# Accelerating the Critical Line Algorithm for Portfolio Optimization Using GPUs

Raja H. Singh[1], Lee Barford[1,2], and Frederick Harris, Jr.[1]

[1] Department of Computer Science and Engineering, University of Nevada, Reno
[2] Keysight Laboratories, Keysight Technologies
rajas@nevada.unr.edu, lee_barford@ieee.org, Fred.Harris@cse.unr.edu

**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.

**Keywords:** Cuda, Markowitz, Stocks, Parallel Computing, Pycuda

## 1 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.

For this paper we utilized the open source (critical line) algorithm for portfolio optimization 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.

## 2 Related Works

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.

We found two papers that address portfolio optimization on the GPUs [6] and [7]. The [6] written by Stchedroff used the Multi Directional optimization which is an example of the Direct Search method, to find an optimal portfolio from an asset population. The sequential code was executed on an unspecified CPU using multiple threads and the parallel code was written on an unspecified GPU. Stchedroff did mention using CUBLAS, BIAS and BLAS libraries for the matrix computations. According to his results he achieved a 10x speed up on the GPU vs CPU and a 60x speed up for the combined GPU and new algorithm approach. However the author didn't provide any psuedo code for the sequential or parallel implementations, and didn't specify any information on the CPU and GPU hardware.

The [7] written by Hu used simulated anealing to derive the optimal portfolio from randomly generated portfolios from asset population of twenty. He provided one psuedo algorithm for his portfolio selection process but didn't provide any psuedo code for the Cuda kernels. He mentioned a 4X speed up using two Nvidia GTX295 cards vs the sequential code executed on Intel i7 Quadcore. Besides these two papers we haven't found other papers that implemented any portfolio optimization algorithms on the GPU.

## 3 Portfolio Optimization

**Concept:** A vital assumption Markowitz's model makes is that "the investor does (or should) consider expected return a desirable thing and variance of return an undesirable thing [1]." 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.

**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 $\sum_{i=1}^{N} X_i = 1$. The Expected Return of the whole portfolio is expressed as: $E = \sum_{i=1}^{N} X_i \mu_i$. Where $X_i$ is treated as a random variable representing the weight assigned to an asset and $\mu_i$ 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 can be represented as: $\sigma_{ij} = E[R_i - E(R_i)][R_j - E(R_j)]$, where $\sigma\sigma_{ij}$ is the co-variance between asset i and j.

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

$$V = \sum_{i=1}^{N} \sum_{j=1}^{N} \sigma_{ij} X_i X_j \tag{1}$$

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].

## 4 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.

**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.

**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 inequality [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
- $\omega$ 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
- the covariance matrix $\sum$, row vector $\omega$, and the mean vector $\mu$ are modeled below: $\sum = \begin{bmatrix} \sum_F & \sum_{FB} \\ \sum_{BF} & \sum_B \end{bmatrix}$, $\mu = \begin{bmatrix} \mu_F \\ \mu_B \end{bmatrix}$, $\omega = \begin{bmatrix} \omega_F \\ \omega_B \end{bmatrix}$
  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 $\sum_{FB}$, $\mu_F$ is the row vector for the means of the free weights,

$\mu B$ is the row vector for the means for the assets lying on the boundary, $\omega_F$ is the row vector for the weights assigned to the free assets and $\omega_B$ 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 $\omega$ and its multipliers $\gamma$ and $\lambda$. The Lagrange Function is shown in Equation 2

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

Bailey and Prado used a modified version of the function in Equation . 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." They modified the Lagrange Multiplier shown in Equation 2 to the format shown in Equation 3, changing the constrained problem to an unconstrained problem and computed the efficient frontier by deriving a convex combination between any two neighboring turning points. Due to the length of the algorithm we have not provided it in the paper, please refer to [5] for the complete algorithm.

$$\begin{aligned} L\left[\omega, \gamma, \lambda\right] = \frac{1}{2}\omega_F\prime\sum_F\omega_F + \frac{1}{2}\omega_F\prime\sum_{FB}\omega_B \\ +\frac{1}{2}\omega_B\prime\sum_{BF}\omega_F\frac{1}{2}\omega_B\prime\sum_B\omega_B - \gamma(\omega_F\prime 1_k + \omega_B\prime 1_{n-k} - 1) \\ -\lambda\left(\omega_F\prime\mu_F + \omega_B\prime\mu_B - \mu_p\right) \end{aligned} \tag{3}$$

## 5 Hardware

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. We didn't use multiple GPUs.

## 6 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. 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.

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. We also used the 10 asset file used by Prado and Bailey to do their own testing, this allowed us to confirm that the parallel algorithm didn't provide erroneous results. Once data was collected, we computed the average daily return for the each asset and computed the covariance matrix for the asset set.

## 7 Sequential Execution

The sequential code was slightly modified. Bailey and Prado wrote 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 contribute the most to 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 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. The reduceMatrix function 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. This function joins two arrays together. However, to accomplish this task it requires creating a new array large enough to hold all the values. With small size arrays this task is very fast and the overhead associated with it is trivial, but as the size of the matrices and the number of calls to the function increase, the overhead associated with this task takes a toll on the overall completion time. This was made evident by the profiling of the code and led us to parallelize the reduceMatrix function.

## 8 Parallel Execution

**Memory Transfer** The most significant overhead associated with parallel computing (GPU specific) comes from the transfer of data from the host to device

and back again. In order to minimize the impact of this overhead we created two large matrices when we first initialized the CLA class. The first matrix contained the covariance matrix for the complete asset population in the device memory (deviceCovar matrix) and the second matrix (intermMatrix) was a square matrix based on the asset population size was used to extract and hold all the necessary columns from the deviceCovar matrix; it acted as the temporary matrix that held the specified columns of the deviceCovar matrix. From the intermMatrix we extracted the specified rows for the reduced returnMatrix. By allocating a large bank of memory and then transferring the needed data to the device once during the initialization phase we reduced overhead associated with memory allocation and data transfer significantly. Once the data and the needed memory were on the device, we could utilize the two matrices repeatedly without incurring the overhead of transferring data from the host to the device repeatedly. However, we did have to transfer the reduced matrix from the device to the host at the end of the reduceMatrix function.

**reduceMatrix function** We created a new reduceMatrix function we named gpuReduceMatrix because we utilized the original reduceMatrix function for reduction of the row vectors. As stated previously, with a small asset population the overhead associated with the reduceMatrix function is very trivial. Keeping that in mind we created a branching instruction where we specified a threshold value that had to be met before we utilized the GPU version of the code else we executed the sequential code written by the Bailey and Prado. The threshold referred to the size of the deviceCovar matrix. If we set the threshold to zero then every time the gpuReduceMatrix was called it executed the parallel code. On the other hand if we specified a threshold then the matrices to be reduced had to meet the specified threshold size before the parallel code was executed. If the threshold was not met then the gpuReduceMatrix function used the same code as the sequential version. In the parallel code section of the gpuReduceMatrix function, we wrote two kernels that replaced the for loops that were executed in the sequential code. These two kernels are discussed later.

**Cuda Kernels** To parallelize the reduceMatrix function we wrote two cuda kernels and called them in the gpuReduceMatrix. The first one (columnAppend) appended the specified columns from the deviceCovar matrix to the intermMatrix. The second one (rowAppend) appended the specified rows from the intermMatrix to the gpuReturnMatrix. Inside the columnAppend kernel each block mapped to a column of the deviceCovar matrix based on its blockId and each block thread appended an element of that column to the intermMatrix. Inside the rowAppend kernel each block mapped to a specified row of the intermMatrix and each thread copied an element of a row to the gpuReturnMatrix.

Since the max size of the asset population was less than the max number of blocks the GPU card allowed us to launch, we didn't stride the blocks for columns or rows. Meaning each block only mapped one column in the columnAppend kernal or row in the rowAppend kernel. However, we did stride the the threads so each thread would be able to append more than one element in each kernel

if the number of elements in the column or row exceeded the max number of threads we could launch. Instead of iterating through a loop to append a single column/row per iteration we were able to append all the columns simultaneously and then all the rows simultaneously by launching the appropriate kernels with specified number of blocks and threads. Algorithm 1 and Algorithm 2 show the code for the cuda kernels.

**Algorithm 1** columnAppend Kernel

```
function APPENDCOLUMN(dest, src, posList, numColsAppendMatrix, padding, size,
realColumns)
    tid ← threadIdx.x
    appendPos ← blockIdx.x
    extractionPosDataMatrix ← posList[blockIdx.x]
    numColsDataMatrix ← realColumns
    for mimickedIndex < size do
        globalTid += tid × numColsDataMatrix + extractionPosDataMatrix
        stride ← blockDim.x×i + tid
        index ← stride × (numColsAppendMatrix + padding) + appendPos
        rowPos ← (index ÷ realColumns)
        mimickedIndex ← index  (rowPos × padding)
        if mimickedIndex ≥ size then
            break
        end if
        dest[index] ← src[globalTid]
    end for
end function
```

**getMatrices function** After we parallelized the reduceMatrix function we determined 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 computational time. We modified the getMatrices function to call the GPU version of the reduceMatrix function (gpuReduceMatrix) only for NXN square matrices and keep using the original sequential version of the reduceMatrix for the row vector reductions.

## 9 Results and Analysis

All of the GPU implementations had a speed up over the sequential trial after a given threshold. 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 threshold the GPU would take over. This threshold was the number of elements in the co-variance matrix

**Algorithm 2** rowAppend Kernel

```
function ROWAPPEND(destMatrix, srcMatrix, rowList, numCols, padding, sizeOfInputMatrix)
    tid ← threadIdx.x
    rowPosDest ← blockIdx.x
    rowPosSrc ← rowList[blockIdxX]
    destIndex ← rowPosDest × numCols+tid
    srcIndex ← rowPosSrc × (numCols+padding)+tid
    while iterator < numCols do
        if iterator > numCols then
            break
        end if
        matrix[destIndex] ← inputMatrix[srcIndex]
        srcIndex += stride
        destIndex += stride
        iterator += stride
    end while
end function
```

(rows × columns ). 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.

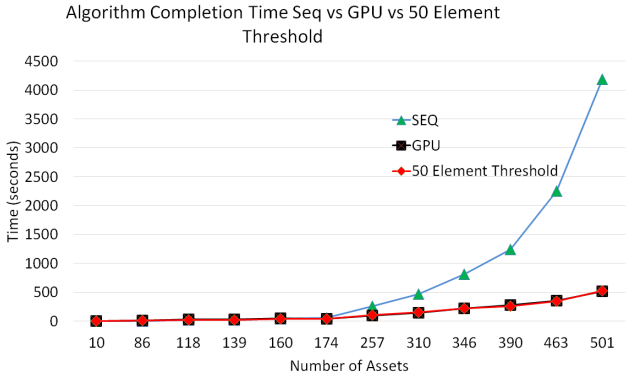


Fig. 1: Completion Time(s) for Sequential, Parallel and 50 Element Threshold

Graph in Figure 1 shows 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. It is easier to see speed up between the reduceMatrix running on the GPU

for all covariance matrix reductions and the threshold approach in the graph in Figure 2. The sequential implementation runs at a much faster time for a small number 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.

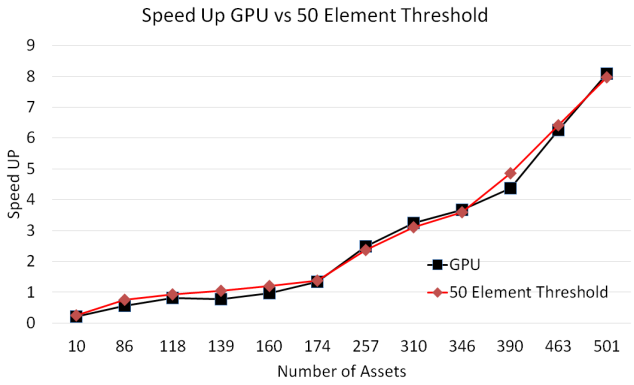


Fig. 2: Speed Up Parallel vs 50 Element Threshold

## 10 Conclusion and Future Work

In this paper we used GPUs to speed up the Critical Line Algorithm to optimize portfolios using Markowitz's Efficient Frontier presented by Bailey and Prado. Prior to parallelizing the bottleneck of the Critical Line Algorithm on the GPU, we did have an inkling that there would be speed up. 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 with an asset population of 501 asset was around 8X.

However, we believe as the size of the asset population increases, the speed up we attain in relation to the sequential completion time will grow drastically. 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. 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.

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. This material is based in part upon work supported by: The National Science Foundation under grant number(s) IIA-1329469, and by Cubix Corporation through use of their PCIe slot expansion hardware solutions and HostEngine. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Cubix Corporation.

## References

1. H. Markowitz "Portfolio Selection." The Journal of Finance 7.1 (1952): 77. JStor. Web. 28 Mar. 2015.
2. P. Kaplan, 'Back to Markowitz', Morningstar, 2014. [Online]. Available: http://corporate.morningstar.com/US/documents/Indexes/Back-To-Markowitz-2014.pdf. [Accessed: 05- May- 2015].
3. T. Balch, P. Romero, 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), Markowitz's "Portfolio Selection: A Fifty-Year Retrospective. The Journal of Finance, 57: 10411045. doi: 10.1111/1540-6261.00453
9. Y. Hilpisch, Python for Finance: Analyze Big Financial Data. Beijing: O'Reilly Media, 2014.
10. N. Bacanin , M. Tuba. Upgraded firefly algorithm for portfolio optimization problem. In UKSim-AMSS 16th International Conference on Computer Modelling and Simulation, 2014.
11. R. Cottle, 'William Karush and the KKT Theorem',Documenta Mathematica, Extra Vol, pp. 255-269, 2010.
12. A. Kloeckner "Welcome to PyCUDA's Documentation!, PyCUDA 2014.1 Documentation, 2015. [Online]. Available: http://documen.tician.de/pycuda/. [Accessed: 28-Mar-2015]