Genetic optimization using Differential Evolution with GPU acceleration

Pawel Wzietek, Feb 2025
pawel.wzietek@universite-paris-saclay.fr

1 Introduction

Many years ago, while searching how to solve my fitting problems, I stumbled on a fitting procedure using a genetic algorithm, posted by Andrew Nelson. The current version can be found here: `https://www.wavemetrics.com/project/gencurvefit`. This version is in XOP form, however at the time I had discovered it was all in an ipf file and I started using it adding my own modifications. At the same time I had some complex fits that were taking hours and days to run and I was thinking how to speed up the calculations (actually the critical part was in my code calculating the curves, so that using the XOP version instead of ipf would not help much). Then, some time later, I discovered this project:
`https://www.wavemetrics.com/project/IgorCL`
This XOP by Peter Dedecker (also at https://github.com/pdedecker/IgorCL, GPL license) provides an Igor interface to the OpenCL framework allowing to write and call functions running on GPU. I then started to adapt the genetic code to be able to use this interface. Recently I have asked Andrew his permission to publish my (quite a lot) modified version of his original code, and so here it is.

To understand what this code does and where the OpenCL coding can be helpful, let me first shortly recall some curve fitting basics. Consider a set of experimental data $\{x_i, y_i\}$ and a model $f(x, \mathbf{c})$ parametrized by a set of coefficients $\mathbf{c} \equiv \{c_1, c_2, ...\}$. The fitting consists in finding the set $\mathbf{c}$ that maximizes the probability of having obtained the experimental values $\{y_i(x_i)\}$ (*maximum likelihood estimator*). If we suppose that the noise in the data follows the normal (gaussian) distribution it is easy to show that to maximize this probability we need to minimize the fitness function

$$\chi^2(\mathbf{c}) = \sum_i \left(\frac{y_i - f(x_i, \mathbf{c})}{\sigma_i} \right)^2$$

in the configuration space $\mathbf{c} = \{c_k\}$ ("least squares method"). The fitting task thus boils down to the problem of function minimization in multidimensional space. Note that normal distribution cannot always be assumed (experimental artifacts etc.), therefore sometimes it is better to consider some other form of χ^2 (e.g. "robust fitting" where the square is replaced by the absolute value).

Usually fitting routines and packages take care of the χ^2 calculation so that we supply the dataset and the model function and run the fit (e.g. CurveFit and FuncFit in Igor).

The standard and widely used least-squares fitting routine is based on the Marquardt-Levenberg (ML) minimization algorithm. This algorithm uses the derivatives (gradient

and second derivatives) of the fitness function to find a *local* minimum nearest to a given starting point and is optimized for the least squares problem. "Local" means that the solution may depend on the starting point, hence it is important to select this point carefully ("guesses" for the fit coefficients). For simple models with few parameters it is easy to find the right guesses however it may be difficult for more complex functions with many parameters or when the fitness function has many local minima. For such cases the nearest local minimum is not a guarantee of the best fit, in fact it is often very difficult to find a starting point such that the algorithm will converge to the true solution.

For more complex fits we would like to be able to find the *global* minimum of the fitness function, however this is more difficult as there is no deterministic algorithm and only heuristic methods are available. A large class of such methods are the so called *evolutionary algorithms.* Here, instead of moving a point in the configuration space towards the minimum, we consider a population of points (trial vectors) submitted to some evolution strategy. The method called *differential evolution* (DE) which mimics, in some way, the darwinian evolution, is considered as one of the most powerful among them.

Here are some references for the DE algorithm and its applications:

[1] R. Storn and K. Price, Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces, Journal of global optimization 11, 341 (1997).
(original algorithm by Storn and Price)

[2] S. Das and P. N. Suganthan, Differential evolution: A survey of the state-of-the-art, IEEE transactions on evolutionary computation 15, 4 (2010).
(Review of different strategies and comparison with other methods.)

[3] M. Wormington, C. Panaccione, K. M. Matney, and D. K.Bowen, Characterization of structures from x-ray scattering data using genetic algorithms, Philosophical Transactions of the Royal Society of London, Series A 357, 2827 (1999).
(applications in x-ray studies)

[4] https://en.wikipedia.org/wiki/Differential_evolution

Evolutionary algorithms are rather computationally expensive, a DE fit will usually require many times more function evaluations compared to the classical ML approach (however, as I already said, in complex cases we may have no choice). On the other hand, evaluating the fitness function for a population of points in the configuration space can be easily parallelized since the calculations for different points are independent of each other. This independence is especially attractive in the context of GPU programming where the means of thread synchronization are very limited.

Adaptation for parallel processing and OpenCL

The Figure 1 shows the outline of the DE algorithm. At each iteration a new population of trial points is generated via mutation and genetic recombination, then better candidates are retained in the offspring population. The process is repeated until a convergence criterion is met. The DE algorithm is constructed in such a way that the whole population will converge towards a single point, considered as the final solution.
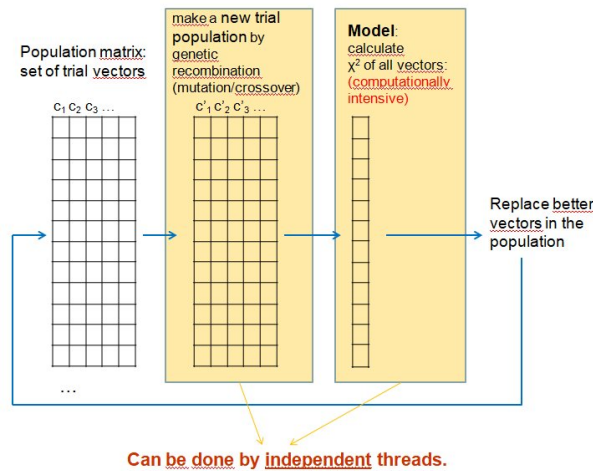
Figure 1: Differential Evolution

The Igor optimization function proposed here (see reference in the last section) performs all steps of the algorithm except the χ^2 calculation on the population, for which a user-defined function has to be supplied. The generation of trial vectors can be done either with Igor code or on GPU (see function parameters in the reference).

In the figure 1 the colored zones show the parts of the process that can be done by independent parallel threads. In order to easily adapt the method for such parallel calculations with maximum performance and flexibility, the function provided in the package is a mere optimization function (like Optimize operation in Igor rather than the CurveFit operation) so that the user has to provide the code for χ^2 calculation. But I think that, since using the package requires some coding anyway, it shouldn't be a big deal to add a few extra lines of code to calculate the fitness function (see demo experiments). On the other hand this also makes the program more general (it can easily be adapted for, e.g. simultaneous fitting of several data sets, a different form of χ^2, adding penalties to implement constraints etc.).

The idea is therefore to calculate the χ^2 of the whole population in a single call to a user-supplied Igor function, if you need speed this function can use multithreading or call a GPU code so that the calculations for different members of the population run in parallel.

As in the original code the function opens a window to display the evolution of the whole population in the form of a "color table": the coefficients are mapped to rainbow colors in function of their values with respect to the bounds (red for the lower bound).

Using OpenCL requires installing the IgorCL XOP. Then first run the function `IgorCLInfo` to check which platforms are available and how they are numbered (see XOP documentation). However you can also use the procedure without it, with all-Igor code (NB. conditional compilation directives are used to avoid errors when the XOP is not installed). In order to double check my calculations I usually code two equivalent versions of the χ^2 function: one in Igor code only and one calling an OpenCL kernel (see demo experiments). The Igor version is of course slower but much easier to debug.

3

OpenCL defines a language and a standard interface to compile and run code on GPU. It is similar to NVidia's CUDA however it is multi-platform (e.g. I could use it on Intel chipsets). The OpenCL language is basically C with some restrictions (e.g. memory layout and allocation) but also additional features (e.g. vector types and operations). The philosophy of the interface is sketched on Figure 2. The IgorCL XOP provides a simplified interface, even if it has some limitations compared to a complete interface it is very simple to use with Igor since it does all the memory allocation based on Igor waves provided to the XOP operation.
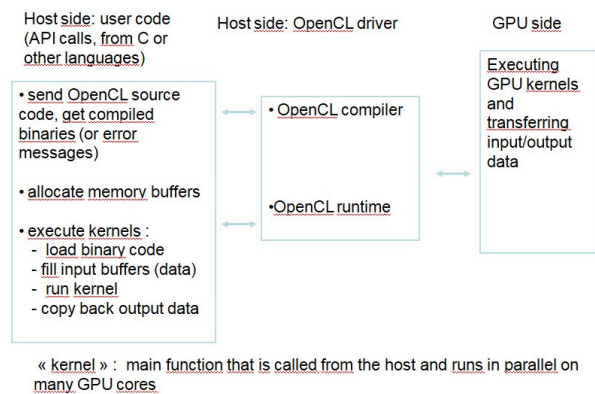

Figure 2: CPU/GPU code layout

2 Demo experiments

The demos, developed under Igor Pro 8, are rather simple examples, I tried to make the code well structured and clear so the demos can be used as an easy starting point for your own projects. All demos provide two equivalent versions of the fitting and χ^2 functions, one in Igor and one in OpenCL, so you can use them with or without the XOP (the mode is selected by two flags as explained below). Since I wanted the two versions to be similar I don't use very much the Igor wave scaling in the calculations. The demos are rather academic examples that I invented and tried to make simple so I won't discuss here the performance and effectiveness of the DE code vs., e.g. a simple FuncFit. And in general the performance greatly depends on the size of the population, initial guesses, the type of GPU etc. Actually I'm using this code to fit my NMR and X-ray data, often involving more than 10 fit parameters and requiring more complex code (convolution, matrix diagonalization etc.), so that GPU speedup is essential (can be a factor of hundreds compared to Igor code, on my gaming GPU).

The `fitdemo1a` and `fitdemo1b` experiments fit a curve to a sum of a gaussian and a lorentzian. It is a simple fit but I put the two curves quite close and overlapping so that the algorithm may miss the best fit if the population size is too small (or if the guesses are too far in case of FuncFit). Both versions deal with the same data but in `fitdemo1b` the fit function is written in the "all-at-once" form. The latter is useless here since we have a simple expression for the function but I wanted to show how to write an equivalent OpenCL code (compare the code of the fit function in the procedure

`proc_model_Igor` and the one in the notebook `nbcl_model`). Note that actually the OpenCL code of the second version will be slower : here the calculations are simple and fast so that allocating a local buffer necessary for this version can slow things a bit.

The `fitdemo2` example involves a Fourier transform and is actually inspired from one of my projects. The problem consists of evaluating an approximate form of an aperture function from the Fraunhoffer diffraction pattern. The diffraction image is given by the squared magnitude of the Fourier transform of the aperture function, but we obviously cannot use the inverse transform with no information on the phase, we have therefore to make some guesses about the general form of the aperture function and fit it. This demo shows an example of a "difficult" fit: a) χ^2 may exhibit an oscillatory behavior with many local minima and b) it is not easy to find the right guesses.

Here how the Igor code is structured in all demo experiments.

The function `fit` in the main procedure prepares the data (a simulated data set, using the wave "coeftrue"), then configures the parameters, the initial coefficients and calls the library fit function `GEN_optimise_CL`. The coefficient waves are also displayed in a table for easy editing. If you don't use OpenCL you should set the `pldev` variable to -1.

The procedure `Proc_fit` defines two functions that have to be supplied as parameters to the main routine of the library (`GEN_optimize_CL`, see reference in the last section) : one function to calculate χ^2 and another that will be called periodically during the fitting process and which receives the current best coefficient set (to update the status of the fit e.g. refresh a curve on a graph according to the current best solution). Both functions can either use an Igor code or call an equivalent OpenCL module, I define the constants `mode_chi2` and `mode_model` used to select which code is used for each task, in the procedure `fit` :

```
constant mode_model=1  //0 for igor code , 1 for cl
constant mode_chi2=1  //0 for igor code, 1 for cl
```

The Igor versions are defined in the procedure `proc_model_Igor` and the OpenCL ones in `proc_model_CL`. The demos will work too if the XOP is not installed, then you have to set these constants to zero.

All Igor functions interacting with the `IgorCL` XOP are in the procedure `proc_model_CL`. Using the OpenCL versions requires one extra step : before running the fit we need to send the source code to the compiler and store the obtained binaries. This is done via the operation `IgorCompileCL` of the XOP.

I'm using notebooks to edit the OpenCL code, this allows to keep everything in the same place and I found it very handy – thanks to Igor's powerful set of operations on notebooks I can turn Igor into an OpenCL IDE :). There is also a procedure file coming with the XOP that provides a code editor window but here I use my own (very simple) code because I prefer to have different parts of the OpenCL code edited in different notebooks (think of different .c and .h files making a C project). The function `GetCLcode` then combines them into a single string variable that will be passed to the compiler (it also pastes the string to another notebook so that the whole assembled code can be inspected). Edit this function following your needs. The function `CompileCL`

gets the code string and passes it to the compiler, if there are errors then another note-book is created to display the build log. Well, it's not VScode yet (and no debugger) but I like it.

For example, in the demo1 experiments the code is in two portions: `nbcl_model` (the fit function) and `nbcl_kernels` (main routines). For a simple curve fitting there is no need to change anything in the kernels when modifying the fitting function. For such simple fits you don't even need to know much about OpenCL, if you are fluent in C it will be easy to write your own fit functions. You can look at the quick reference guide: https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf to check for the data types and math functions.

When I tested the demo1 experiments on my two GPUs installed (Intel UHD Graphics 770 and NVidia GTX 3060) I found that they were was running faster on the Intel platform even though the other one is more powerful. The main difference here is the number of cores (you can check it using the utility "GPU Caps Viewer"), however with a relatively small population we might not use the full potential of the gaming board. On the other hand, the Intel integrated GPU shares the memory with CPU, so it seems that data transfer speed can be a dominant factor when calculations are simple.

3 Library reference

Here is the list of functions exported by the module `GeneticOptimisation_CL.ipf` :

`GEN_optimise_CL(...)` main function: runs the fit, see parameters below
`GEN_clear()` discards some of internal data (may be used before saving the pxp file to disk to save space)
`GEN_chi2best()` returns the value of the fitness function for the last best vector
`GEN_chi2avg()` returns the average of the fitness function for the last population
`GEN_chi2dev()` returns the std.dev. of the fitness function for the last population
`GEN_maxdist()` returns the maximum euclidean distance for the last population

The following functions do some statistics on the current population. In principle error estimation and covariance matrix calculation can (should?) be done using an ML step after the fit (as Andrew's XOP does, this can also be done calling FuncFit after the fit). However, due to the "self-calibrating" nature of the DE algorithm [2] the statistics done on the last population should give a relevant information about the uncertainty. It also allows error estimation in cases where derivative calculation suffers from numerical problems (e.g. the model need evaluating a histogram). Note that since these functions operate on the last population matrix they can't be used (and will return NaN) after `GEN_Clear` was executed (population matrix discarded) :

`GEN_coefdev(i)` returns the std.dev. of the coefficient number i for the last population (error estimation)
`GEN_coefcov(i,j)` returns the covariance matrix element i,j calculated on the last population
`GEN_coefcorr(i,j)` returns the correlation coefficient (cov(i,j)/(sdev(i)*sdev(j)))

exported function templates:

```
GEN_func_chi2array, GEN_func_updmodel
```

Description of the main function :

```
Function GEN_optimise_CL(f_chi2array, f_updmodel, coefs, limits,
[holdwave, refwave, popmul, k_m,recomb,bfrac, iters, updrate_ct,
updrate_m ,q, chi2tol,xtol, pldev,maxCLthreads])
```

This main routine requires the following mandatory parameters:

`f_chi2array, f_updmodel` : external functions to be supplied, the syntax has to follow the following templates:

```
Function GEN_func_chi2array(popmatrix, chi2array, [refwave])
wave popmatrix //(input) matrix of coefficient vectors :
               //   popmatrix[parameter number][vector number]
wave chi2array // output : 1D wave containing results
wave/wave refwave  // (input) : references of other waves
End
```

The optional parameter `refwave` is passed if it was specified in the call of `GEN_optimise_CL`, it allows to pass any other waves that may be needed without need for runtime lookup. Typically, it will be the waves representing the experimental data (e.g. x and y coordinates of data points) and the wave to be filled with results.

```
Function GEN_func_updmodel(coefs, [refwave])
Wave coefs   //(input) : current best vector
   wave/wave refwave  // (input) : references of other waves
End
```

This function will be called periodically : every `updrate_m` iterations (unless the best vector didn't change since last call), and at the end. `refwave` (same as for the previus function) will typically also contain the reference of a wave to fill with the modeled data (fit output).

For example, in the demo1 I use a `refwave` specified as:
`Wave/WAVE wr = ListToWaveRefWave("xdata;ydata;fitgen_ydata")`

So the `GEN_func_chi2array` function uses the first two and `GEN_func_updmodel` uses `xdata` and `fitgen_ydata`.

`coefs` : (input/output) this parameter is a 1D wave containing the initial coefficients (these will be used as a member of the initial population, the remaining members are generated randomly within bounds), and will receive results after the fit.

`limits` (input): two column matrix defining the bounds for coefficients (we always need this for global optimization since we cannot scan the whole space): `limits[i][0]` and `limits[i][1]` are the lower and upper bounds.

Optional parameters:

holdwave (input): 1D wave of same length as coefs, specifies which coefficients should be held constant during the fit (set those positions to 1, otherwise 0).

refwave (input): as explained above, if not provided the user functions will be called without the refwave parameter.

popmul (default=100) : population multiplier - defines the size of the population as popmul times the number of coefficients (size of coef wave). The population has of course to increase with the number of coefficients, in principle the volume of the configuration space increases rather exponentially than linearly but this is how it is done in other implementations so I didn't change it.

k_m (default=0.7) : mutation constant (or "differential weight", "F" in [4])

recomb (default=0.5) : recombination constant (or "crossover probability", "CR" in [4])

bfrac (default=1): fraction (between 0 and 1) of best vectors from which the "base vector" ([4]) will be chosen e.g. bfrac=1 means from all vectors (strategy named "DE/rand/1" in [2]) and bfrac=0 means it is always the best vector (strategy named "DE/best/1" in [2]), it gives faster convergence at the expense of genetic diversity (more chances to miss the best solution). bfrac<1 needs sorting of all vectors at each step and thus can be a bit slower.

iters (default=500) : first stopping criterion: maximum number of iterations

chi2tol (default=0.005) : second stopping criterion : stop if the ratio stddev/avg of χ^2 distribution of the population falls below. Set to zero to disable this criterion.

xtol (default=0.02) : third stopping criterion : stop if maximum euclidean distance in population falls below xtol. Set to zero to disable. The distance is based on normalised coordinates (0-1 between lower and upper bounds for each coefficient).

updrate_ct, updrate_m (default=1) : intervals (in number of generations) for updating the color table and the calls of the user update function. If calculations are fast you can gain some speed setting higher values.

quiet (default=0) : set to 1 for quiet mode (don't print results in history)

pldev (default=-1) : if negative the generation of trial vectors will use Igor code; if non-negative it will run on GPU, then pldev defines the platform (bits2,3) and device (bits 0,1), i.e. pldev=4*platform nnumber + device number (e.g. set pldev=4 for device 0 on platform 1). The numbering is according to the values obtained via IgorCLInfo of the XOP. If you set a non-negative pldev and the XOP is not detected then the routine will display a message and continue falling back to the Igor version.

maxCLthreads (default=3000) : if pldev>=0, will define the max. number of threads to use (set approx. to the number of cores or its multiple for optimum performance).