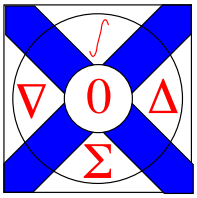


Computational Science Technical Note **CSTN-161**

Particle Swarm-Based Meta-Optimising on Graphical Processing Units

A. V. Husselmann and K. A. Hawick

2013

Optimisation (global minimisation or maximisation) of complex, unknown and nondifferentiable functions is a difficult problem. One solution for this class of problem is the use of meta-heuristic optimisation. This involves the systematic movement of n-vector solutions through n- dimensional parameter space, where each dimension corresponds to a parameter in the function to be optimised. These methods make very little assumptions about the problem. The most advantageous of these is that gradients are not necessary. Population-based methods such as the Particle Swarm Optimiser (PSO) are very effective at solving problems in this domain, as they employ spatial exploration and local solution exploitation in tandem with a stochastic component. Parallel PSOs on Graphical Processing Units (GPUs) allow for much greater system sizes, and a dramatic reduction in compute time. Meta-optimisation presents a further super-optimiser which is used to find appropriate algorithmic parameters for the PSO, however, this practice is often overlooked due to its immense computational expense. We present and discuss a PSO with an overlaid super-optimiser also based on the PSO itself.

Keywords: swarm; optimization; particles; PSO

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-161,
        author = {A. V. Husselmann and K. A. Hawick},
        title = {Particle Swarm-Based Meta-Optimising on Graphical Processing Units},
        booktitle = {Proc. Int. Conf. on Modelling, Identification and Control (AsiaMIC
                2013)},
        year = {2013},
        pages = {104-111},
        address = {Phuket, Thailand},
        month = {10-12 April},
        publisher = {IASTED},
        note = {CSTN-161},
        institution = {Computer Science, Massey University},
        keywords = {swarm; optimization; particles; PSO},
        owner = {kahawick},
        timestamp = {2012.12.01}
}
```


Prof Ken Hawick, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand.

Technical Report CSTN-161

Particle Swarm-based Meta-Optimising 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

October 2012

ABSTRACT

Optimisation (global minimisation or maximisation) of complex, unknown and non-differentiable functions is a difficult problem. One solution for this class of problem is the use of meta-heuristic optimisers. This involves the systematic movement of n-vector solutions through n-dimensional parameter space, where each dimension corresponds to a parameter in the function to be optimised. These methods make very little assumptions about the problem. The most advantageous of these is that gradients are not necessary. Population-based methods such as the Particle Swarm Optimiser (PSO) are very effective at solving problems in this domain, as they employ spatial exploration and local solution exploitation in tandem with a stochastic component. Parallel PSOs on Graphical Processing Units (GPUs) allow for much greater system sizes, and a dramatic reduction in compute time. Meta-optimisation presents a further super-optimiser which is used to find appropriate algorithmic parameters for the PSO, however, this practice is often overlooked due to its immense computational expense. We present and discuss a PSO with an overlaid super-optimiser also based on the PSO itself.

KEY WORDS
swarm; optimization; particles; PSO.

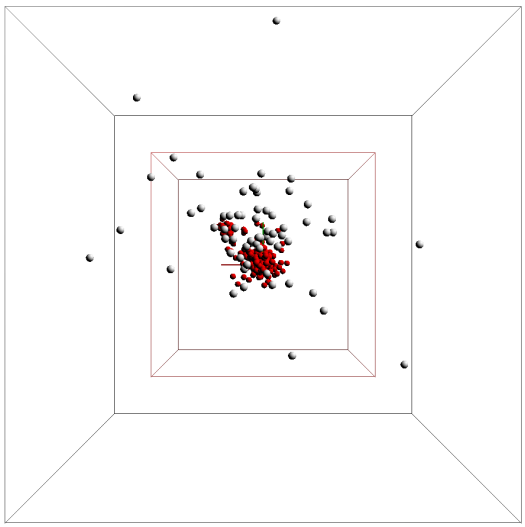


Figure 1: A snapshot of the GPU-based super-optimiser (grey particles) and the end-resulting particle distribution of sub-optimisers (dark-red particles) after several frames of the super-optimiser.

1 Introduction

Computationally minimising a range of unknown, discontinuous, and potentially multi-modal functions is difficult in general. This is especially the case when the parameter space is loosely constrained, and consists of an arbitrary number of dimensions. Such problems present themselves in many industries, and as a result, have drawn widespread interest; especially since Holland first published his Genetic Algorithm [1] in 1975. Some successful applications of parametric optimisers (more specifically, *meta-heuristic* optimisers) include: economics [2, 3], of which the Economic Load Dispatch problem draws much interest; manufacturing process improvement [4], scheduling problems [5], image compression [6] and many in engineering [7–10] mostly involving structural and antenna design.

The Particle Swarm Optimiser (PSO) by Kennedy and Eberhart [11–13] has enjoyed sustained research effort and improvement for many years since its initial publication in 1995. Many variations of the PSO have been proposed, including the Lévy PSO by Richer [14], the Many Optimizing Liaisons PSO [15], the Optimised PSO [16], the Stretched PSO [17] and many more. Like the original, these methods typically involve maintaining a population of "Particles" in n-dimensional parameter space, where each n-vector represents a candidate solution to a optimisation function $f(x)$ with n real-valued parameters.

Large system sizes in interactive Agent-based Models are often the bane of simulation environments, and yet often bring the most interesting aspects of a model to light

[18, 19]. Similarly in the PSO and other meta-heuristic algorithms, large system sizes dramatically improve convergence, especially in highly multi-modal functions. Generally, interactive optimisers which allow particles in parameter space to communicate locally or globally are arguably the most effective and the most expensive simultaneously.

An adequate trade-off between these properties is in the use of parallelism. Parallelism has gained much interest in the Agent-based modelling and parametric optimisation fields due to the inherent parallelism in many of these simulations. For example, in population-based optimisers, particles must be updated one at a time, and then once all have been updated, the process is repeated. By performing the particle update in this loosely synchronised parallel manner, the frame time is dramatically reduced.

The focus of this article is on the population-based meta-optimiser for PSO. To mitigate the extreme compute complexity, we have devised a GPU-based strategy for computing a frame of the super-optimiser, which is also implemented as a PSO.

The article is structured as follows. In Section 2 we discuss the Particle Swarm and the variant we use in this article, and also the parallel implementation of this. Section 3 contains a short introduction on meta-optimisation and how it is applied to population-based optimisers. Then, we present our methodology for combining parallel Particle Swarm Optimisers with meta-optimisation in Section 4. Following this we present results from experiments in Section 5. Finally, we discuss our results and conclude in Sections 6 and 7 respectively.

2 CUDA Particle Swarm Optimisers

The original Particle Swarm Optimiser (PSO) due to Kennedy and Eberhart [11–13] operates by maintaining a population (or "swarm") of particles which are usually constrained within a finite sized hypercube with n dimensions. The significance of the limited size and n dimensions depend on the problem to be solved, which generally involve maximising or minimising an unknown, discontinuous and non-differentiable function with n parameters. In the literature, many test functions have been devised for measuring the efficacy of the numerous meta-heuristic algorithms. Some of these include: Rosenbrock, de Jong (uni-modal); Ackley's Path function, Rastrigin function, the Schwefel function, the Griewangk function, and the Michalewicz function (multi-modal) [20–22]. The difference between multi-modal and uni-modal functions are the complications presented by local optima, which are distinct from the global optimum which one seeks to obtain.

We present here a short introduction into the Particle Swarm. The original PSO was formulated as follows [12]:

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

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

These two simple equations contain the full mechanics of the swarm. The position of a particle in n dimensions is denoted by $\mathbf{x}$, and its velocity moving through the pseudo-space of the optimisation function is denoted by $\mathbf{v}$.

The first term of Equation 1 is known as the inertia term, where $0 < \omega < 1$ represents the weight. The following two terms involve relative vectors to the local and global best solution vectors found so far. The parameters ϕ_p and ϕ_g represent the weights for pursuing local and global best solutions respectively. Finally, r_p and r_g are uniform random deviates in the range of 0 to 1. Equation 2 is simply the application of the new velocity vector to the position vector. Both of these equations are applied to each particle once, for every frame.

The particular PSO variant that we make use of in this article is the Many Optimizing Liaisons PSO (MOL PSO) devised by Pedersen and Chipperfield [15]. This algorithm differs in only a small aspect, but grants a much smaller memory footprint. The difference is simply $\phi_p = 0$, which redefines Equation 1 into Equation 3.

$$\mathbf{v_{i+1}} = \omega \mathbf{v_i} + \phi_g r_g(\mathbf{g} - \mathbf{x_i}) \quad (3)$$

The motivation behind this change is explained by Pedersen and Chipperfield in greater detail. They show in their article that this small change improves the efficacy of the PSO slightly over the original for neural network training [15]. The effect of this change on other problems are not yet clearly known. We use it in our experiments because of its smaller memory footprint, far less random deviate consumption, one less variable parameter to optimise, and as a result of these and the term elimination, greater efficiency. These changes result in the process shown in Algorithm 1.

Algorithm 1 The MOL PSO algorithm.

```
initialise x_i where i = 1..n with d uniform random values
each
with upper and lower bounds u and l.

initialise vector g with random locations in all d dimensions
evaluate the fitness of vector g
for i ← 0 to i_{n-1} do
    calculate the fitness of vector x_i
end for
for i ← 0 to i_{n-1} do
    r_g ← U[0, 1)
    v ← ωv_i + φ_g r_g(g − x_i)
    ensure velocity is within bounds
    x ← x + v
    ensure position vector is within bounds
end for
```

We also use another modification which enhances the exploration behaviour of the MOL PSO. Lévy flights [23] have been shown to enhance exploration and also avoid local minima to some extent. Lévy flights are simply random

walks with a long-tailed probability distribution of step-lengths which produces random deviates with a combination of large and small magnitudes in step lengths. This change is incorporated into the r_g vector in Algorithm 1, where a vector of n gaussian random deviates is normalised and then multiplied by a Lévy deviate for magnitude.

Algorithm 2 Algorithm for generating a Lévy deviate.

```
procedure levydeviate(c, α) begin
double u, v
u = π * (U(0, 1] − 0.5)
//When α = 1, the distribution simplifies to Cauchy
if α == 1 then
  return (c tan u)
end if
v = 0

while v == 0 do
  v = − log(U(0, 1])
end while
//When α = 2, the distribution defaults to Gaussian
if α == 2 then
  return (2c√v sin(u))
end if
//The following is the general Lévy case
return (c sin(αu) / cos(u)^(1/α)) (cos(u(1 − α))/v)^((1−α)/α)
end
```

In Algorithm 2, U is a conventional uniform random deviate generator [24, 25]. This algorithm consumes two uniform random deviates to produce a single Lévy deviate, although there can be special optimisation for the case where $\alpha = 1$, ie. the distribution simplifies to the Cauchy distribution [26]. We must concede however, that the trigonometrical and power functions typically add considerably to the cost of generating these Lévy deviates, even on modern CPUs. Our use of GPUs mitigate this to some extent, enough for us to take advantage of the improved space exploration that it offers. To gain more control over these random movements, we also introduce another parameter in the simulation, $0 < \theta$, which is used to affect the c-parameter of the Lévy distribution.

Parallelising the Particle Swarm Optimiser on CUDA is not extremely difficult. The only complications involve mostly excessive memory requirements, especially for large systems, or in systems where there are many dimensions. For n dimensions, each particle must have n floating point numbers, and a further n floating point numbers for the best position discovered so far by this particle. Then, the particle also requires another n floating point numbers for a velocity vector. For difficult and n-dimensional problems, storage can become an issue very quickly.

With modern GPU hardware such as the CUDA Fermi architecture from NVidia, storage is usually in abundance. Common memory sizes on commodity GPUs are around 1GB of global memory. Tesla GPUs have much more.

The CUDA architecture is a potent arrangement of MIMD and SIMD operating across an array of streaming multiprocessors (SMs). These SMs compute in a parallel MIMD fashion, and they are assigned discrete work units named CUDA "blocks". CUDA blocks are usually arranged in a 2D grid, where individual cells correspond to a separate thread. When executing such a block, the threads are divided into groups of 16, named a "warp", which is then executed in a SIMD fashion on the "CUDA cores" in each SM. This is sometimes referred to as SIMT execution.

There are some idiosyncrasies that require special consideration such as memory access coalescence, and memory hierarchy considerations, but for brevity, we omit an extensive description of this. Essentially, CUDA has a hierarchy of memories available to the application developer, and each memory bank has different scope and access penalties. The general process of executing code directly on the GPU traditionally follows this process: copy data to the GPU global memory bank; perform GPU-specific code (CUDA "kernel"); copy resulting data back to the CPU memory. Recent advances in architecture including host page-locked memory remove the need to perform expensive memory copies between the host and the device.

In particle swarms, each particle effectively maps directly to a CUDA thread, and these threads can efficiently compute the new positions and velocities of each particle in parallel. Typically, an entire frame is computed on the GPU in one kernel call, and then the data is copied back to main memory, and the process repeats. It is possible to improve performance beyond this, such as updating all the dimensions of each particle in parallel as well. This simply involves having a thread for every dimension.

There have been several successful attempts to parallelise the PSO on GPU [27–30]. These include a Beowulf cluster PSO [28], the IPPSO [29], a CUDA PSO [27], and also a Lévy flight CUDA PSO [31]. Clearly there are different ways of parallelising the PSO, and that is ignoring the many variants of the original PSO which can also be parallelised.

We focus our efforts on the use of NVidia's CUDA for parallelising our algorithms. Typically with the architectural restrictions imposed by CUDA, some compromises must be made. One such issue which is not as prominent in serial versions of these algorithms is the problem of random deviates [30]. For the trivial parallelisation of the PSO on GPU, it makes sense to generate random deviates on the host, and copy these onto the GPU in tandem. One must be careful to copy enough random numbers for the GPU to consume.

3 Meta-optimisation

Meta-optimisation is the act of tuning the parameters of an optimiser, and it is an instrinsic aspect of most optimisation efforts [15]. Where optimisers require parameters to guide the stochastic search through parameter space, another (albeit simpler in some cases) optimisation attempt must occur.

Most algorithms require expert tuning to obtain good results. The original PSO required the appropriate tuning of

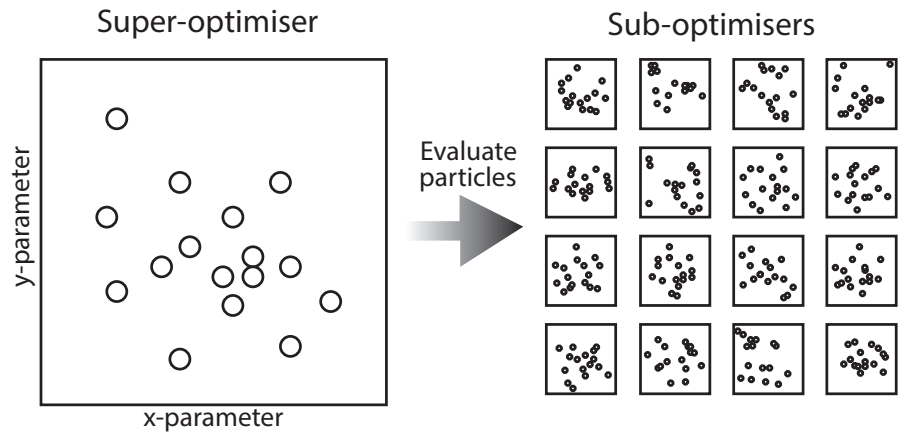


Figure 2: A pictorial representation of the relationship between the meta-optimiser, and the fitness evaluation of its population.

3 separate parameters, the inertial constant begin the most important [17]. Computational meta-optimisation alleviates the time consuming and laborious task of hand-tuning these algorithm parameters. This usually involves using an overlaid super-optimiser, or *meta*-optimiser. Population-based methods are extremely compute-intensive as for every individual agent in the super-optimiser, an entire sub-optimiser must be created and evaluated. Due to the stochastic nature of these algorithms, one often also has to average the result of all the sub-optimisers over many runs of many frames to obtain a reliable indication of the fitnesses of the different set of parameters generated by the super-optimiser. It is for this reason that the super-optimiser is generally better suited to being a trajectory-based method such as Local Unimodal Sampling [32] which maintains only one solution in super-parameter space.

In this context, super-parameter space is the parameter space wherein the super-optimiser moves candidate solutions (or "particles" represented by an n-vector) through space.

Meta-optimisation itself is not a technique restricted to the Particle Swarm in any way. Numerous authors have used trajectory-based, and even population-based meta-optimisers [16] for other algorithms as well, such as the Genetic Algorithm [33], and also the more recent Differential Evolution algorithm [34].

Figure 2 shows the relationship between the super-optimiser and the sub-optimisers. Let x and y be parameters of a hypothetical optimisation algorithm called OA. OA requires the parameters x and y to guide its search through the parameter space of a function to be minimised, in this case, let this function be the two-dimensional Rosenbrock function $f(x, y) = (x - 1)^2 + 100(y - x^2)^2$ [20, 22]. The super-optimiser is set up with 16 particles, each with initially random locations, and then, these particles must be evaluated. The evaluation process is what differentiates meta-optimisation from regular optimisation. Each particle in the super-optimiser has their spatial location coordinates (x and y) propagated into a sub-optimiser, distinct to that super-optimiser's particle, which will now be referred to as a super-particle. The sub-optimiser will execute a certain number of frames in a normal PSO (or other meta-heuristic algorithm) fashion, and then return the average fitness of all its particles in the final frame. This average fitness is then propagated back to the super-particle, and then the super-optimiser will perform a single particle update, and the process is repeated for the next super-particle.

4 Implementation

We combine meta-optimisation with the MOL PSO by using a PSO-based overlaid super-optimiser. With a few exceptions, population-based super-optimisers gain relatively little research interest because of the compute complexity involved. By using CUDA, we mitigate this problem and obtain a real-time meta-optimiser complete with a visualisation of super-particles superimposed on sub-particles to give a good indication of performance.

The super-optimiser is configured for a static 3-parameter (ie. 3D) space, with the principal axes x, y and z representing the sub-optimiser parameters ω, ϕ_g and θ respectively. In our observation, we found that bounds of -4 to 4 for each principal axis was appropriate for the super-optimiser. We define $S(\omega, \phi_g, \theta)$ as the best result obtained by a sub-optimiser with the given parameters. Assume $F(\mathbf{x})$ is the actual function to be optimised.

The super-optimiser was therefore intialised within its bounds with 16 particles, and each particle's spatial location (x, y, z) would correspond to one sub-optimiser, when it comes time to evaluate its fitness by calculating $S(x, y, z)$. To evaluate $S(x, y, z)$, we used a CUDA kernel to calculate 500 frames of the sub-optimiser with the given parameters, and then average this over 20 separate runs. Essentially, evaluating S involves computing 500 frames (with 20 particles), twenty times (re-initialising between each) and computing $F(\mathbf{x})$ exactly $500(20)(20) = 200,000$ times. Bearing in mind that S must be computed exactly once for every 16 super-particles for one frame of the super-optimiser to be completed. As reported by Meissner et al [16], it is neces-

sary to average the results of the sub-optimisers due to the stochastic nature of the search. While 500 frames may be enough to optimise certain functions, it may not be enough in other cases. Increasing the number of frames each internal sub-optimiser calculates causes a dramatic increase in computation time.

The excessive computation involved in this process would certainly bring with it an extreme delay which would render it redundant against the parameter tuning effort of even a novice operator. We use CUDA to mitigate this problem greatly. Each sub-optimiser occupies one CUDA block, which alleviates synchronisation issues. The width of the block (*blockDim.x*) is the number of parameters, including velocities and the fitness value for that particle. The height of the block (*blockDim.y*) corresponds to the number of particles in that block, which we chose to be 20 for our experiments. In essence, we assign a thread to every single data parameter. This does present some limitations however. CUDA block dimensions are limited to 1024 threads maximum, which means that relatively low numbers of parameters and particles are necessary.

We restrict the number of particles in the sub-optimisers to 20, the sub-particles' velocities were restricted to a maximum magnitude of 1.0 (initialised randomly in that range), and we only calculate 300 frames of the super-optimiser. The process was repeated 40 times, and the mean results from the super-optimiser were collected.

The parameters hand-tuned for the super-optimiser are shown in Eq. 4.

$$\omega = 0.9, \;\; \phi = 0.03, \;\; \theta = 4.0, \;\; v_{max} = 1 \qquad (4)$$

5 Results

We present our results for the Meta PSO only, as the fitnesses obtained from the sub-optimisers were themselves a measure of the efficacy of the final result in using a MOL PSO to optimise each of the test functions. In our experience in [35] we observed that small differences in ω and large differences in θ were necessary to optimise the MOL PSO for these different test functions. What the super-optimiser generated had dramatic variations, but as Pedersen and Chipperfield note, they too are undecided on whether there can exist multiple parameter sets which give approximately equal results [15].

Table 1 contains our convergence results for each of our test functions. Table 2 shows the best results produces from all the runs, accompanied by the resulting function value generated. The mean results shown in Table 1 are the result of averaging the best result produced from 40 independent runs of 300 super-optimiser frames consisting of 20 evaluations of $S(\omega, \phi_g, \theta)$ per test function. Effectively these are a good indication as to the ability of the super-optimiser to find good parameters for a sub-optimiser to use in optimising a particular test function.

The Ackley function in 16 dimensions is perhaps the easiest of the test functions to evaluate due to its gentle sloping.

	Lower	Upper	Optimum
Rosenbrock	-2	2	0
Rastrigin	-5	5	0
Schwefel	-500	500	-3351.8632
Ackley	-20	20	0
Griewangk	-600	600	0
Michalewicz	0	$\pi/2$	-4.687

Table 3: The boundaries imposed on the particles in the sub-optimisers and the optimum value for each test function.

However, it is indeed very difficult to obtain the correct parameters to use. The ability of this meta-optimiser to generate parameters (in table 2 in this case) for a MOL PSO which can consistently optimise the Ackley function with just 20 particles is indeed impressive.

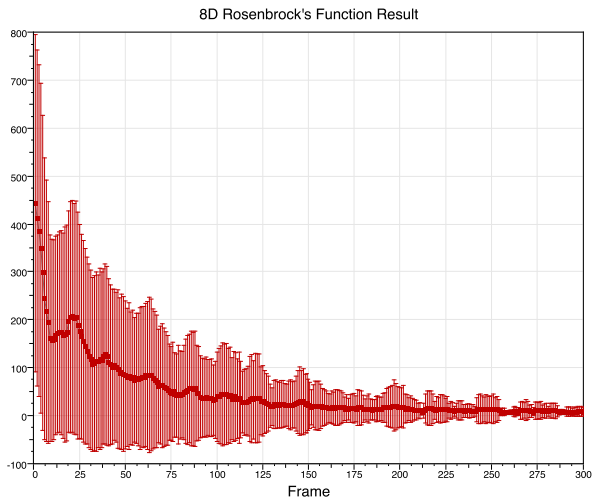
In our previous experiments in [35] using higher quality random movements for exploration, we used ϕ_g values which were much lower than those generated by this algorithm. In our hand-tuned experiments, we normally set ϕ_g in the range $[0.01, 0.1]$. The large values of ϕ_g and ω for the Schwefel and Griewangk functions were expected somewhat, due to the very large search space accompanied with these functions (these are shown in Table 3). This is made clearer in the discussion of performance graphs in Fig. 3.

The great freedom allowed in the generous bounds of each of ω, ϕ_g and θ allowed the algorithm to seek out unconventional parameters. The θ chosen for the Schwefel function reflects this. For ease of reference, the θ variable corresponds to the c variable of the Lévy distribution which we use for higher quality exploration [35]. For brevity, this value simply corresponds to the magnitude of the Lévy distribution. The large value can also be attributed to the need to move quickly through the huge parameter space in order to obtain a better solution in less than the maximum 500 frames of the sub-optimiser.

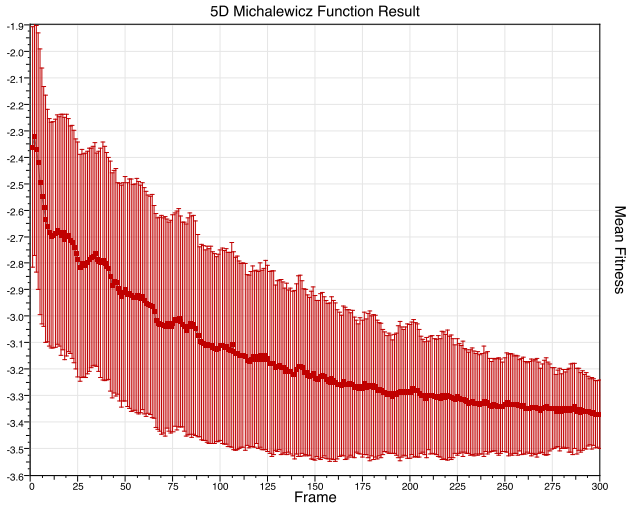
Apart from the huge parameter spaces of the Schwefel and Griewangk functions, it seems that the ω parameter plays a smaller role than previously thought. Similar observations were made by Pedersen and Chipperfield [15], albeit, for a different application.

Figure 3 shows plots of the means for each test function. Each plot represents one full 300-frame execution of a super-optimiser, where each frame corresponds to the average best result from computing $S(\omega, \phi_g, \theta)$ of 16 different (ω, ϕ_g, θ) combinations, 20 times. The fitness axis represents the mean $F(\mathbf{x})$ result obtained, and these values are the result of averaging the means obtained from all 40 runs for every frame. This gives us a good indication of the performance of the generated parameters from the super-optimiser on every frame. The error bars represent the mean of the standard deviations of the result for every frame.

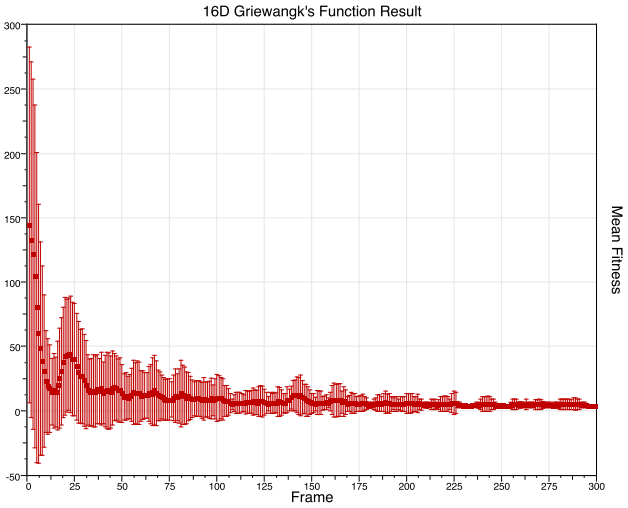
The plot of the Schwefel function's optimisation attempt is a conclusive indication of the failure of sub-optimisers to optimise the test function, regardless of the (ω, ϕ_g, θ) parameter combinations attempted. This is clear because the standard



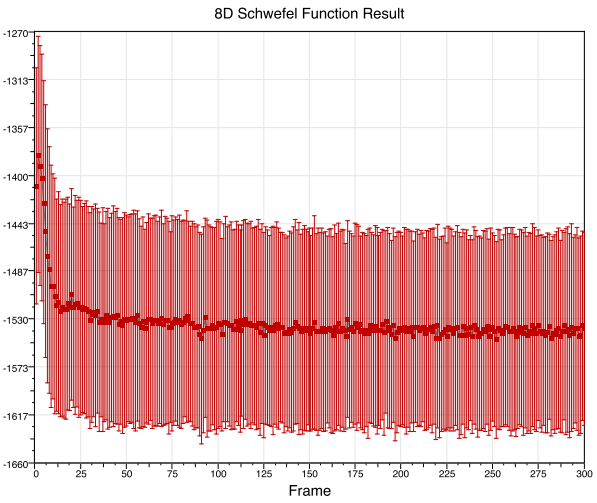
(a) Meta-optimiser on Rosenbrock's function.



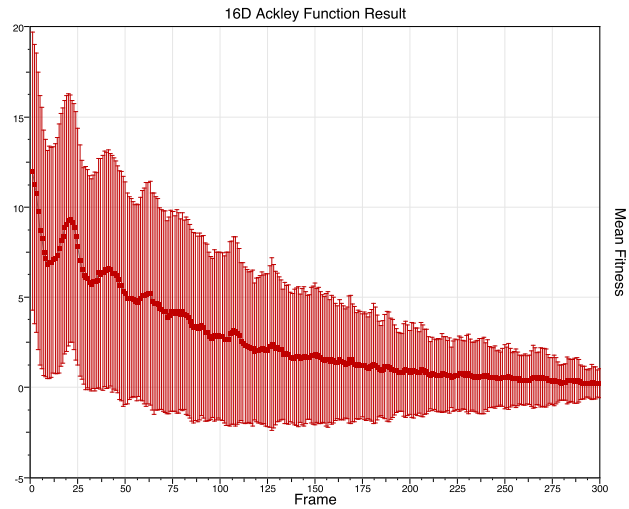
(b) Meta-optimiser on Michalewicz's function.



(c) Meta-optimiser on Griewangk's function.



(d) Meta-optimiser on Schwefel's function.



(e) Meta-optimiser on Ackley's Path function.

Figure 3: Plots of the mean fitness obtained from the super-optimiser across all 40 separate runs, including error bars representing the average standard deviations of each point (sub-optimiser) across the separate runs.

	Rosenbrock 8D	Schwefel 8D	Griewangk 16D	Michalewicz 5D	Ackley 16D
Meta-PSO					
Mean Result	4.05	-1941	2.3	-3.69	0.0
Std. Dev.	0.14	32.4	0.13	0.011	0.0

Table 1: The mean result generated by our PSO-PSO meta optimiser for several test functions. Low values denote higher quality solutions. All functions had an optimum result of 0, except for the Schwefel function (-3351.8632) and the Michalewicz function (-4.687).

	Rosenbrock 8D	Schwefel 8D	Griewangk 16D	Michalewicz 5D	Ackley 16D
ω	0.2557	3.8690	1.7098	-0.3540	0.6784
ϕ_g	2.1863	3.0114	4.0	0.3332	0.4677
θ	0.7154	4.000	3.5909	3.8098	0.3988
$F(\mathbf{x})$	3.7471	-2021.1965	1.9462	-3.6953	0.0000

Table 2: Best sub-optimiser parameters generated by the meta-optimiser throughout the 40 separate runs.

deviations are not decreasing, and therefore, no convergence is apparent. The Schwefel function is too difficult for a small PSO of just 20 particles to optimise, due to the sheer size of the parameter space. The initial improvement from frames 0 to 100 indicate that the results generated are still better than a random search, however.

Through each of the plots shown, a decreasing sine-wave seems to be a common characteristic. This can be attributed to the inertia weight of the super-optimiser. Overshoot occurs as a result of the inertial movement of the super-particles, and as a result, this is reflected in the oscillating means.

6 Discussion

While it seems that there is indeed benefits in using meta-optimisation in order to obtain good optimisers, it is still very much a computationally expensive process. Our sub-optimisers were limited to low dimensions in the test functions (5D to 16D), due to the CUDA block size limitations. In order to overcome this limitation, one has to make use of inter-block synchronisation, which is not supported by CUDA and goes against the architectural design; even though some attempts have been made in the past. Apart from re-using the same thread to compute the velocity and position of the same index, not much can be done to improve the process.

The results generated seem to take advantage of the individual characteristics of the test functions that they are assigned, which is, in usual circumstances, an undesirable effect. Generally, meta-heuristics which are able to optimise wide ranging distinctive functions are deemed to be "better" optimisers; so one must have a very good reason to use a meta-optimiser on only one test function. It seems that given the excellent compute power of the GPU that a meta-optimiser such as the one described here may be used as the optimiser itself, as it generates solutions as well as effective optimisers.

It is possible to use a meta-optimiser for generating optimisers which are good for solving a specific set of problems as well. This is done by modifying the fitness function to find the mean of a set of test functions. However, it is then the impossible task of the operator to define the set of test functions which are to be deemed representative of all test functions that may be optimised. Therefore, it is still very much an art to determine the correct optimiser for a particular task.

The focus of this article was to explore the efficacy of super-optimisers based on the MOL PSO. To enable us to obtain meaningful statistical results, we devised a CUDA-based algorithm in order to compute much faster, and this has led us to further contemplate the role of GPUs and population-based meta-optimisation. The computation time of the algorithm rarely reaches into a minute for a full run, depending of course on the test function being used.

7 Conclusion

We have presented a PSO-based meta-optimiser capable of generating appropriate parameter values for optimisers based on the PSO. Our use of NVidia's CUDA streamlines the process considerably as opposed to single-threaded meta-optimisers. Population-based super-optimisers has received little attention due to the excessive computational expense; but through CUDA, we successfully mitigated this problem to obtain the benefits of the PSO as a super-optimiser.

The parameters generated by the super-optimiser were not always intuitive, but seemed to take advantage of the individual characteristics of the test functions. The value of this effect depends on the application.

In conclusion, population-based meta-optimisation has become a viable option thanks to the data-parallel architecture of CUDA, and there is some scope for further work in this area; particularly the use of the Local Unimodal Sampling method in a more parallel environment to further make

use of the incredible processing power made available by CUDA. The notion of a hierarchy of meta-optimisers appears to have great promise and there is also scope to extend the hierarchy with further levels as well as with hybrid algorithms.

References

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

[2] Yang, X.S.: Firefly algorithm for solving non-convex economic dispatch problems with valve loading effect. Applied Soft Computing **12** (2012) 1180–1186

[3] Apostolopoulos, Vlachos: Application of the firefly algorithm for solving the economic emissions load dispatch problem. International Journal of Combinatorics **1** (2011)

[4] Aungkulanon, P., Chai-ead, N., Luangpaiboon, P.: Simulated manufacturing process improvement via particle swarm optimisation and firefly algorithms. Prof. Int. Multiconference of Engineers and Computer Scientists **2** (2011) 1123–1128

[5] Khadwilard, A., Chansombat, S., Thepphakorn, T., Thapatsuwan, P., Chainate, W., Pongcharoen, P.: Application of firefly algorithm and its parameter setting for job shop scheduling. First Symposium on Hands-On Research and Development **1** (2011)

[6] Horng, M.H.: Vector quantization using the firefly algorithm for image compression. Expert Systems with Applications **38** (2011)

[7] Azad, S.K., Azad, S.K.: Optimum design of structures using an improved firefly algorithm. International Journal of Optimization in Civil Engineering **1** (2011) 327–340

[8] Gandomi, A.H., Yang, X.S.: Mixed variable structural optimization using firefly algorithm. Computers and Structures **89** (2011) 2325–2336

[9] Chatterjee, A., Mahanti, G.K., Chatterjee, A.: Design of a fully digital controlled reconfigurable switched beam coconcentric ring array antenna using firefly and particle swarm optimization algorithms. Progress in Electromagnetic Research **B** (2012) 113–131

[10] Basu, B., Mahanti, G.K.: Firefly and artificial bees colony algorithm for synthesis of scanned and broadside linear array antenna. Progress in Electromagnetic Research **32** (2011) 169–190

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

[12] Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: Proc. IEEE Int. Conf. on Neural Networks, Piscataway, NJ. USA (1995) 1942–1948

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

[14] Richer, T.J.: The levy particle swarm. In: IEEE Congress on Evolutionary Computation. (2006)

[15] Pedersen, M.E.H., Chipperfield, A.J.: Simplifying particle swarm optimization. Applied Soft Computing **10** (2009) 618–628

[16] Meissner, M., Schmuker, M., Schneider, G.: Optimized particle swarm optimization (opso) and its application to artificial neural network training. BMC Bioinformatics **7** (2006)

[17] Parsopoulos, K.E., Vrahatis, M.N.: Recent approaches to global optimization problems through particle swarm optimization. Natural Computing **1** (2002) 235–306

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

[19] Husselmann, A.V., Hawick, K.A.: Parallel parametric optimisation with firefly algorithms on graphical processing units. Technical Report CSTN-141, Computer Science, Massey University (2012) accepted for and to appear in Proc. International Conference on Genetic and Evolutionary Methods (GEM'12), 16-19 July, 2012, Las Vegas, USA.

[20] Rosenbrock, H.H.: An automatic method for finding the greatest or least value of a function. The Computer Journal **3** (1960) 175–184

[21] Molga, M., Smutnicki, C.: Test functions for optimization needs. Technical report, Univ. Wroclaw, Poland (2005)

[22] Yang, X.S.: Test problems in optimization. In: Engineering Optimization: An Introduction with Metaheuristic Applications. Wiley (2010)

[23] Mandelbrot, B.B.: The Fractal Geometry of Nature. W.H. Freeman (1982)

[24] Preez, V.D., M.G.B.Johnson, A.Leist, K.A.Hawick: Performance and quality of random number generators. In: International Conference on Foundations of Computer Science (FCS'11). Number FCS4818, Las vegas, USA (July 2011)

[25] Coddington, P., Mathew, J., Hawick, K.: Interfaces and implementations of random number generators for java grande applications. In: Proc. High Performance Computing and Networks (HPCN), Europe 99, Amsterdam. (April 1999)

[26] Chambers, J.M., Mallows, C.L., Stuck, B.W.: A method for simulating stable random variables. Journal of the American Statistical Association **71** (1976) 340–344

[27] Mussi, L., Cagnoni, S., Daolio, F.: Gpu-based road sign detection using particle swarm optimization. In: Proc. Ninth Int. Conf. on Intelligent Design and Applications (ISDA'09), Pisa, Italy (30 November 2009) 152–157

[28] Schutte, J.F., Fregly, B.J., Haftka, R.T., George, A.D.: A parallel particle swarm optimizer. Technical report, University of Florida, Department of Electrical and Computer Engineering, Gainesville, FL, 32611 (2003)

[29] Hung, Y., Wang, W.: Accelerating parallel particle swarm optimization via gpu. Optimization Methods and Software **27**(1) (February 2012) 33–51

[30] Bastos-Filho, C., Junior, M.O., Nascimento, D.: Running particle swarm optimization on graphic processing units. In Mansour, N., ed.: Search Algorithms and Applications. In-Tech (2011) ISBN: 978-953-307-156-5.

[31] Husselmann, A., Hawick, K.: Lévy flights for particle swarm optimisation algorithms on graphical processing units. Technical Report CSTN-161, Computer Science, Massey University (July 2012)

[32] Pedersen, M.E.H.: Tuning & Simplifying Heuristical Optimization. PhD thesis, University of Southampton (2010)

[33] Kantschik, W., Dittrich, P., Brameier, M., Banzhaf, W.: Meta-evolution in graph gp. Lecture Notes in Computer Science **1598** (1999) 15–28

[34] Pedersen, M.E.H.: Good parameters for differential evolution, hl1002. Technical report, Hvass Laboratories (2010)

[35] Husselmann, A.V., Hawick, K.A.: Random flights for particle swarm optimisers. Technical Report CSTN-160, Computer Science, Massey University (July 2012)