

# MULTI-DIRECTIONAL OPTIMISATION ON THE GPU

NIELS STCHEDROFF

ABSTRACT. The multi-directional (MD) technique is a general purpose tool for optimisation, that is, finding the global maxima or minima of some objective function in a given domain. Any function that produces a relatively continuous surface may therefore be suitable.

Using a graphics processing unit (GPU) for MD optimisation demonstrates an increase in speed of up to 400-fold compared to using a central processing unit (CPU). More than a 100-fold speed up was seen across a range of problems. This was achieved despite non-trivial amounts of branching in the algorithm.

The main problem investigated was a particular form of portfolio optimisation. As a further test, the same algorithm was used to find the minima of the Schwefel function, which has several local minima and is a particularly difficult candidate for numerical optimisation. The great flexibility of the MD technique and the performance obtained strongly suggests that the GPU implementation has great potential in finance.

## 1. INTRODUCTION

Direct search (the general class of optimisation methods of which MD is an example) has undergone a renaissance in the last two decades [1]. This has been due to a better understanding of the mathematics underpinning the Nelder-Mead [2] and the Multi-Directional methods [3]. They are often effective when the problem is sufficiently complex or computationally difficult so that an exact answer cannot be obtained [1].

Direct search methods do not rely on the objective function's derivatives or its numerical approximations. The function to be optimised can be regarded as a black box, with only the input parameters and the resulting value being exposed to the optimisation method [4].

The effectiveness of quasi-Newton methods and the availability of software tools which ease their use has caused direct search techniques to be somewhat overshadowed [5, 1]. However, Direct Search techniques are invaluable [1] for problems that meet the following conditions:

- Calculation of the  $f(x)$  is very expensive or time-consuming
- Exact first partial derivatives of  $f(x)$  cannot be computed.
- Numerical approximation of the gradient of  $f(x)$  is impractical due to time/resources.
- The values of  $f(x)$  are noisy, due to inaccurate source data or the nature of the algorithm.

We use the multi-directional search (MDS) method of Torczon [6, 7, 4], rather than the more common direct search technique, the Nelder-Mead method [5, 4]. Both MDS and Nelder-Mead are relatively simple to code and analyse. However, it has been found that, unlike MDS, the Nelder-Mead algorithm can converge to non-minimizers when the dimension of the problem is large enough, for a wide range of test problems [7, 5].

## 2. MULTI DIRECTIONAL SEARCH

This method involves generating an  $n$  by  $n + 1$  simplex of values, where  $n$  is the number of variables. This represents a start point. Each row in the simplex represents the input values for a test of the function in question.

The best result (i.e. the one evaluating to the smallest function value) in the simplex is located and then used to generate possible new values for each variable.

$$(2.1) \quad S = \{v_0 \dots v_n\},$$

where  $S$  is the simplex,  $v_0, \dots, v_n$  are the vertices and  $v_0$  is the best vertex found so far:

$$(2.2) \quad r_i = v_0 - (v_i - v_0),$$

$$(2.3) \quad e_i = (1 - \mu) * v_0 + r_i,$$

where  $\mu$  is the *expansion* coefficient. A common default value is 2.0 [3].

$$(2.4) \quad c_i = (1 + \theta) * v_0 - \theta * r_i,$$

where  $\theta$  is the *contraction* coefficient. A common default value is 0.5 [3].

If  $f(r_i) < f(v_0)$  and  $f(e_i) \geq f(v_0)$  then  $S_{new} = \{v_0, e_1, \dots, e_n\}$

If  $f(r_i) < f(v_0)$  and  $f(e_i) < f(v_0)$  then  $S_{new} = \{v_0, r_1, \dots, r_n\}$ .

Otherwise,  $S_{new} = \{c_0, \dots, c_n\}$

This process continues until either a maximum number of evaluations is exceeded, or the computed values of the rows of the simplex match specified criteria – often a predetermined maximum range between best and worst.

From this, we can see that parallelisation can take place at several levels [3]. The simplest is to run the optimisation for different starting simplices concurrently. Another is to compute and evaluate the three tests in parallel within each iteration of the optimisation function. Finally, we can evaluate the transformations of the simplex in parallel as matrix operations. The implementation discussed in this paper looks at the first and last options.

An implementation of Multi-Directional in MATLAB is available from [8].

## 3. WHY GPU?

As we have seen, the MD algorithm is inherently parallelisable at different levels. GPU devices are well suited to matrix operations - transforming the simplex is a good fit. But, the algorithm requires branching, which is generally difficult for GPUs, given the use of a 32-lane SIMD architecture that implements, but penalizes, divergence within groups of 32 threads (warps) [9].

The aim of this work was to investigate whether it was possible to overcome the branching issue – at least to the point where there were significant performance gains from using GPU.

4. GPU ARCHITECTURE

CPUs are considerably faster on a thread by thread basis. But the number of threads that a GPU can run in a truly parallel fashion gives it a considerable advantage, for a suitable problem.

The NVIDIA GPU architecture (used in this study) comprises at least one multiprocessor. Each contains:

8 cores (SIMD type) 16384 registers (8192 on older hardware) 16KB of shared memory 8KB cache for constants 8KB cache for textures

The nature of the hardware means that it is ideal for performing the same (or similar) operations on a large number of threads - 1024 per multi-processor (maximum) on the latest hardware [10].

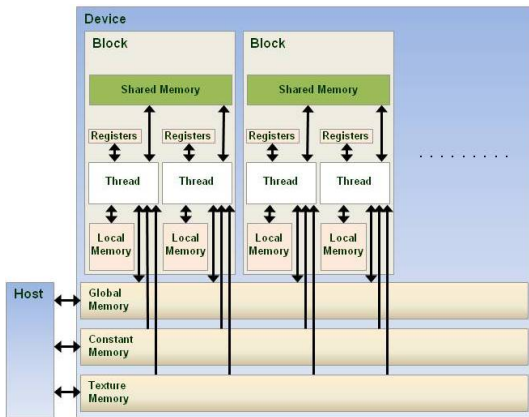


Figure 1: NVIDIA GPU Architecture (simplified)

Space is at a premium - for example, 2KB of shared per block when running 8 blocks. It is possible to store and retrieve data from the main memory on the host machine but this can be very slow and is generally avoided.

5. THE PROBLEM MODELLED

The test problem examined was a simplified portfolio optimisation. We consider a portfolio consisting of  $n$  risky stocks, no risk-free asset, and (positive definite) VCV matrix  $\sigma$ . Our objective is to find the (column) vector of portfolio weights  $w$  which minimises the total portfolio variance. We therefore need to find the global minimum of the variance function

$$V(w, \sigma) = w' \sigma w,$$

where the prime denotes transpose. The only constraint we impose is that the portfolio weights must sum to one, i.e.  $\mathbf{1}'w = 1$ , where  $\mathbf{1}$  denotes a column vector whose elements are all equal to 1. As can easily be verified, the solution is given by

$$w(\sigma) = \frac{\sigma^{-1}\mathbf{1}}{\mathbf{1}'\sigma^{-1}\mathbf{1}}.$$

In particular, in the case where  $\sigma_{ii} = d > 0$  and  $\sigma_{ij} = c > 0$ , for  $i, j = 1 \dots, n$ ,  $i \neq j$  and  $d > c$ , the solution is simply  $w_k = 1/n$ , for  $k = 1, \dots, n$ . This is the case we consider, with  $n = 15$ ,  $d = 10$ ,  $c = 7.5$ .

This problem is trivial, but it has a sufficiently complex numerical calculation to make testing illustrative. Furthermore, due to the particularly simple choice of VCV matrix, e.g. for  $n = 3$  (100,101,100) will have the same overall "cost" as (101,100,100). This makes the function a less than ideal candidate for Direct Search optimisation (an ideal function would produce unique values for all combinations of inputs).

```

float optimise(float * weights, int size)
{
    float variance 0.0f;
    float temp 0.0f;

    // The Lagrangian calculation
    for( int iLoopOuter(0); iLoopOuter <
        size; iLoopOuter++)
    {
        for( int iLoopInner(0); iLoopInner <
            size; iLoopInner++)
        {
            temp = weights[iLoopOuter];

            if( iLoopOuter == iLoopInner )
            {
                variance += temp * temp * float
                    (10.0);
            }
            else
            {
                variance += temp * temp * float
                    (7.5);
            }
        }
    }

    return variance;
}

```

Figure 2: Source code for function to optimise

Figure 2 shows the function to be optimised expressed as code - it generates the VCV matrix on the demand.

## 6. DETAIL OF THE SOLUTION

The hardware used was an NVIDIA Tesla C1060 and the code was written in the CUDA-C programming environment.

As mentioned above, the parallelisation was in two parts: First, the system runs a single start in each block allowing a maximum of 240 simultaneous runs on the C1060. If more than 240 starts are required, each block runs multiple starts. Secondly the matrix operations within the

blocks are parallelised among the threads in the block.

A maximum of 128 threads per block is specified, for reason of efficiency and performance. When optimising for less than 12 variables, this means that the entire simplex can be evaluated at once. When more variables are required, the threads loop, computing sections of the simplex one after the other.

To reduce the amount of shared memory to a minimum, only two copies of the simplex are held – the original values (from the previous iteration) and a test version. The test version is used in turn to hold the reflection, expansion and contraction trials. The winner is evaluated, and used to modify the original simplex.

This does mean that 4 transformations are required rather than 3 (for each iteration), but it does cut down the shared memory required. For example, 2Kb is required for  $n = 15$ . In any case, in the traditional implementation, the winner has to be copied to the array holding the original.

$n = 15$  is the maximum possible before the requirements for shared memory reduce the number of blocks that can be run at any one time.

Considerable care was taken to reduce branching, in so far as was possible.

The simplex is not sorted at each transformation in order of value of the results. Instead, the best value was identified and moved to the head of the simplex. While a number of implementations of the Multi Directional algorithm sort the simplex, this is not required by the mathematics

of the process and makes the implementation slower.

At the end of each iteration a test was included to prevent an infinite loop when all the values in the simplex were too similar [11].

The results are written to global memory at the end of the run for a given "start". A different location is used for each result. Writing data to main memory so frequently may seem strange in the light of the known poor performance of access 400-600 clock cycles is an often quoted number. In fact this is the time that is taken until the value written would be accessible. From the point of view of the GPU thread, the write would take 4 cycles and then continue with the rest of the thread [3].

To validate this structure, tests were run removing the writes to global memory entirely. Only a minor effect on performance was encountered, matching the predictions of the costs of the writes, indicating that this concept is valid.

The fact that global memory can be used like this is perhaps not as well known as it should be – it is a valuable tool due to the resource constraints of on-chip memory.

## 7. RESULTS

For comparison purposes, two other problems were modeled as well. The results are given below.

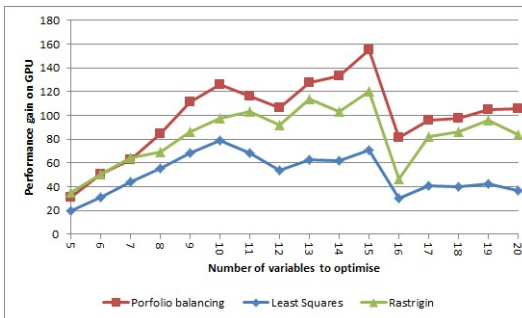


Figure 3: Performance for three problems

Figure 3 shows the results for the portfolio balancing problem over a range of values for the number of variables.

These problems were run with 20,000 starts and a maximum of 10,000 evaluations. The values for the reflection, expansion and contraction coefficients were 1.0, 2.0, and 0.5 respectively.

When run with  $n = 15$ , the GPU implementation running on a C1060 returned a solution in 1.65s. A reference CPU implementation took 257.95s (Intel i7). This represents speed increased by a factor of 155. The selected result had values within 0.15% of the optimum result.

A run for ten variables showed an increased speed factor of 110. A run for five variables gave a 30 times performance gain. This is due to the smaller numbers of variables, requiring a smaller number of threads per block, and thus leading to lower use of the GPU resources.

Beyond  $n = 15$ , performance drops, since fewer blocks can run at the same time on the GPU – but performance increase factors of 80-90 are still common.

The lower performance for Least Squares and the Rastrigin [12] function

seem to be related to their lower complexity, compared to the portfolio balancing problem. The shape of the plots is similar, validating reasons suggested above for the varying performance according to the number of variables.

As a further experiment, the Schwefel [13] function was investigated. This provides a complex surface with a number of minima.

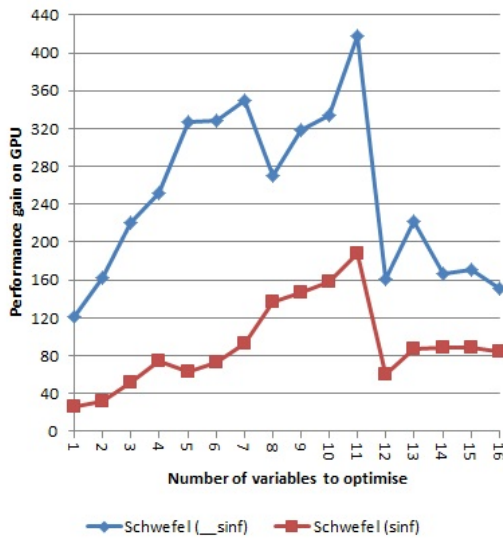


Figure 4: Schwefel Function for 2 variables

After initial experiments, the number of starts was increased to 100,000. Two versions of the problem were coded on the GPU. They differed in the use of the standard `sinf` and the GPU specific `_sinf` functions. `_sinf` is considerably faster (taking a single clock cycle [14]) but sacrifices accuracy. The results are impressive, with a peak of 417 times speedup achieved at  $n = 15$ . As expected the `sinf` version is much slower.

<sup>1</sup>The single precision version of the `sin` function.

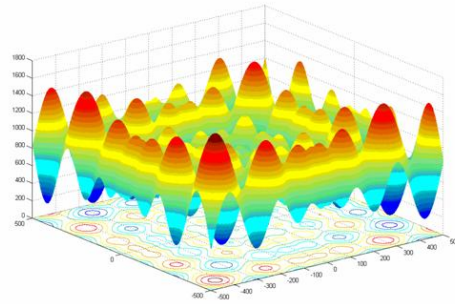


Figure 5: Schwefel function performance [15]

## 8. CONCLUSIONS

The implementation of the Multi-Directional search optimisation technique has been demonstrated for GPUs despite the using branching, which causes a performance penalty due to the SIMD architecture [9]. High levels of performance (at least 100 times faster than single thread on CPU) have been demonstrated for several problems.

The MD techniques flexibility and this level of performance strongly suggest that it will be of value in finance. Any suitable function that takes a number of values and returns a single overall cost can be optimised in this manner, with little or no changes required to the optimiser.

The limitation to this implementation is chiefly in the complexity and data requirements of the function – fitting it into the limited resources available on the GPU can be a challenge.

A possible enhancement would be a modification to improve performance for problems with less than 10 variables. Below this level, GPU use falls away, since

the number of threads required to compute the simplex falls below the number available for use. This could be dealt with by modifying the structure to make use of more threads for level of operation.

## REFERENCES

- [1] M.H. Wright. Direct search methods: once scorned, now respectable. In *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis)*, pages 191–208. Addison Wesley Longman, Harlow, United Kingdom, 1996.
- [2] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
- [3] V. Torczon. On the convergence of the multi-directional search algorithm. *SIAM J. Optim.*, 1:123–145, January 1991.
- [4] C.T. Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, 1999.
- [5] Robert M. Lewis, Virginia Torczon, and Michael W. Trosset. Direct search methods: then and now. *Journal of Computational and Applied Mathematics*, 124:191–207, 2000.
- [6] E. Boyd, Kenneth W. Kennedy, Richard A. Tapia, and Virginia J. Torczon. Multi-directional search: A direct search algorithm for parallel machines. Technical report, Rice University, 1989.
- [7] V. Torczon. *A Direct Search Algorithm for Parallel Machines*. PhD thesis, Rice University, 1989.
- [8] C.T. Kelley. Multidirectional search source code in matlab. Available at [http://www.siam.org/books/kelley/fr18/OPT\\_CODE/mds.m](http://www.siam.org/books/kelley/fr18/OPT_CODE/mds.m).
- [9] R. Fernando M. Pharr. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.
- [10] NVIDIA. Cuda occupancy calculator. Available at [http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls).
- [11] K.I.M. McKinnon. Convergence of the nelder-mead simplex method to a non-stationary point. *SIAM J Optimization*, 9:148–158, 1999.
- [12] H. Muhlenbein, D. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.
- [13] H. P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, 1981.
- [14] NVIDIA. Cuda programming guide. Available at [http://developer.download.nvidia.com/compute/cuda/2\\_3/toolkit/docs/NVIDIA\\_CUDA\\_Programming\\_Guide\\_2.3.pdf](http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf).
- [15] A. Hedar. Schwefel function. Available at [http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar\\_files/TestGO\\_files/Page2530.htm](http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page2530.htm).