GPU Accelerated Genetic Algorithm and Solver Information

Version 1.1

Dr. Matthew Smith, msmith@astro.swin.edu.au

ADACS @ Swinburne University of Technology

Algorithm Description

The algorithm is outlined in the paper below:

Smith, M.R., Kuo, F.-A., Hsieh, C.-W., Yu, J.-P., Wu, J.-S. and Ferguson, A., Rapid optimization of blast wave mitigation strategies using Quiet Direct Simulation and Genetic Algorithm, Computer Physics Communications, 181[6], 2010.

A brief summary of the approach used is as follows: consider a parameter X which contributes in some unknown way to a fitness function F(X). A population of N children each store their own value of X - unique to each child. The N children together make up a single generation of children - the characteristics of each child (i.e. the values of X) are used in the creation of new generation of children, loosely based around the ideas of genetics.

In the case where we wish to maximize the value of the fitness, one of these children will have a higher value than the others - we shall call this X_1. When creating new children, we choose another parent randomly from the population - X_2 - and a third parent is chosen for the sake of measuring the populations' current variance (X_3). When combined, the value of the new child's X is:

$$X_c = GF\ X_1 + (1 - GF)X_2 + \sigma R_N(X_2 - X_3)$$

where GF is:

$$GF = FR * \max(R_f, 1 - R_f)$$

where Rf is a uniformly generated random fraction (between 0 and 1) and RN is a normally distributed random variable from a normal distribution with a mean of 0 and a variance of 1. The value of FR - normally equal to one - may be used to accelerate convergence by increasing its value, while sigma - attached to the mutation term - may be increased to increase stability of convergence.

The fitness is then computed for each child; if it is found to be an improvement over that parent previously occupying its space in memory, we choose its value of X and associated fitness and save them as part of the next generation information. Otherwise, the parent information is passed on. This process is repeated across all individuals from generation to generation, the result being generations of children which:

- Have higher measures of fitness (i.e. are better), and
- Demonstrate decreasing variation in X from generation to generation.

A benchmark is built-into the code for demonstration purposes - specifically Ackley's function, which is (for two variables):

$$f(x,y) = -20\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right) - \exp\left(0.5(\cos(2\pi x) + \cos(2\pi y))\right) + \exp(1) + 20$$

This function has many local minima which make it challenging to solve using conventional root finding algorithms. It may also be expanded to include multiple variables.

The graphics shown in Figure 1 show a contour plot of the two-variable Ackley's function as described above, while the convergence history for a 3 variable problem is also shown.

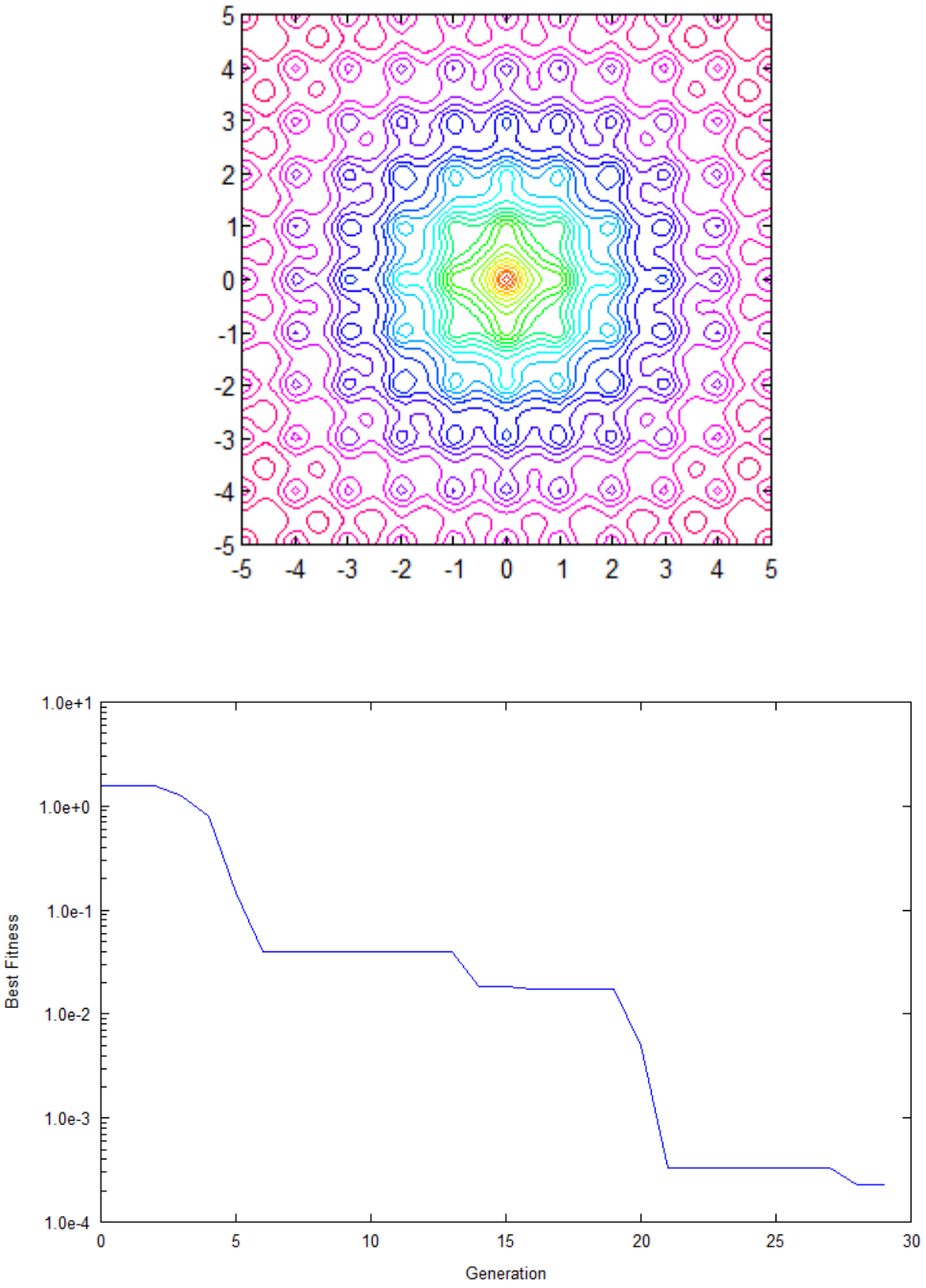


Figure 1: [Top] Contour function of the two-variable Ackley's function showing the single root (i.e. lowest value) at x,y = 0,0 where f(x,y) = 0. [Bottom] Fitness function vs generation for a three variable Ackley function using FR = 1.0 and SIGMA =

Key Code Variables

A brief description of key variables and constants is found below.

NOTE: In general,

- variables beginning with d_ are stored in GPU memory while h_ are stored in host (CPU) memory.
- UPPER_CASE constants are #defined values which are replaced at compile time.

config.h

NO_GEN	No. of Genetic Algorithm generations to run. Increase if problem is not converged.
TPB	Threads Per Block executed within each Streaming Multiprocessor on the GPU
BGP	The number of thread blocks to use on the device.
NO_KIDS	Number of children in each generation of the GA. Currently set such that each streaming processor in use on the GPU controls a single child.
NO_VAR	Number of parameters required to be solved in our optimization problem.
DEBUG	Set to 1 to turn on CUDA and debugging messages during runtime.
FR	Over-convergence parameter on crossover computation. Standard value is 1.0; can be increased to accelerate convergence, which also creates dangers.
SIG	Mutation parameter, larger values encourage larger mutations and prevent capture in local minima while also slowing convergence.
NO_TESTS	Used for ensemble testing of the GA method and analysis of convergence. For a single simulation, this is 1. Currently (default) is 1000.

gpu_main.cu

_x	One dimensional arrays holding our solution parameters. (d_x for GPU, h_x for host)
_x_new	New (next generation) values of x. Computed and stored between generations.
_fitness	Current fitness as computed according to the values
_fitness_new	New (next generation) values of fitness, computed according to x_new.
x_min, m_max	Two arrays of length NO_VAR which contain the minimum and maximum values of x for each variable. These are only used during initialization; they are not used to enforce the ranges of x during computation.
_fitness_history	History of the best fitness, one dimensional array NO_GEN in length.
_x_history	History of the best values of x, one dimensional array of length NO_GEN*NO_VAR.
_BestKid	Index (i.e. name) of current best child in its generation. Integer.
_BestFitness	The fitness of the current best child in its generation.
_state	cuRAND random number generator state; each thread requires its own state.
_avg_fitness	Used in ensemble testing; this holds the best fitness for each ensemble.

Data Structure

The solver accommodates for NO_VAR variables / parameters in our optimization problem, which is stored across NO_KIDS children. This genetic information is stored inside a one dimensional array of length (NO_VAR*NO_KIDS). A figure demonstrating how each variable is stored for each child in the one dimensional array is shown below:

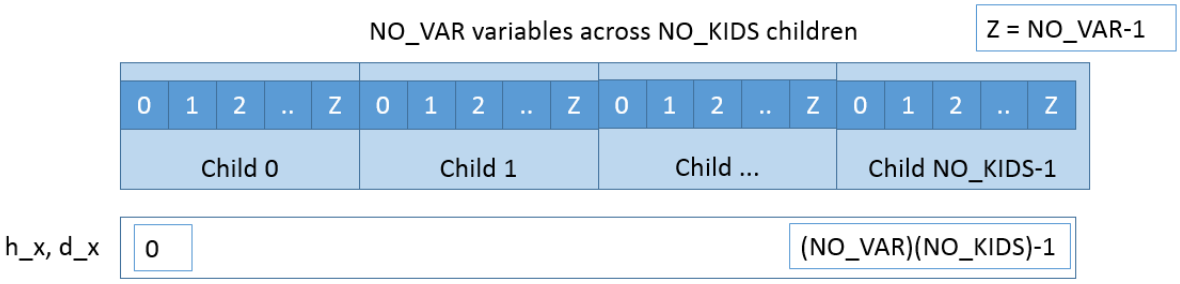


Figure 2: Picture describing how solution data is held in one-dimensional memory for NO_KIDS children and NO_VAR solution parameters.

The majority of other variables - d_fitness, for example - are one dimensional arrays of length NO_KIDS which hold single values for each child, i.e. the fitness for child j is held in the jth element of the fitness array (d_fitness[j]).

Function Descriptions

Host Functions / Wrapping Functions

Allocate_Memory()	Allocate memory on host and device. Perform prior to computation.
Free_Memory()	Free memory on host and device. Last step prior to exit() of program.
ComputeFitness(float *x, int N)	Compute fitness on CPU using CPU held data.
ComputeNewGenCall()	Call GPU kernel (ComputeNewGenerationGPU) which computes the new fitness and new values of x.
UpdateNextGeneration()	Overwrites values of _x and _fitness on device with _x_new and _fitness_new.
FindBestGPUCall()	Calls GPU kernel (FindBestGPU) which uses a single block with MAX_TPB threads to find the current best child.
SetupRfGPUCall()	Calls GPU kernel (SetupRFGPU) which initializes the random number generator engine state for each thread.
SetupGPUCall()	Calls GPU kernel (SetupGPU) which primarily computes the fitness for each child in the starting (0^{th}) generation.
Init()	Creates the initial values of x (solution parameters) for each child on the host, and saves them to file.

Randf()	Generate a random fraction between 0 and 1.
RandNf()	Generate a normally distributed random variable from a normal distribution with a mean of 0 and a variance of 1.
SaveFitness()	Saves the fitness data for each child after NO_GEN generations. Also saves to file the history data for analysis of its convergence.

GPU Kernels / Device Functions

MaxFrac()	Device function (single thread). Generates the maximum of A or (1-A).
ComputeFitnessGPU()	Device kernel (multi-thread). Computes the fitness for each thread using the parameters contained in d_x. Note: This is the part of the code which primarily needs to be modified in order to customize this code to specific applications. Note: The code currently solves minimization problems.

ComputeNewGenerationGPU()	Device kernel (multi-threaded). Compute the next generation value for each thread (i.e. child) in the current generation.
	
	Note: We assume here that a lower fitness value is optimal, i.e. minimization. In the event maximization is required, the line of code which reads:
	
	if (fabs(fitness_new[i] < fabs(fitness[i])) {
	
	should be changed to read:
	
	if (fabs(fitness_new[i]) > fabs(fitness[i])) {

SetupGPU()	Device kernel (multi-threaded). Computes the fitness for each child using d_x, and initializes the on-device generation counter (d_Step).
SetupRfGPU()	Device kernel (multi-threaded). Initializes the CUDA random number generator state for each thread.
FindBestGPU()	Device kernel (multi-threaded). Compute the best child in the current generation using info held in d_fitness. Also saves the fitness history for each generation and increments the d_Step variable.
TicGPU(), TocGPU()	Functions used for timing the GA computations. Current implementation has two levels of timing; one using CUDA timers, the other using a gettimeofday() approach.

Contact Information

Name: Dr. Matthew Smith

Address: Center for Astrophysics and Supercomputing (CAS), Swinburne University of Technology.

Email: msmith@astro.swin.edu.au

Installing in Linux

Required Drivers

CUDA and CUDA-capable drivers are required. You can check to see if CUDA is working by calling the NVCC (CUDA) compiler - type “nvcc” and press enter - if CUDA is properly installed, you should see the following message:

```
[                              ]$ nvcc
nvcc fatal   : No input files specified; use option --help for more information
```

Download, Building and Executing

You can download the code manually, or from a linux terminal, use git to clone. From your home directory, type:

git clone https://github.com/archemabud/GAGPU

This will create a new directory (GAGPU) in your home directory. Move into the directory and make the executable:

cd GAGPU <enter>
make <enter>

This will create the executable (GAGPU.run). To run this program, type:

./GAGPU.run <enter>

If DEBUG (inside config.h) is set to 0, you should see something like this:

```
*** Ensemble Average 997 of 1000 ***
Size of x_min() and x_max() arrays = OK
CUDA Elapsed time = 675.072 microseconds
CPU Elapsed time = 676 microseconds
Best Index = 0, Best Fitness = 0
X Values for best = 3.63859e-07, 1.81323e-07,
*** Ensemble Average 998 of 1000 ***
Size of x_min() and x_max() arrays = OK
CUDA Elapsed time = 673.632 microseconds
CPU Elapsed time = 675 microseconds
Best Index = 0, Best Fitness = 0
X Values for best = 3.1201e-07, -1.02122e-07,
*** Ensemble Average 999 of 1000 ***
Size of x_min() and x_max() arrays = OK
CUDA Elapsed time = 674.592 microseconds
CPU Elapsed time = 676 microseconds
Best Index = 0, Best Fitness = 0
X Values for best = 2.72103e-07, -8.53891e-08,
```

Current Ensemble (Run)

Best Fitness in current ensemble

Associated X values (here for a two parameter problem)

Processing Results

After executing, GAGPU.run will create a couple of text files containing results:

History.txt Contains the best fitness and associated X values for each generation 1 to NO_GEN.

TestHist.txt Contains the best fitness for each ensemble for each generation.

Increasing generation from 1 to NO_GEN

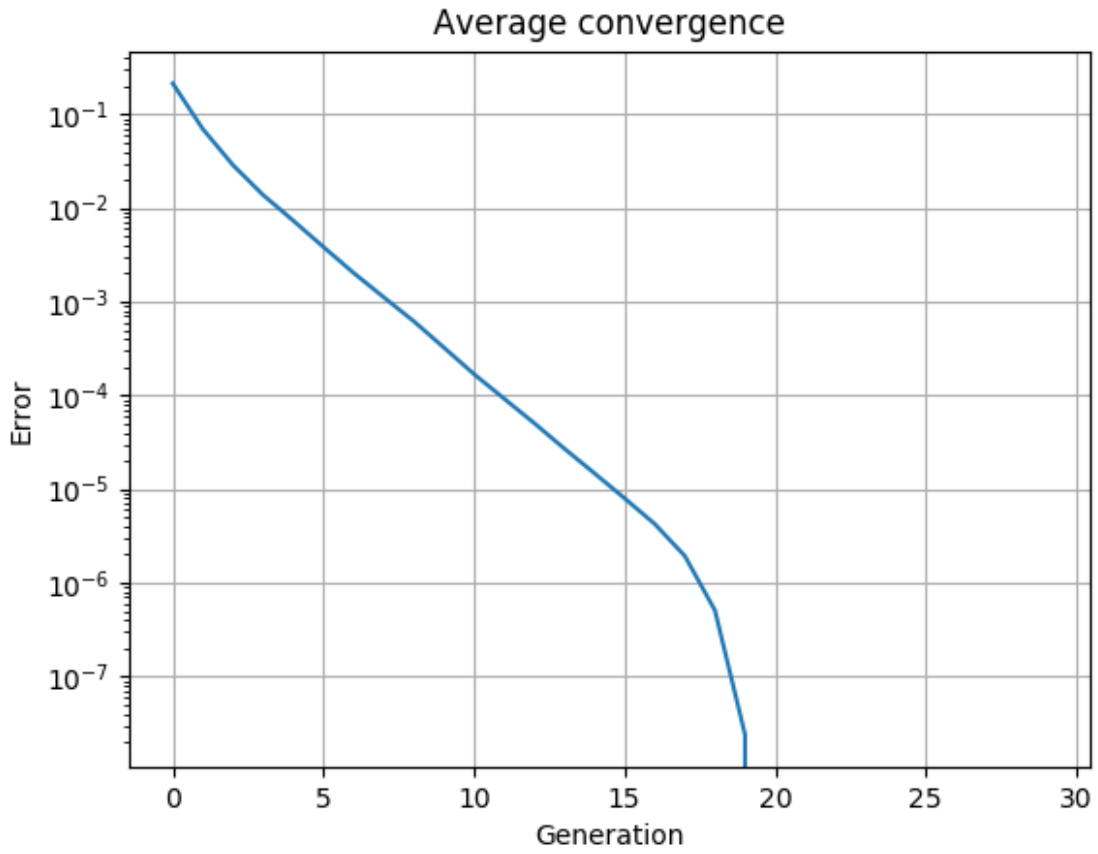
```
  GNU nano 2.3.1              File: TestHist.txt

0.0283716       0.00366187      0.00366187      0.00366187      0.00366187
0.0283716       0.0283716       0.0283716       0.00956678      0.00955915
0.0283716       0.0283716       0.0283716       0.0223453       0.0159574
0.0283716       0.0283716       0.0255365       0.0154994       0.0154994
0.0283716       0.0283716       0.0283716       0.0171421       0.014878
0.0283716       0.0283716       0.0283716       0.00930738      0.00930738
0.0283716       0.0283716       0.0162821       0.0112493       0.00684762
0.0283716       0.0283716       0.0194948       0.0194948       0.0128024
0.0283716       0.0283716       0.0282536       0.00980353      0.00721812
0.0283716       0.0253365       0.0253365       0.0140378       0.00487542
0.0283716       0.0283716       0.0283716       0.0234027       0.0100956
0.0283716       0.0283716       0.0130155       0.00350928      0.00275707
0.0283716       0.0283716       0.0283716       0.0104074       0.0104074
0.0283716       0.0283716       0.0283716       0.0159218       0.0128205
0.0283716       0.0283716       0.0283716       0.0257852       0.0180528
0.0283716       0.0283716       0.0220354       0.00674677      0.00674677
```

Increasing test (ensemble) from 1 to NO_TESTS

This data may be analyzed as required by the user, or plotted. There is a python code included with the repository called loadresults2.py, which can be called using:

python loadresults2.py <enter>

This will compute the average best fitness for each generation, and will create a figure like this:



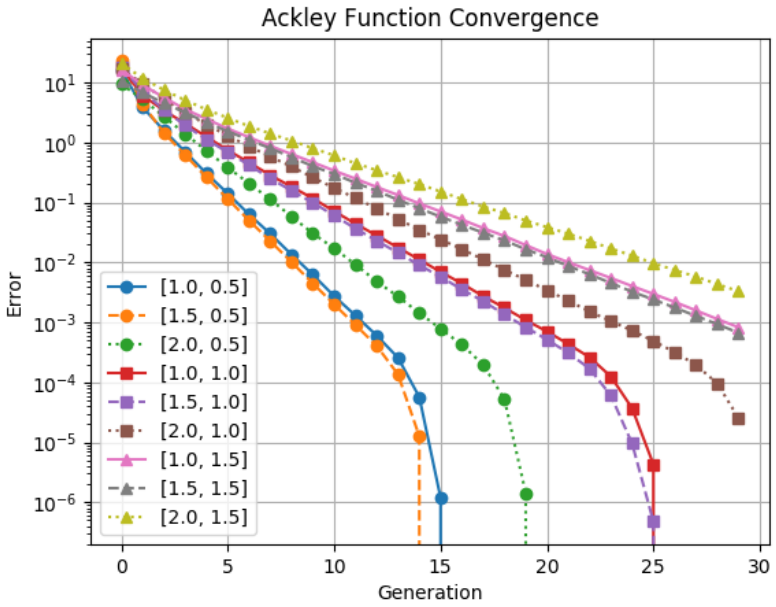
NOTE: This requires **python** to be installed, together with the **numpy** and **matplotlib** libraries. If you are running this from a server, you will need to ensure X11 information can be passed to your computer. Windows users can use Xming if connected to Linux servers for this purpose.

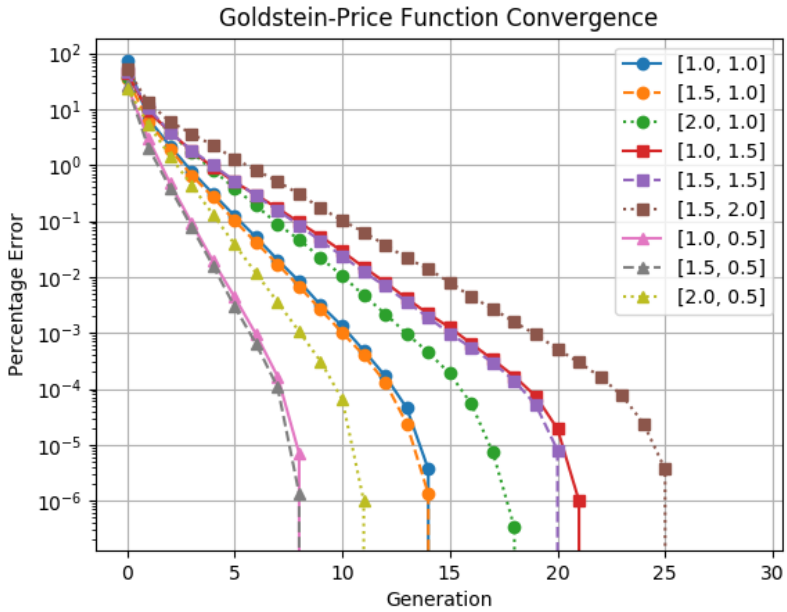
Testing

Several commonly used benchmarks are included with the code. These benchmarks are found in the __device__ function (ComputeFitnessGPU()) inside the gpu_main.cu file, and are:

- Ackley's Function, with a global minima of 0 located at [0,0] (for two parameters), and
- Goldstein-Price function, with a global minima of 3 located at [0, -1].

The convergence properties of these two benchmark tests for the public release versions of this code are shown below:





where each test uses the default number of children and NO_TESTS = 1000. The first parameter in the legend is the value of FR while the second is SIGMA - both these parameters are found in config.h. If these are changed, you need to rebuild your GAGPU.run file for these changes to take effect.