Accelerating the Critical Line Algorithm for Portfolio Optimization Using GPUs

Raja H. Singh, Frederick C. Harris Jr., Lee Barford
Department of Computer Science and Engineering
University of Nevada/0171
Reno, NV 89577-0171 USA

***Abstract*—Since the introduction of the Modern Portfolio Theory by Markowitz in the Journal of Finance in 1952, it has been the underlying theory in several portfolio optimization techniques. With the advancement of computers, most portfolio optimization are done by CPUs. Over the years, there have been papers that introduce various optimization methods including those introduced by Markowitz, implemented on the CPUs. In the recent years, GPUs have taken the front seat as a technology to take computational speeds to new levels. However, very few papers published about portfolio optimization utilize GPUs. Our paper is about accelerating the open source Critical Line Algorithm for portfolio optimization by using GPU's, more precicely using NVIDIA'S GPUS and CUDA API, for time consuming parts of the algorithm.**

I. Introduction

Since its introduction in 1952 by Harry Markowitz, the Modern Portfolio Theory has become a solid foundation for the creation of various methods of portfolio optimization. The paper he wrote [1] provided mathematical model which took into account expected return and risk when constructing an optimal portfolio [2]. Other methods heavily used since the departure from Markowitz model such as "market-cap weighting" and "fundamental weighting" don't explicitly make assumptions about the risk and return parameters like the one used in Markowitz's theory. However, there has been a shift from market-cap weighted indexing to other schemes that take market anomalies into consideration rather than the "standard investment theory" that are very similar to the one Markowitz proposed; These modern methods tend to focus on minimizing the variance of the portfolio and focusing on portfolio diversification measures [2].

Since the model's introduction in 1952, the equity indexes have come full circle according to Kaplan [2]. This shift might have to do with the market slumps in the late 2000's or just be the cyclical nature of the market. Large hedge funds utilize various methods for portfolio optimization and prior to their poor performance in 2008, commanded large amounts of money to manage various funds (both private and public) on the notion of minimal exposure to risk [3].

With the advancement in computing capabilities, there has been an increase in utilizing computers for optimization algorithms. However, though there is an abundance of literature highlighting various methods of optimization, very few if any provide an open source implementation of such algorithms. Most methods used by non-professionals are either too slow or rely too much on VBA/Excel for portfolio optimization. Though there are some open source implementations and guides on using VBA such as [4], most examples utilize a small set of assets and as the size of the co-variance matrix starts increasing dramatically, excel starts having issues with the size of the data.

For this paper we utilized the open source (critical line) algorithm presented by Bailey and Prado in their paper [5]. The main reason for utilizing this algorithm besides the fact that it is open source was that it computed the same optimized portfolio as a VBA algorithm with a faster computation time [5] and it can handle any number of assets restricted only by the hardware limitations. The focus of this paper is to parallelize the bottle neck of the sequential algorithm to reduce the computation time for deriving an optimized portfolio. So far no one has implemented a GPU optimized Critical Line Algorithm and published the results.

II. Related Works

We found two papers that address portfolio optimization on the GPUs [6] and [7]; however neither of the papers provide any source code. The primary reason we selected to parallelize parts of the implementation provided by Bailey and Prado in their paper [5] for portfolio optimization was that they provided the source code in their work. The open source code allowed us to compare the performance of the sequential code and the one we optimized by parallelizing the hot spots side by side without worrying about any errors being produced by us in re-creating the sequential code.

III. Portfolio Optimization

A. Concept

Markowitz first published his paper [1] in the Journal of Finance in 1952 providing a mathematical model behind optimizing portfolios. In the paper Markowitz assumed that "the investor does (or should) consider expected return a desirable thing and variance of return an undesirable thing." This concept is referred to the "expected returns-variance of returns rule."

The idea is that for a given risk there should not be another portfolio that yields a greater or equal return with a risk that is less than or equal to the optimal portfolio. Another way Markowitz stated this is that for a given return there should not be a risk that is less than the risk of the optimal portfolio.

B. Model

Markowitz's model is based on portfolio returns, portfolio variances and the weights assigned to the assets included in the portfolio. We therefore summarize the ideas behind these components below.

The sum of the weights assigned to each asset have to sum up to one:

$$\sum_{i=1}^{N} X_i = 1 \textbf{ (1)}$$

The Expected Return of the whole portfolio is expressed as:

$$E = \sum_{i=1}^{N} X_i \mu_i \textbf{ (2)}$$

Where X_i is treated as a random variable representing the weight assigned to an asset and μ_i represents the expected return of an asset.

The risk in the model is expressed as the variance of the portfolio. However to understand the variance we first need to understand the co-variance between two assets. The co-variance between two assets is the expected value of the product of the deviations of the two assets from their mean. This equation is shown in equation 3, where $\sigma sigma_{ij}$ is the co-variance between asset i and j.

$$\sigma_{ij} = E[R_i - E(R_i)][R_j - E(R_j)] \textbf{ (3)}$$

We can use the fact that the variance of asset i is the co-variance of σ_{ii}. We can express the variance of the whole portfolio as:

$$V = \sum_{i=1}^{N} \sum_{j=1}^{N} \sigma_{ij} X_i X_j \textbf{ (4)}$$

From the co-variance matrix we are able to compute the correlation between the assets. An ideal portfolio usually contains assets with minimal correlation between each other [9] and where the variance of the assets with respect to other assets in the portfolio is more important than a particular asset's own variance alone [8].

IV. Critical Line Algorithm

Markowitz's theory focuses on deriving an efficient portfolio that yields the maximum return for a minimum risk(or volatility). When Several efficient portfolios are computed, they make up the Efficient Frontier. There can be no portfolios that exceed the frontier, the optimal portfolio(max return or min variance) will be on the frontier and all other portfolios will be inside the boundary of the curve. The portfolios on the Efficient Frontier will dominate the rest of the portfolios [9]. Harry Markowitz referred to the the method of deriving the entire efficient frontier as the "Critical Line" Method. It is also known as Markowitz's Efficient Frontier. However we will call it Critical Line because it is referred to by that name in [1] and [5].

This method has changed over time from minor to major modifications such as including negative weight for short sales resulting in the problem taking on the form of an unconstrained problem. However, the algorithm provided by Bailey and Prada doesn't allow negative weights(associated with short sales), keeping the method as a constrained problem with inequality conditions and equality conditions as specified in their paper [5]. According to Bailey and Prada, there isn't an analytic solution to the optimal portfolio problem so it must be solved as an optimization problem.

Several works such as [9] suggest solving the efficient frontier using a gradient based approach. However, the approach is very slow and can lead to local maximal and minimal based on the starting position and the seed vector. Bailey and Prado point out that the gradient based optimization approach "require a separate run for each portfolio" in the efficient frontier; this might explain the slow computation for gradient based optimization algorithms. They continued to state that the Critical Line Algorithm is specifically designed for the inequality constrained optimization problem and it guarantees an exact solution after a set number of iterations. The Critical Line Algorithm also derives the whole frontier of efficient portfolios in one run vs multiple runs for each portfolio as experienced by gradient based approaches.

A. Various Approaches

Some literature points towards setting up the problem as a convex function and then using quadratic programming to solve the problem with linear constraints [9] or non-linear programming for convex functions with non-linear constraints. While others use heuristic approaches such as firefly algorithm [10] or genetic algorithms. Mixed integer programming is another approach which came up when we searched for various optimization methods of the portfolio.

B. Optimization Problem

The solution set is found by setting up the problem as a quadratic problem subject to linear constrains in inequalities and one linear constraint in equality [5]. In order to understand the model, it is first important to know the variables Bailey and Prado used for the model in [5]:

- N = 1,2,3 contain the indices for the asset universe
- ω is the row vector of the weights assigned to the assets (a optimizer variable that needs to be solved)
- l is the row vector with the lower bound for the weights $\omega_i \geq l_i$, for all i in N
- u is the row vector with the upper bound for the weights $\omega_i \leq U_i$, for all i in N
- F $\subseteq$ N represents the "assets that don't lie on their boundaries" where $l_i \leq \omega_i \leq \mu_i$
- B $\subseteq$ N represents the "assets that lie on their boundaries" where $1 \leq k \leq n$ and B $\cup$ F = N
- $\sum = \begin{bmatrix} \sum_F & \sum_{FB} \\ \sum_{BF} & \sum_B \end{bmatrix}$, $\mu = \begin{bmatrix} \mu_F \\ \mu_B \end{bmatrix}$, $\omega = \begin{bmatrix} \omega_F \\ \omega_B \end{bmatrix}$

  where $\sum_F$ is the covariance matrix for Free assets, $\sum_{FB}$ is the covariance between free assets F and assets lying on the boundary B, $\sum BF$ is the transpose of the $\sum_{FB}$, μ_F is the row vector for the means of the free weights, μB is the row vector for the means for the assets lying on the boundary, ω_F is the row vector for the weights assigned to the free assets and ω_B is the

row vector for the weights assigned to the assets lying on the boundary.

The solution to the problem is obtained by minimizing the Lagrange Function with respect to the weights in the row vector row and its multipliers sigma and gamma. The Lagrange Function is shown below:

$$L[\omega, \gamma, \lambda] = \frac{1}{2}\omega\prime\sum\omega - \gamma\left(\gamma\prime 1_n - 1\right) - \lambda\left(\omega\prime\mu - \mu_p\right) \textbf{ (5)}$$

Bailey and Prado used a modified version of the above function. They stated due to the conditional inequalities they would not be able to solve for the variables or would be required to use the Karush-Kuhn-Tucker conditions. The Karush-Kuhn-Tucker conditions are used in mathematical optimization to set up first order conditions so that the solution is optimal in non-linear programming [11]. Instead they used the divide and conquer method where they converted the constrained problem into an unconstrained problem by using the "two mutual funds theorem." This is shown below:

$$\begin{aligned} L\left[\omega, \gamma, \lambda\right] &= \tfrac{1}{2}\omega_F\prime\textstyle\sum_F\omega_F + \tfrac{1}{2}\omega_F\prime\sum_{FB}\omega_B \\ &+ \tfrac{1}{2}\omega_B\prime\textstyle\sum_{BF}\omega_F\tfrac{1}{2}\omega_B\prime\sum_B\omega_B - \gamma(\omega_F\prime 1_k + \omega_B\prime 1_{n-k} - 1 \\ &- \lambda\left(\omega_F\prime\mu_F + \omega_B\prime\mu_B - \mu_p\right) \textbf{ (6)} \end{aligned}$$

By applying the "two mutual funds theorem", they didn't have to worry about the constraints. The efficient frontier was computed by deriving a convex combination between any two neighbour turning points. The pseudo code below outlines the critical line algorithm used for the sequential code provided by Bailey and Prado in [5]:

1: initialize the critical line class
2: initialize the algorithm
3: Determine which assets could be added to F (free assets)
4: Derive the B set from F set
5: Prepare/reduce the matrices for $\sum_F$,$\sum_{FB}$, μ_F , ω_B
6: Compute λ (lambda)
7: Decide λ (choosing the best alternative between λs)
8: Decide which asset could be removed from the F set
9: Compute the new turning point that is associated with the new F set
10: Compute and store the new solution vector
11: Compute the minimum variance portfolio and store
12: Search for the minimum variance portfolio
13: Search for the Maximum Sharpe Ratio portfolio
14: Compute the Efficient Frontier

V. Hardware and Languages

The algorithms were executed on the CUBIX by the CUBIX Corporation. The box has 8 NVIDIA GTX-780 cards, 2 Intel Xeon E5 2620s and 65.9 Gigabytes of Ram. The GPU code utilized only one of the GPUs. The Xeon E5 has 6 CPU cores with each core having a base processor frequency of 2 GHz with a turbo frequency of 2.5 GHz. The NVIDIA GTX-780 card has 2304 CUDA cores with a base clock of 863 MHz and a boost clock of 900 MHz. Each card had the ability to launch 65535 blocks with 1024 threads per block. The tests were all conducted on the CUBIX BOX to ensure that the testing was consistent.

Bailey and Prado wrote the sequential algorithm in python (2.xx) so we decided to keep the code in Python. The GPU code was written in Pycuda because it allowed us to access Nvidia's CUDA API from within Python [12]. The kernels in Pycuda are still written in C. However using Python allowed us to keep all the non-parallelized code in python without the need for modifications.

VI. Data Collection

The data gathered for the trials was in the same format as specified in their paper [5]. The assets were collected from Yahoo and contained daily returns for each asset for a little over two years. We only used stocks as the assets for the portfolio optimization to keep the data gathering process simple. The size of the assets collected varied. We originally tried to start off at one hundred assets and increment the number of assets by fifty up to five-hundred assets. We dropped any asset that had missing value(s) for daily returns. This resulted in the number of assets per file not being increments of 50 and the asset universe varying in size based on the number of assets that were dropped.

Instead of spending too much time on controlling then number of assets in each file, we decided to focus instead on making sure that the data was clean and consistent. We therefore didn't try to control the increment in the number of assets. The end results was we had files with the following number of assets: 86,118,139,160,174,257,310,346,390,463, and 501. This allowed us to test the Critical Line Algorithm on various asset sizes.

Once data was collected, we computed the average daily return for the each asset and computed the co-variance matrix for the asset set. This was stored in a file in the same format that Bailey and Prado used for their paper [5]. Once the data had been cleaned and stored, we were able to utilize the critical line algorithm written by Bailey and Prado.

VII. Sequential Execution

The sequential code was slightly modified. The Bailey and Prado wrote the all the methods within a class. However due to minimal amount of comments and nature of Python language we decided to bring all the functions outside of the class so that we could modify and test each function as we needed. We also modified the main driver slightly so that we could profile and test the code using various asset numbers. However, all the modifications we made left the computations and the control flow intact.

The code was profiled to both time the program completion and determine which part(s) of the program consumed the greatest part of the completion time. This allowed for a more precise target to parallelize using CUDA. After profiling the code it was determined that the majority of the time was spent

def getMatrices$(dataMean, dataCovar, solutionSet, f)$:
$covarF \leftarrow reduceMatrix(dataCovar, f, f)$
$meanF \leftarrow reduceMatrix(dataMean, f, [0])$
$b \leftarrow getB(data_mean, f)$
$covarFB \leftarrow reduceMatrix(dataCovar, f, b)$
$wB \leftarrow reduceMatrix(solutionSet[-1], b, [0])$
return $covarF, covarFB, meanF, wB$

Figure 1. getMatrices Function

in the reduce matrix function. The purpose of the function is to derive a sub matrix for a set of assets from a larger matrix.

The getMatrices function calls the reduceMatrix function as shown in Figure 1. The reduceFunction in Figure 2 contains two for loops, with the first for loop extracting specified columns and appending them to an intermediate matrix. The second for loop extracts specified rows from the intermediate matrix and appends them to the return matrix. The loops rely heavily on the numpy.append function which calls the numpy.concatenate function. After profiling the code we were able to determine that the reduceMatrix function was the bottleneck for the program and ideal for parallelization on the gpus.

def reduceMatrix$(inputMatrix, listX, listY)$:
if if either lists are empty **then**
  return
end if
$intermMatrix \leftarrow$ first column of inputMatrix
for every ith column position stored in listY **do**
  $temp \leftarrow inputMatrix$[:,i:i+1]
  $intermMatrix \leftarrow$ np.append(intermMatrix,temp,1)
end for
returnMatrix $\leftarrow$ first row from intermMatrix
for every jth row position stored in listX **do**
  temp $\leftarrow$ intermMatrix[j:j+1,:]
  $returnMatrix \leftarrow$ np.append(returnMatrix,temp,0)
end for
return returnMatrix

Figure 2. reduceMatrix Function

VIII. Parallel Execution

In order to parallelize the reduceMatrix function we had to modify the getMatrices function. Each instance of the getMatrices calls the reduceMatrix four times: twice to reduce two NXN matrices (both covariance matrices) and twice to reduce two row vectors (one for average returns of the assets, and the other for the weights of those assets). We determined through trial and error that parallelizing the calls for the row vectors was not efficient and led to an increase in computation time. We therefore created a GPU version of the reduce matrix which was called only when reducing an NXN matrix and utilized the sequential code written by the authors for the row vector reductions. The modification for the getMatrices function is shown in figure 3.

def getMatrices(dataMean, dataCovar,solutionSet,f, intermMatrix, deviceCovar):
covarF $\leftarrow$ gpuReduceMatrix(dataCovar,f,f,intermMatrix, deviceCovar)
meanF $\leftarrow$ cpuReduceMatrix(dataMean,f,[0])
$b \leftarrow getB(data_mean, f)$
covarFB $\leftarrow$ gpuReduceMatrix(dataCovar,f,b,intermMatrix, deviceCovar)
wB $\leftarrow$ cpuReduceMatrix(solutionSet[-1],b,[0])
return covarF,covarFB,meanF,wB

Figure 3. modified getMatrices Function

As the code in Figure 3 shows, the function takes in two additional parameters, the intermMatrix and the deviceCovar. When the algorithm is initialized, we pre-allocated two large matrix (based on the number of assets) that would be large enough to handle any reduced matrices for the duration of the program. This saved us the overhead of creating new matrices every time the reduceMatrix function was called. This decreased the overhead associated with transferring data from host to device. The intermMatrix was used to extract all the necessary columns from the input matrix; it acted as the temporary matrix from which we extracted specific rows for the reduced return matrix. The deviceCovar matrix contained the covariance matrix for the complete asset population in the device memory. This allowed us to pull data from the device version of the inputMatrix without having to transfer data back and forth from host to device. The pseudo code is shown in Figure 4 for the reduceMatrix function.

The gpuReduceMatrix function invoked the kernels to make the process parallel.We wrote two kernels to parallelize the for loops in the reduceMatrix function. We didnt use dynamic parallelism, instead we opted to write the code so that each block mapped an iteration of the for loop for both the column and row reductions. Figure 5 shows the kernel that replaced the for loop to reduce the columns.

IX. Test Results

All of the GPU implementations had a speed up over the sequential trial after a given threshold as seen in the timings in Figure 7. So it makes sense that once the number of assets grow, the computations should be shifted over to the GPU. However, one of the interesting procedures we tried was a hybrid approach where prior to a certain threshold, the CPU would run the computations after which time the GPU would take over. This threshold was the number of elements in the co-variance matrix ($rowsxcolumns$). After trying different thresholds we were able to see that the optimal CPU/GPU combination was at fifty elements in the co-variance matrix. That is when the number of elements in the matrix was below fifty the CPU reduced the matrix otherwise the GPU reduced the matrix for the co-variance matrices.

def gpuReduceMatrix(inputMatrix,listX,listY, intermMatrix, deviceCovar):
if listX or listY are empty **then**
  return from function
end if
if the size of the return matrix is $\geq$ specified threshold **then**
  numberOfCols $\leftarrow$ num of columns stored in listY
  numberOfColsInterm $\leftarrow$ real columns in intermMatrix
  numOfThreads $\leftarrow$ num of threads column reduction
  numOfBlocks $\leftarrow$ length(listY)
  paddingSize $\leftarrow$ padding size column reduce
  size $\leftarrow$ size to emulate for the intermMatrix
  gpuListY $\leftarrow$ transfer listY from host to device
  launch columnReductionKernel with above variables
  numOfBlocks $\leftarrow$ num of blocks row reduction
  numOfThreads $\leftarrow$ num of threads row reduction
  numOfColsReturnMatrix $\leftarrow$ num of columns returnMatrix
  sizeOfReturnMatrix $\leftarrow$ the size of the returnMatrix
  gpuListX $\leftarrow$ copy listX onto the device from the host
  launch the rowReductionKernel with variables above
  copy the return matrix from device to host
  cpuReturnMatrix $\leftarrow$ gpuReturnMatrix
  return cpuReturnMatrix
else
  reduce the columns and rows using sequentail code
  return cpuReturnMatrix
end if

Figure 4. modified reduceMatrix Function

void appendColumn(float* dest, float* src, int* posList, int numColsAppendMatrix, int padding,int size, int realColumns)
tid $\leftarrow$ threadIdx.x
appendPos $\leftarrow$ blockIdx.x
extractPos $\leftarrow$ colum pos to append based on listY[blockIdx.x]
realNumCol $\leftarrow$ real numb of cols in src matrix
declare variables: stride, index, mimickedIndex, globalTid, rowPos
for i=0, index=0, strid=0; mimickedIndex $<$ sizeOfInterMMatrix; ++i **do**
  globalTid += tid*realNumCol+extractPos
  stride $\leftarrow$ blockDim.x*i+tid
  index $\leftarrow$ stride*(realNumCol+padding)+appendPos
  rowPos $\leftarrow$ (index/realColumns);
  dest[index] $\leftarrow$ src[globalTid];
end for

Figure 5. column reduce kerenel

void rowAppend(float* matrix, float* inputMatrix,int* rowList, int numCols, int padding, int sizeOfInputMatrix)
tid $\leftarrow$ threadIdx.x
stride $\leftarrow$ blockDim.x
rowPosDest $\leftarrow$ blockIdx.x
rowPosSrc $\leftarrow$ rowList[blockIdx.x]
destIndex $\leftarrow$ rowPosDest * numCols+tid
srcIndex $\leftarrow$ rowPosSrc * (numCols+padding)+tid
while iterator $<$ numCols **do**
  matrix[destIndex] $\leftarrow$ inputMatrix[srcIndex]
  srcIndex += stride
  destIndex += stride
  iterator +=stride
end while

Figure 6. row reduce kerenel

Num Assets	seq	GPU	50_Element_Threshold
10	0.087	0.406	0.342
86	7.138	12.858	9.484
118	23.694	28.842	25.655
139	24.62	31.586	23.299
160	46.224	47.26	38.223
174	57.593	43.222	41.968
257	256.207	103.2	107.755
310	472.257	145.633	151.866
346	815.848	221.863	226.384
390	1243.621	284.487	256.166
463	2248.878	359.467	350.585
501	4189.16	518.535	525.206

Figure 7. Table 1: Completion Time(s) for Sequential, Parallel and 50 Element Threshold

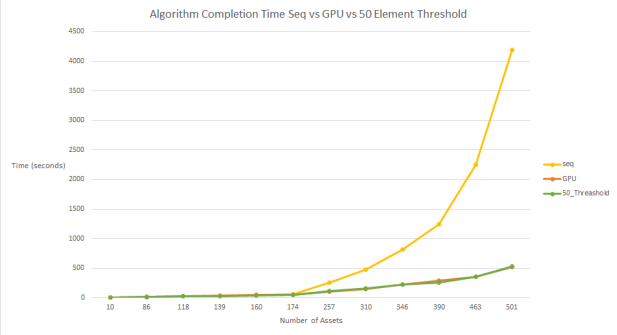

Figure 8. Graph 1: Completion Time(s) for Sequential, Parallel and 50 Element Threshold

Figures 8 and 9 show the graphs of the completion times for the sequential vs parallel vs the threshold method. The 50 threshold method runs slightly faster than the GPU reduceMatrix however after 257 assets, the GPU version runs slightly faster. This can be seen in Figure 9 where the graph has been normalized by taking log base 2 of the computation times.

It is easier to see speed up between the reduceMatrix running on the GPU for all covariance matrix reductions and the threshold approach in Figures 10 and 11. The sequential implementation runs at a much faster time for a small number

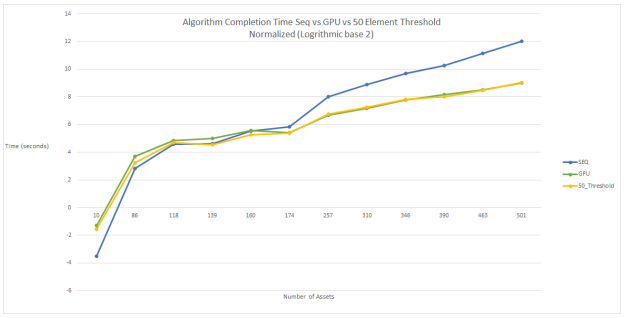

Figure 9. Graph 2: Lognormal Completion Time(s) for Sequential, Parallel and 50 Element Threshold

Num Assets	GPU	50_Element_Threshold
10	0.21428571	0.254385965
86	0.55514077	0.752636019
118	0.8215103	0.923562658
139	0.77945925	1.056697712
160	0.97807871	1.209324229
174	1.33249271	1.372307472
257	2.48262597	2.37768085
310	3.24278838	3.109695389
346	3.67726029	3.603823592
390	4.37145107	4.854746532
463	6.2561459	6.414644095
501	8.07883749	7.976222663

Figure 10. Speed Up for Parallel and 50 Element Threshold

of assets, however once the number of assets exceeds 139, the 50 element threshold method starts getting speed up with the GPU version attaining speed up at 174 assets and above. However, please note at around 500 assets, the GPU method has a slightly higher speed up than the threshold method. This was tested multiple times, and it seems that after 500 assets, when the GPU does all the reductions for the covariance matrix, it will be faster than both the sequential and the 50 element threshold method.

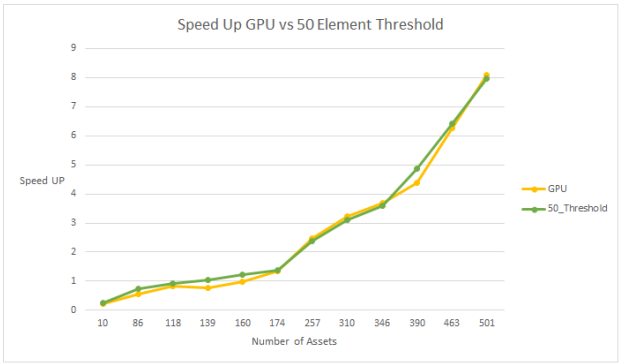

Figure 11. Speed Up Parallel vs 50 Element Threshold

Subsubsection text here.

X. Conclusion

Prior to parallelizing the bottleneck of the Critical Line Algorithm on the GPU, we did have an inkling that there would be speed up. However, we were not sure about the extent of speed up. However after the results, we were able to determine that the speed up tends to grow in relation to the number of assets. The most speed up we were able to attain was 8X due to the max number of assets we utilized.

The method in which the Critical Line Algorithm has been optimized, it can handle 65535 assets on one GPU before it will require additional GPUs for the matrix reduction. However, we would expect a slowdown after 1024 assets because the threads would have to stride to reduce the columns and rows due to the device property limitations. The purpose of this paper was to speed up the Critical Line Algorithm presented in [5] and to hopefully encourage others into utilizing the power of GPUs for computational finance and publishing the results.

XI. Future Work

There are several modifications that can be made to the Critical Line Algorithm, such as including negative weights for short sales. The speed up achieved was the result of parallelization of a small part of the code. However, as the number of assets increase, we believe the other parts of the algorithm will start adding to the time and could be viable for parallelization.

Acknowledgment

We would like to thank D.H. Bailey and M. Lopez de Prado for releasing their algorithm and filling the gap between theory and practice, as they had intended.

References

[1] H. Markowitz “Portfolio Selection.” The Journal of Finance 7.1 (1952): 77. JStor. Web. 28 Mar. 2015.
[2] P. D. Kaplan.(2014).Back to Markowitz [Online]. Available: http://corporate.morningstar.com/US/documents/Indexes/Back-To-Markowitz-2014.pdf
[3] P. Romero and T. Balch, What Hedge Funds Really Do: An Introduction to Portfolio Management.
[4] C. C. Kwan (2007) “A Simple Spreadsheet-Based Exposition of the Markowitz Critical Line Method for Portfolio Selection,” Spreadsheets in Education (eJSiE): Vol. 2: Iss. 3, Article 2
[5] D.H. Bailey, M. Lopez de Prado. An Open-Source Implementation of the Critical-Line Algorithm for Portfolio Optimization. Algorithms 2013, 6, 169-196
[6] N. Stchedroff. (2013), Portfolio Optimization. Wilmott, 2013: 5257. doi: 10.1002/wilm.10184
[7] J. Hu, Stock Portfolio Optimization Using CUDA GPU, unpublished.
[8] M. Rubinstein. (2002), Markowitzs Portfolio Selection: A Fifty-Year Retrospective. The Journal of Finance, 57: 10411045. doi: 10.1111/1540-6261.00453
[9] Y. Hilpisch. “Chapter 11. Statistics.” Python for Finance:. Beijing: O’Reilly, 2015. N. page. Print
[10] M. Tuba and N. Bacanin. Upgraded firefly algorithm for portfolio optimization problem. In UKSim-AMSS 16th International Conference on Computer Modelling and Simulation, 2014.
[11] R. W. Cottle. “William Karush and the KKT Theorem.” Documenta Mathematica Extra (2012): 255-69. Web.
[12] A. Kloeckner. Welcome to PyCUDA’s Documentation! PyCUDA 2014.1 Documentation. N.p., n.d. Web. 28 Mar. 2015