PORTFOLIO OPTIMISATION

N. STCHEDROFF

Abstract. Portfolio optimisation is computationally intensive and has potential for performance improvement. This paper examines the effects of evaluating large numbers of proposed solutions in parallel for use with Direct Search [1] optimisation. This leads to a method that has a considerable performance increase. A GPU implementation demonstrates an order of magnitude performance improvement over the best available multi-threaded CPU. The new algorithm and GPU combined offer a performance increase of more than 60x compared to CPU.

1. Introduction

Assuming we have: a portfolio consisting of n risky stocks, no risk-free asset, and (positive definite) VCV matrix $\boldsymbol{\sigma}$. Our objective is to find the vector of portfolio weights $\boldsymbol{w}$ that minimises the total portfolio variance. We therefore need to find the global minimum of the objective function:

$$f(w) = V(w, \sigma) = w'\sigma w \tag{1.1}$$

where the prime denotes a transpose. The constraint we impose is that the portfolio weights must sum to one, i.e. $\mathbf{1}'\boldsymbol{w} = 1$, where $\mathbf{1}$ denotes a vector whose elements are all equal to 1.

This paper examines the possibilities for increasing computational efficiency by evaluating multiple values for $\boldsymbol{w}$ in parallel. This is achieved by applying Direct Search techniques to the optimisation.

2. Direct Search methods

Direct Search (the general class of optimisation methods of which Multi Directional [2] optimisation is an example) has undergone a renaissance in the last two decades [1]. This is down to a better understanding of the mathematics underpinning the Nelder-Mead [3] and the Multi-Directional methods. They become effective when the problem is sufficiently complex or computationally difficult such that a perfect answer is unachievable [1].

Direct Search methods do not rely on the objective function's derivatives or numerical approximation. The function to be optimised may be a "black box", with only the input parameters and the resultant value exposed to the optimisation method [4].

The effectiveness of quasi-Newton methods and availability of software tools that ease their use has caused direct search techniques to be overshadowed [5, 1]. However, Direct Search techniques are

invaluable [1] for problems that meet the following conditions:

- Calculation of $f(x)$ is very expensive or time-consuming
- Exact first partial derivatives of $f(x)$ cannot be determined
- Numerical approximation of the gradient of $f(x)$ is expensive in terms of time/resources
- $f(x)$ is noisy, due to source data or the nature of the algorithm.

Two Direct Search methods of interest are the multi-directional search (MDS) method of Torczon [6, 7, 4] and the Nelder-Mead method [5, 4]. Both MDS and Nelder-Mead are simple to code and analyse. It has been found that the Nelder-Mead algorithm (unlike MDS) can converge to non-minimisers when the dimensions of the problem are large enough [7, 5].

For this reason MDS was selected as the algorithm of choice for this work.

Both methods use a n by n matrix (simplex). This represents a start point. Each row in the simplex represents the n input values for the function in question. The start point is randomly generated, within the required constraints for the input variables.

The best set of input values (i.e. the ones giving the smallest result) in the simplex is located and used to generate possible new values for each variable. Several different simplicies are generated and evaluated against each other, as a simple step in the algorithm. The best is taken as the input for the next iteration.

This process of finding the best and generating a new simplex from it is continued until it is judged that an optimum value has been found. This means that for each iteration, n sets of test values must be evaluated a number of times times.

3. MDS in detail

$$S = \begin{pmatrix} v_1 & .. & v_n \end{pmatrix} \tag{3.1}$$

where S is the simplex. $v_0, \ldots, v_n$ are the vertices where a vertex is defined as a vector of length n representing one possible set of portfolio weights, where v_0 is the best vertex found so far:

Three trials are made – reflection (R), expansion (E) and contraction (C).

$$R = \begin{pmatrix} r_1 & .. & r_n \end{pmatrix} \tag{3.2}$$

where:

$$r_i = v_0 - (v_i - v_0), \tag{3.3}$$

$$E = \begin{pmatrix} e_1 & .. & e_n \end{pmatrix} \tag{3.4}$$

where:

$$e_i = (1 - \mu) * v_0 + r_i, \tag{3.5}$$

and μ is the *expansion* coefficient. A common default value is 2.0 [2].

$$C = \begin{pmatrix} c_1 & .. & c_n \end{pmatrix} \tag{3.6}$$

where:

$$c_i = (1 + \theta) * v_0 - \theta * r_i, \tag{3.7}$$

and θ is the *contraction* coefficient. A common default value is 0.5 [2]. Then the updated simplex is computed as S_{next}.

If $R_{best} < f(v_0)$ and $E_{best} >= f(v_0)$

then $S_{next} = R$

If $R_{best} < f(v_0)$ and $E_{best} < f(v_0)$

then $S_{next} = E$.

Otherwise, $S_{next} = C$

3.1. **Evaluation.** We have seen in the previous section that we need to evaluate the objective function at n vecticies for each evaluation of the simplex. This naturally leads to the suggestion that we consider a modified version where $\boldsymbol{w}$ is an nxn matrix itself. This can be represented in matrix format as below:

$$\begin{pmatrix} f_1 & .. & .. \\ .. & f_{..} & .. \\ .. & .. & f_n \end{pmatrix} = S'\sigma S \tag{3.8}$$

where

$$f_1....f_n$$

is the set of objective function values for the trial weights (verticies) :

$$v_1....v_n$$

At first sight this might seem an inefficient method of calculation – we are throwing away :

$$(n * n) - n$$

values in the final result. However, as we shall see, the performance issues are not obvious.

3.2. **Weights.** In the simplest formulation, the only constraint is that the weights sum to 1 :

$$\sum_{k=1}^{n} w_k = 1.0 \tag{3.9}$$

The approach taken is to sum the actual weights, and use that value to scale the weights to 1.0. We can calculate the scale on the matrix of weights by multiplying by a vector of 1s:

$$\begin{pmatrix} v_1^T \\ \\ v_{..}^T \\ \\ v_n^T \end{pmatrix} * \begin{pmatrix} 1 \\ \\ .. \\ \\ 1 \end{pmatrix} \tag{3.10}$$

We can carry out the scaling by calculating the scaling factors with 3.10, after each stage in the Multi Directional algorithm.

4. Implementation

The problem has been reduced to 2 matrix multiplications, a matrix transposition, vector-matrix multiplication and a vector-vector dot product multiplication. In the case of coding for a BLAS library, the transposition is handled by the settings for the DGEMM (or SGEMM) function. This means that for a BLAS, the core of the problem can be coded with two calls. The scaling operations are trivial in performance terms.

Various trials were run in developing this method. The discovery that the second matrix multiplication was more efficient in terms of time taken was a surprise - the original intention was to simplify implementation. The calculation times were tested using the Intel MKL BLAS library running on an i7 920 and on an NVIDIA C1060 GPU running the CUBLAS BLAS.

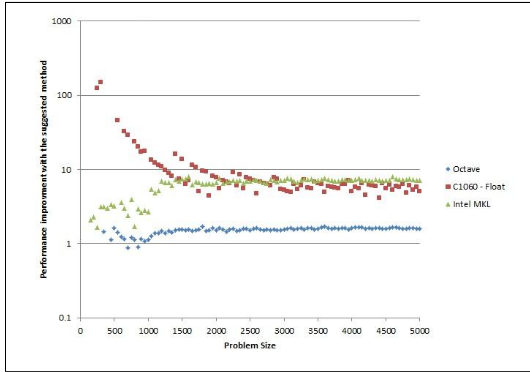

Figure 1: Performance improvement of the suggested method compared to using vector multiplications

Of interest is the fact that little or no improvement is seen when running this calcuation in Octave – which uses an unoptimised single threaded version of the ATLAS BLAS for matrix operations. This strongly suggested that the performance gains are from the levels of optimisation possible between matrix*matrix and matrix*vector calculations in the high performance libraries (CUBLAS and Intel MKL). The initial very high gains for GPU are caused by the massive reduction in the number of kernels to be launched to complete the calculation.

The reason for the gain is that for:

$$T_s = 2 * T_{mm} \tag{4.1}$$

$$T_o = n * (T_{vv} + T_{mv}) \tag{4.2}$$

$$T_s < T_o \tag{4.3}$$

for the high performance BLAS options, where T_s is the time taken for the suggested method, T_o is the time taken for the original method, T_{mm} is the time taken for a a matrix*matrix operation, T_{vv} is the time taken for a vector*vector operation and T_{mv} is the time taken for a matrix*vector operation.

In terms of raw speed up of GPU vs CPU, the GPU is around 9 times faster for computing matrix multiplications. This is a severe test; the CPU is a high end device, and the GPU is not the latest hardware. Many comparisons of GPU vs CPU hardware use single thread CPU code, whereas this case is a truly competitive comparison.

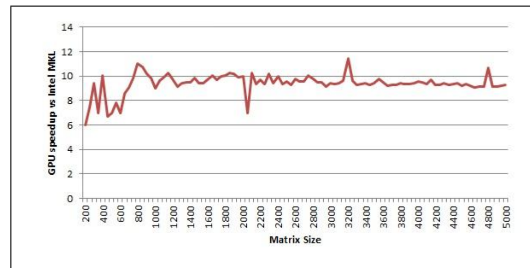

Figure 2: Performance of GPU vs CPU for matrix*matrix BLAS operations

5. Applying Multi Directional Search on the GPU

With the potential performance outlined above, it seemed likely that the results of applying this evaluation method would be interesting. The MDS transformations were implemented on GPU, as well as the evaluation method outline above. The selection of the new best simplex was done on the CPU – a naturally single thread function. All that the CPU is required to do is decide which path to follow – a comparison of 2 numbers.

5.1. GPU performance analysis. The scaling and MDS transformations were extremely efficient – they take on average less than 10% of the execution time for each iteration of the MDS algorithm.

The other 90% is consumed by the evaluation. For example, for a 583x583 covariance matrix, the MD transformation took 0.135697 ms, the scaling 0.196647 ms, while the evaluation took 3.718785 ms.

The efficiency of the MD transformations on GPU boosts the overall performance vs CPU to 10x. Again, it should be noted that this is using a high performance CPU with a top notch BlAS library.

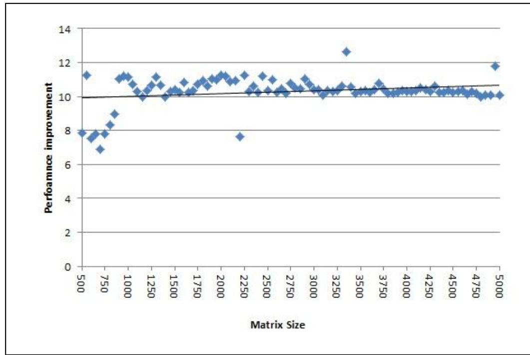

Figure 3: Performance of GPU vs CPU for MDS

The lower relative performance for GPU for portfolios smaller than 500 is due to the GPU not being fully loaded. If portfolios of this size are of particular interest, then an enhancement would be to compute the reflection, expansion and contraction steps in parallel on the GPU. This would increase utilisation on the GPU, bringing performance back to the 10x level.

6. Application in action

A variety of size of portfolio were run. For a 583 item portfolio (created from UK FTSE data), the minima was reached in under 60 seconds, and the minimum variance for the weights selected was an order of magnitude smaller than that produced by a genetic algorithm optimiser. The MDS optimiser produced consistent results when started from different initial points; showing a strong and rapid optimisation. It was noticeable the the first iteration returned a result only 4 times greater than the final result in less than 1 second.

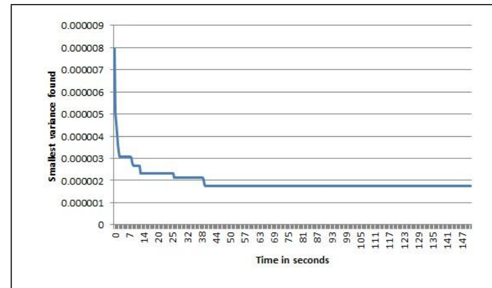

Figure 4: MDS performance on GPU for a 583 item portfolio

7. Returns vs Variance

Following the success of the methods outlined, when used for the simple portfolio optimisation problem (1.1), the work was extended to include returns.

We define the vector of returns as such that:

$$\mu = (\mu_1, \mu_2 \mu_n)^T \tag{7.1}$$

The mean return μ_ρ is given by:

$$\mu_\rho = \mu^T w \tag{7.2}$$

when w is the vector of weights for the assets that we are evaluating. If the risk free rate of return is r, then the Sharpe ratio is defined as:

$$S = (\mu_\rho - r)/\sigma_\rho \tag{7.3}$$

where σ_ρ is the variance of return of the portfolio as calculated by (1.1). Then

the portfolio w^* with the optimal risk/return trade-off is the one with the highest Sharpe ratio, and is given by the solution to the following optimisation problem:

$$\max S \tag{7.4}$$

$$w^T\mathbf{1} = 1 \tag{7.5}$$

$$w_i >= lb_i \tag{7.6}$$

$$w_i <= ub_i \tag{7.7}$$

Where $\mathbf{1}$ is a column vector of 1s, lb_i and ub_i denote the individual lower bound and upper bound respectively. The bounds were implemented by giving a large penalty to sets of weights that exceeded them. The penalty is proportional to the amount that the given bounds are exceeded by. This establishes a trust region [8], which is highly effective in ensuring that the optimisation algorithm tends towards solutions within the specified region.

7.1. **Validation.** When run for a 49 asset portfolio (picked from FTSE stocks randomly), the Sharpe ratio optimisation reached a stable value in less than 3 seconds. The risk-free interest rate was taken as 0.5%

	GPU	MATLAB
Mean return (ρ)	1.068057	1.065769
σ	0.006045	0.006038
Sharpe	175.844	175.670008

Figure 5: Sharpe ratio optimisation results

The results were compared witha reference MATLAB implementation, using the *fmincon* optimisation function (above).

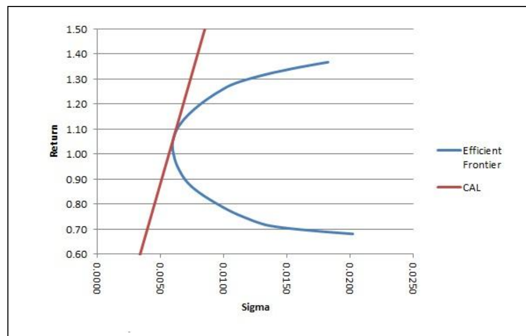

Figure 6: Efficient frontier for the 49 asset portfolio

The efficient frontier was computed, and the CAL (Captial Allocation Line) was plotted (Figure 6). As expected the results of the optimisation matched the tangency portfolio values.

To evaluate performance for a large portfolio, an optimisation was run on a 610 asset portfolio, selected from FTSE stocks. This achieved a stable Sharpe value in approximately 120 seconds. Each iteration took aproximately 15 milliseconds.

8. Conclusions

A very noticeable speed up has been obtained when evaluating trial values for the weights in blocks, for high quality BLAS solutions. In addition, this method drastically simplifies the coding problem to a few standard calls to a BLAS library. This performance gain can be exploited by Direct Search optimisation techniques.

Multi Directional Search (MDS) was implemented and demonstrated it's suitability for the task, with stable performance and high speed. When implemented on GPU, a 10x speed up is seen,

compared to a multi-threaded implementation of the same algorithm on a high end CPU.

Combining GPU and the new algorithm gives a speed up of over 60x over a CPU running the simple line by line evaluation of the simplex – even with optimal multi-threaded code on the CPU.

ACKNOWLEDGEMENTS

Special thanks are due to my colleague Jimmy Law at Riskcare, who wrote the MATLAB reference implementation, and provided help with the theoretical side of the work.

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] V. Torczon. On the convergence of the multidirectional search algorithm. *SIAM J. Optim.*, 1:123–145, January 1991.

[3] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[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. Multidirectional 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] Y. Yuan. A review of trust region algorithms for optimization. In *ICIAM 99: Proceedings of the Fourth International Congress on Industrial And Applied Mathematics*. Oxford University Press, USA, 1999.