be executed concurrently at reasonable speeds using this, but the limitation in lattice size is a considerable issue. The third strategy is termed "many-blocks, one-model" (*B1M), where multiple blocks are assigned to a single candidate model. This allows much larger model sizes, but race conditions become more difficult to eliminate, which require further strategies.

Another considerable issue is the source of random deviates on the GPU hardware. There are various methods to accomplish this [39]. In this case, this was implemented by maintaining a separate GPU array with three unsigned long integers for every candidate model. These integers represent the u, v and w parameters of the Ran random number generator [40]. They are initialised by the host before being copied to the device. The host computes a new Ran state for each candidate model using a master Ran for providing a seed. This allows the SOL code to use as many random deviates as it needs to operate, since Ran also provides a colossal period, approximately $3.138(10^{57})$ [40]. To generate a random number, code is automatically generated from a macro function to update the Ran state and compute a random deviate of the specific thread.

4. Selected Results

Two experiments are conducted to evaluate convergence and computing performance. The first is The Santa Fe Ant Trail problem which Koza solved using Genetic Programming [41]. This problem was approached using a select recombination structure, in order to evolve the appropriate decision tree (expression tree) for an ant to collect all food placed on an irregular trail. The second experiment pertains to a more classical model: the Predator-prey model [42], [43]. The chosen objective in the Predator-prey model is to evolve a list of ordered rules which are most suitable for the predators to catch the prey.

The parameters used for these experiments are shown in Table 1.

Santa Fe Parameter	Value
Population Size	500
P(mutate)	0.2
P(crossover)	0.8
Program head length	12
Timesteps per Generation	400
Predator-prey Parameter	**Value**
Population Size	500
P(mutate)	0.2
P(crossover)	0.8
Timesteps per Generation	200
Generation Repeats	3

Table 1: Optimisation parameters used for the Santa Fe Ant Trail problem and the Predator-prey model.

4.1 Santa Fe Ant Trail

The fitness plot for this experiment is shown in Figure 1. There is a gradual (though small) decrease in mean fitness up to generation 120, and followed later by a slight drop around generation 280. The minimum fitness does decrease over time, indicating progress in the search. Though, during this run, the optimum fitness (zero) was not achieved, a number of semi-suitable programs were generated. It is worth noting that no generated program can achieve maximum fitness by brute force iteration of the entire lattice due to the 400 time step maximum [41].

The time taken to compile all 500 typed Terra trees to *host* code (in single-threaded SOL) is approximately

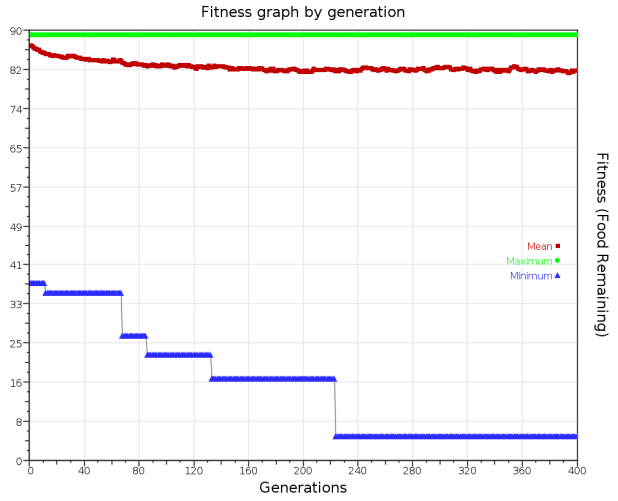
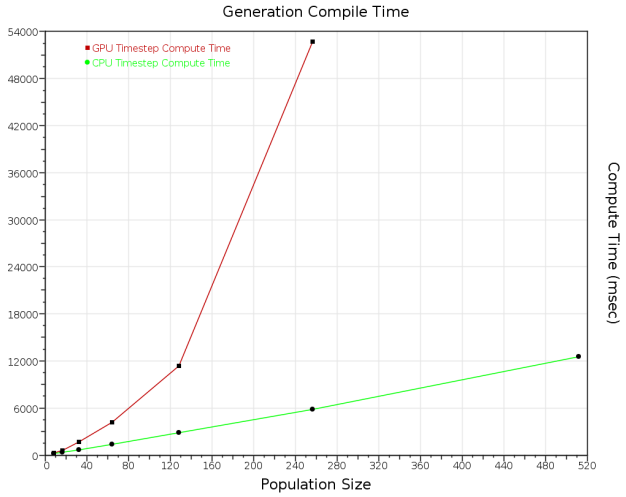
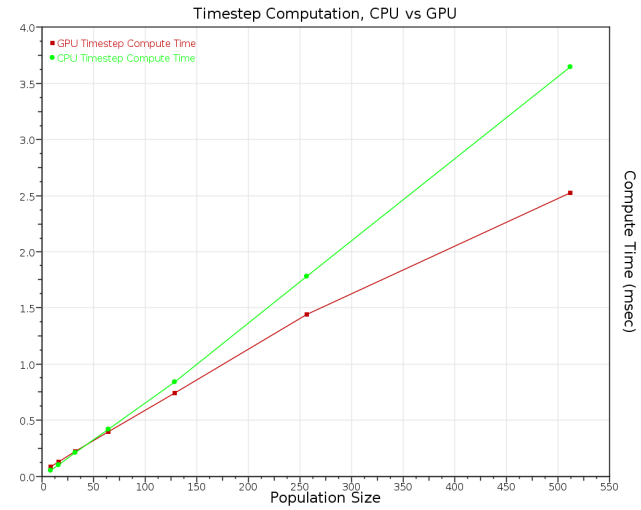


Fig. 1: Fitness plot by generation for the Santa Fe Ant Trail problem computed by the host using the single-threaded version of SOL.



(a) Generation population compilation. Data point for GPU at 512 population size not shown: 539000msec. Parallelisation strategy used was "one-block, one-model."



(b) Timestep computation.

Fig. 2: Performance plots of timestep computation and population compiling between CPU and GPU compiled SOL code. Data is averaged over the first 300 time steps of randomly initialised runs.

14.8 seconds (averaged five times). Whereas, the time to compile 500 unique typed trees to PTX is approximately 473 seconds. This is clearly an undesirable amount of time, considering that this dwarfs the evaluation time.

The time taken to compute a single timestep of a system with 500 candidates is approximately 3.57msec (averaged five times) for the single-threaded host version, whereas the GPU-parallel version computes a frame in 2.5msec (averaged five times). These are both of the order of a second for evaluating the entire population in the current generation. Additional scaling data for increasing population sizes are shown in Figures 2(a) and 2(b). In these plots it is clear that unless there is excessive computation necessary in the evaluation of a population of candidates, then it is likely that the GPU version of SOL is not necessary. At this point it is not clear why there appears to be an exponential rise in compute time required for compiling larger simulations at runtime for GPU, though the use of CUDA run-time code generation (RTCG) in Terra is experimental at this point. Also interesting in the timestep plot is that the GPU code surpasses the CPU code at relatively small population sizes. Though the CPU code is faster for population sizes of 32 and less, it was expected that the CPU code will surpass the GPU until at least 128 candidate programs. Altogether, the total lattice size operated on for the largest population was sized $512(32) = 16384$ by 32 lattice sites. This is a colossal 524,288 lattice sites, which makes the computing time for one frame considerably more reassuring for both CPU and GPU code.

4.2 Predator-Prey Model

For the Predator-prey model, timestep computation times were measured for the single-threaded and GPU-parallel compilations for different population sizes and lattice sizes. This model is more computationally expensive to compute per timestep than the Santa Fe Ant Trail. The reason for this is that both predators and prey must execute a SOL program, meaning that at times, every lattice site will execute a program, whereas in the Santa Fe Ant Trail, there was only one program being executed, that of the ant.

The optimisation objective is to select a permutation (with replacement) of rules for prey, to maximise the number of prey. Usually, the evasion strategy for prey is to find the closest predator, and move directly away from it. This is a simple but effective strategy. Emphasis here is given on performance results instead of convergence results.

The measured data in log-linear plots are shown in Figure 3. Different configurations of candidate model lattice sizes (8x8,16x16,32x32,64x64 and 128x128) were used, along with the "one-block, one-model" (1B1M) CUDA

parallelisation strategy, the "many-blocks, one model" (*B1M) strategy, as well as single-threaded CPU configurations. The largest candidate model lattice (128x128) proved prohibitively expensive to compute by the single-threaded CPU configurations. There exist rapid increases in computing time between candidate lattice sizes, but increases in population size of these are relatively slower. The mere ability to simulate a lattice of size 128x128 with 64 heterogeneous candidates is enormously encouraging, considering that a population of this size is advantageous with regard to population-based optimisers. Unfortunately, while timestep computation scales well, compile time does not, however.

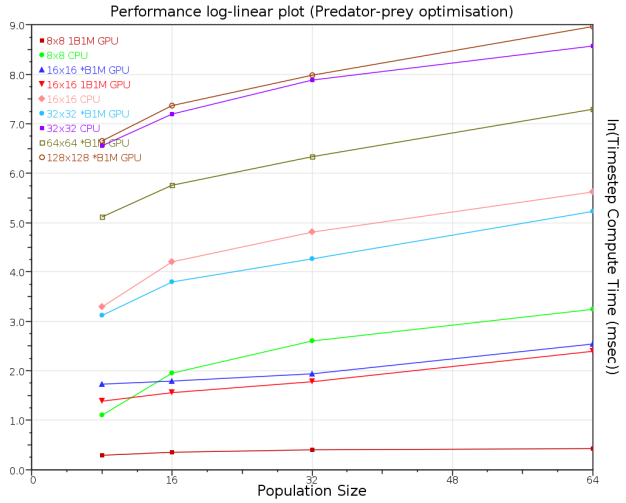


Fig. 3: Performance plot for different system sizes both in candidate model lattice sizes and number of candidates in populations for both the parallelisation strategies "one-block, one-model" (1B1M) and "many-blocks, one model" (*B1M). CUDA block sizes in *B1M were restricted to 16x16.

For the complete program shown in Listing 1, compile times from a SOL typed syntax tree to Terra code takes between 25msec and 30msec, however, some programs take up to 130msec to compile for the predator-prey model with heterogeneous programs (caused by the optimiser exploring the search space). The CUDA kernel compile time for 32 heterogeneous programs is approximately 40 seconds. For a trivially simple program (sol move left end), compile time from SOL typed tree to Terra is approximately 1msec, and kernel compilation is 530msec for 32 identical programs. A slightly more complex trivial program, but one still containing an optimisation construct such as the following takes on average 5msec to compile from a SOL typed tree to Terra, and approximately 1msec for a CUDA kernel to compile:

```
sol select recombination to minimise(1) move left end end
```

5. Conclusion

The use of run-time code generation is a good method for improving performance, provided that the evaluation phase of a population of candidate models is sufficiently complex. For models in which evaluation is less expensive (such as the Santa Fe Ant Trail model), it is more appropriate to use the single-threaded version of SOL. This allows programs to be compiled faster, while suffering a very small drop in timestep computation performance.

Systems larger than 512 candidate models appear to be out of reach of GPU run-time code generation. Such a result was expected, given that Cupertino et al. chose to evolve PTX code itself rather than use the CUDA run-time library to compile C code [44] noting that the latter would be too computationally expensive.

These limitations reaffirm that it is unwise to ignore the power of the newer multi-core processors available [45]. At the same time, the GPU should be applied when most or all of its theoretical computing power can be achieved. This demands proper choice in parallelisation strategy, which is anticipated to be automatically selected using relevant model information in the future. It is certainly possible to generate multi-threaded code on the host processor instead of CUDA code. This is a promising area for future work.

In summary, our SOL system is considerably more powerful and useful when it is supported by underpinning parallel computing capabilities. At the time of writing, SOL is a useful tool for experts, but there is scope for additional debugging and other user-friendly features that would make it useful for application domain users. We believe this approach has great power in providing a basis for developing initial application domain ideas into practical running simulations.

References

[1] Macal, C.M., North, M.J.: Tutorial on agent-based modeling and simulation part 2: How to model with agents. In: Proc. 2006 Winter Simulation Conference, Monterey, CA, USA. (3-6 December 2006) 73–83 ISBN 1-4244-0501-7/06.

[2] Figueredo, G.P., Siebers, P.O., Aickelin, U.: Investigating mathematical models of immuno-interactions with early-stage cancer under an agent-based modelling perspective. BMC Bioinformatics **14**(6) (2013) 1–38

[3] Macy, M.W., Willer, R.: From factors to actors: Computational sociology and agent-based modeling. Annual Review of Sociology **28** (2002)

[4] Gorochowski, T.E., Matyjaszkiewicz, A., Todd, T., Oak, N., Kowalska, K., Reid, S., Tsaneva-Atanasova, K.T., Savery, N.J., Grierson, C.S., di Bernardo, M.: BSim: An agent-based tool for modeling bacterial populations in systems and synthetic biology. PLoS ONE **7**(8) (August 2012)

[5] 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** (2005) 47–62

[6] Birks, D., Townsley, M., Stewart, A.: Generative explanations of crime: Using simulation to test criminological theory. Criminology **50** (2012) 221–254

[7] Lotka, A.J.: Elements of Physical Biology. Williams & Williams, Baltimore (1925)

[8] Volterra, V.: Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. Mem. R. Accad. Naz. dei Lincei, Ser VI **2** (1926)

[9] Luke, S., Spector, L.: Evolving teamwork and coordination with genetic programming. In: Proceedings of the First Annual Conference on Genetic Programming, MIT Press (1996) 150–156

[10] Jim, K., Giles, C.: Talking helps: evolving communicating agents for the predator-prey pursuit problem. Artificial Life **6**(3) (2000) 237–254 Summer.

[11] Calvez, B., Hutzler, G., et al.: Adaptative dichotomic optimization: a new method for the calibration of agent-based models. In: A. Tanguy C. Bertelle J. Sklenar et G. Fortino, éditeurs, Proceedings of the 2007 European Simulation and Modelling Conference (ESMâĂŹ07). (2007) 415–419

[12] Calvez, B., Hutzler, G.: Automatic tuning of agent-based models using genetic algorithms. In: Proceedings of the 6th International Workshop on Multi-Agent Based Simulation (MABS 2005). (2005) 41–57

[13] Stonedahl, F., Wilensky, U.: Finding forms of flocking: Evolutionary search in abm parameter-spaces. In: Multi-Agent-Based Simulation XI. Springer (2011)

[14] Said, L.B., Bouron, T., Drogoul, A.: Agent-based interaction analysis of consumer behaviour. In: Proceedings of the First International Joint Conference on Autonomous agents and multiagent systems: part 1, ACM (July 2002) 184–190

[15] Epstein, J.M.: Agent-based computational models and generative social science. Generative Social Science: Studies in Agent-Based Computational Modeling (1999) 4–46

[16] Ryan, C., Collins, J., O'Neill, M.: Grammatical evolution: Evolving programs for an arbitrary language. In: Proceedings of the First European Workshop on Genetic Programming. Volume 1391 of LNCS., Paris, Springer-Verlag (April 1998) 83–95

[17] Tisue, Wilensky: NetLogo: A simple environment for modeling complexity. In: International Conference on Complex Systems. (2004)

[18] van Berkel, S.: Automatic discovery of distributed algorithms for large-scale systems. Master's thesis, Delft University of Technology (2012)

[19] van Berkel, S., Turi, D., Pruteanu, A., Dulman, S.: Automatic discovery of algorithms for multi-agent systems. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion. (July 2012) 337–334

[20] Junges, R., Klügl, F.: Evaluation of techniques for a learning-driven modeling methodology in multiagent simulation. In Dix, J., Witteveen, C., eds.: Multiagent System Technologies. Volume 6251 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 185–196

[21] Privošnik, M., Marolt, M., Kavčič, A., Divjak, S.: Construction of cooperative behavior in multi-agent systems. In: Proceedings of the 2nd International Conference on Simulation, Modeling and optimization (ICOSMO 2002), Skiathos, Greece, World Scientific and Engineering Academy and Society (2002) 1451–1453

[22] Husselmann, A.V., Hawick, K.A.: Particle swarm-based meta-optimising on graphical processing units. In: Proc. Int. Conf. on Modelling, Identification and Control (AsiaMIC 2013), Phuket, Thailand, IASTED (10-12 April 2013)

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

[24] Shi, Y., Eberhart, R.: A modified particle swarm optimizer. In: Evolutionary Computation Proceedings. (1998)

[25] Reynolds, C.: Flocks, herds and schools: A distributed behavioral model. In Maureen C. Stone, ed.: SIGRAPH '87: Proc. 14th Annual Conf. on Computer Graphics and Interactive Techniques, ACM (1987) 25–34 ISBN 0-89791-227-6.

[26] Privošnik, M.: Evolutionary optimization of emergent phenomena in multi-agent systems using heuristic approach for fitness evaluation. In: Evolutionary Computation, 2009. CEC '09. IEEE Congress on. (May 2009) 1829–1834

[27] Perumalla, K.S., Aaby, B.G.: Data parallel execution challenges and runtime performance of agent simulations on GPUs. In: SpringSim '08: Proceedings of the 2008 Spring simulation multiconference, New York, NY, USA, ACM (2008) 116–123

[28] Husselmann, A.V., Hawick, K.A.: Multi-stage high performance, self-optimising domain-specific language for spatial agent-based models. In: The 13th IASTED International Conference on Artificial Intelligence and Applications, Innsbruck, Austria, IASTED (February 2014)

[29] Husselmann, A.V., Hawick, K.A.: Towards high performance multi-stage programming for generative agent-based modelling. In: INMS Postgraduate Conference, Massey University. (October 2013)

[30] Husselmann, A.V., Hawick, K., Scogings, C.: Model structure optimisation in lattice-oriented agent-based models. Technical Report CSTN-222, Computer Science, Massey University (2014) Submitted to the International Journal of Modelling and Simulation (ACTA Press).

[31] Taha, W.: Domain-specific languages. In: Pro. Int. Conf. Computer Engineering and Systems (ICCES). (25-27 November 2008) xxiii – xxviii

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

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

[34] NVIDIA: CUDA C Programming Guide. 5.0 edn. (July 2013)

[35] DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J.: Terra: a multi-stage language for high-performance computing. In: PLDI. (2013) 105–116

[36] Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation. Springer (2004) 30–50

[37] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE (2004) 75–86

[38] LLVM: User guide for NVPTX back-end. http://llvm.org/docs/NVPTXUsage.html accessed 12 March, 2014.

[39] Leist, A.: Experiences in Data-Parallel Simulation and Analysis of Complex Systems with Irregular Graph Structures. PhD thesis, Massey University, Auckland, New Zealand (November 2011)

[40] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes. Cambridge University Press (2007)

[41] Koza, J.R.: Genetic Programming: On the programming of computers by means of natural selection. Massachusetts Institute of Technology (1992)

[42] Hawick, K.A., Scogings, C.J., James, H.A.: Defensive spiral emergence in a predator-prey model. Complexity International **12**(msid37) (October 2008) 1–10 ISSN 1320-0682.

[43] Scogings, C.J., Hawick, K.A.: Altruism amongst spatial predator-prey animats. In Bullock, S., Noble, J., Watson, R., Bedau, M., eds.: Proc. 11th Int. Conf. on the Simulation and Synthesis of Living Systems (ALife XI), Winchester, UK, MIT Press (5-8 August 2008) 537–544

[44] Cupertino, L., Silva, C., Dias, D., Pacheco, M.A., Bentes, C.: Evolving CUDA PTX programs by quantum inspired linear genetic programming. In: Proceedings of GECCO'11. (2011)

[45] Chitty, D.M.: Fast parallel genetic programming: multi-core cpu versus many-core GPU. Soft. Comput. **16** (2012) 1795–1814