INTERNATIONAL INSTITUTE OF
INFORMATION TECHNOLOGY
HYDERABAD

Introduction to Parallel Scientific Computing

Course Project Report

“Parallel Logistic Regression Using CUDA”

<u>Submitted By</u> :

Aman Agrawal (2018201006)
Archit Kumar (2018201051)
Tarpit Sahu (2018201089)

Introduction

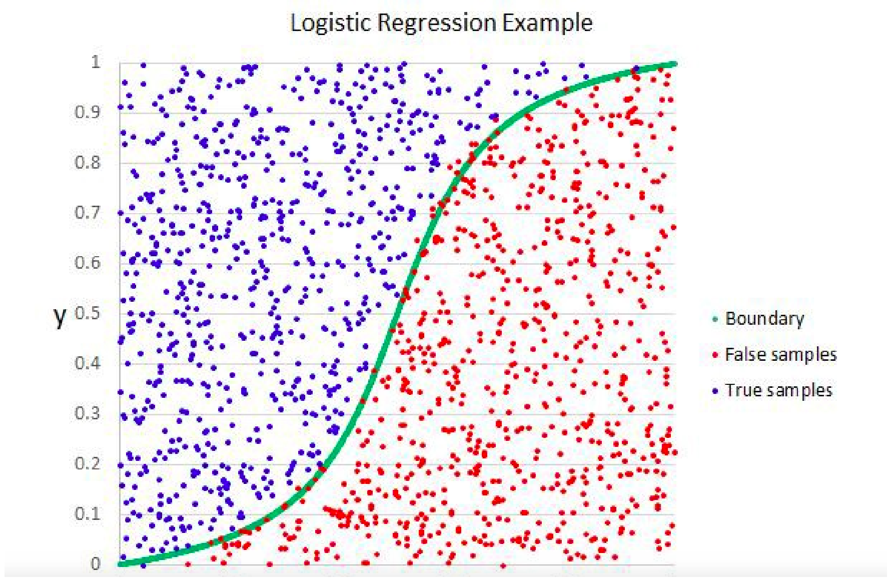
Logistic Regression is a Machine Learning classification algorithm that is used to predict the probability of a categorical dependent variable. In logistic regression, the dependent variable is a binary variable that contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.). In other words, the logistic regression model predicts P(Y=1) as a function of X.

Logistic Regression is one of the most popular ways to fit models for categorical data, especially for binary response data in Data Modeling. It is the most important (and probably most used) member of a class of models called generalized linear models. Unlike linear regression, logistic regression can directly predict probabilities (values that are restricted to the (0,1) interval); furthermore, those probabilities are well-calibrated when compared to the probabilities predicted by some other classifiers, such as Naive Bayes. Logistic regression preserves the marginal probabilities of the training data. The coefficients of the model also provide some hint of the relative importance of each input variable.

Logistic Regression is used when the dependent variable (target) is categorical.

For example,

To predict whether an email is spam (1) or (0)
Whether the tumor is malignant (1) or not (0)



Need of Parallel Logistic Regression

Parallel processing is much faster than sequential processing when it comes to doing repetitive calculations on vast amount of data. This is because a parallel processor is capable of multithreading on a large scale, and can therefore simultaneously process several streams of data. The one advantage of parallel processing is that it is much faster for simple, repetitive calculations on vast amounts of similar data. If a difficult computational problem needs to be attacked, where the execution time of program code must be reduced, parallel processing may be useful.

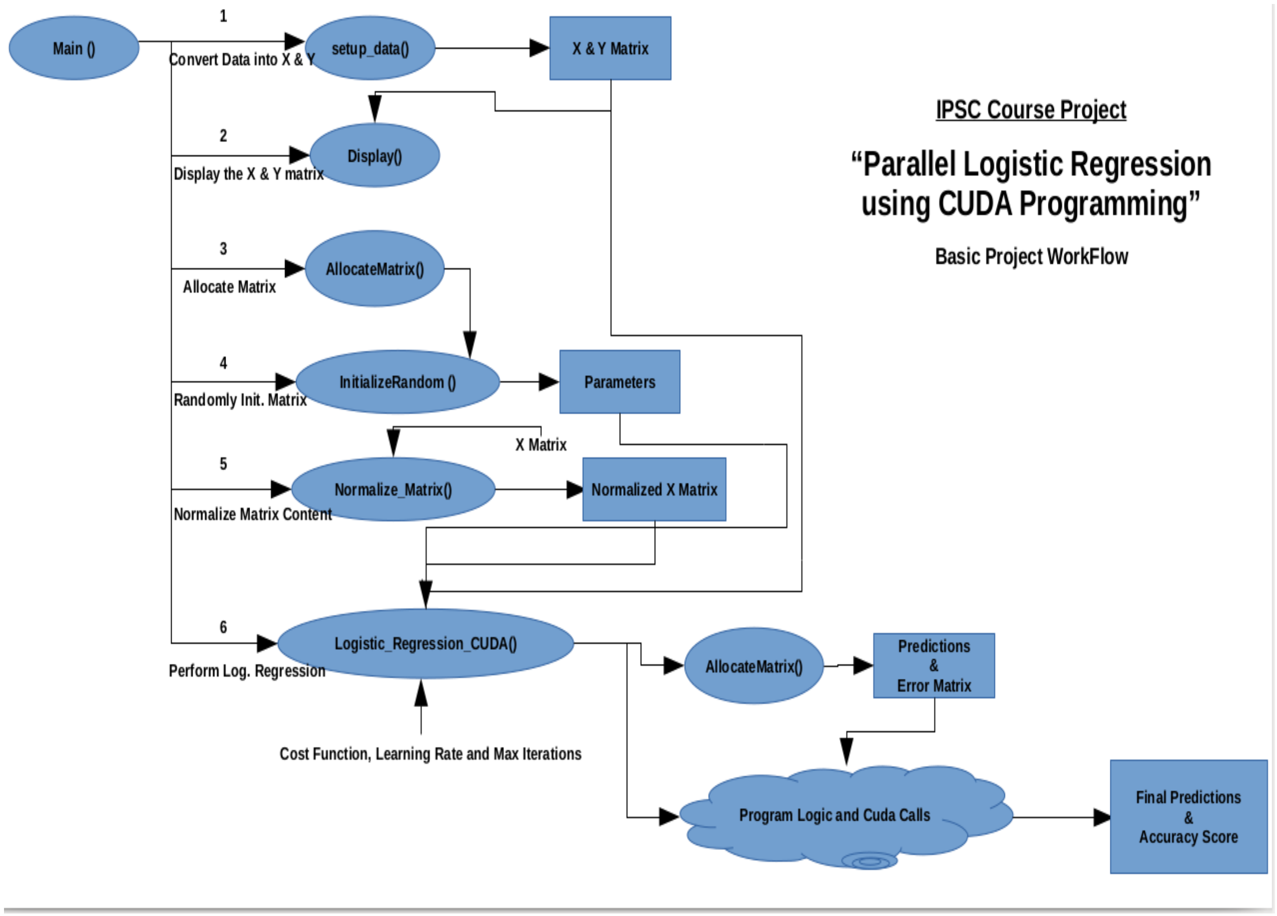
The logistic regression model plays a pivotal role in machine learning tasks. The model is suited for classification problems and is supported by a substantial body of statistical theories. In recent years, many modern datasets have grown drastically in both data volume and data dimensionality. Large data volume and high data dimensionality bring both resource and computational challenges to machine learning algorithms.
To reduce computational time , we use parallel programming based Logistic Regression.

Statistics Of Logistic Regression

Logistic regression is a probabilistic, linear classifier. It is parameterized by a weight matrix W and a bias vector b. Classification is done by projecting an input vector onto a set of hyperplanes, each of which corresponds to a class. The distance from the input to a hyperplane reflects the probability that the input is a member of the corresponding class.
It is split in two phases. First of all there is the training phase wherein one sample is taken from the input dataset to learn the best value of the parameters W and b.
Finally, there is the test phase where model previously trained is tested to see the response.

Project Work-Flow

The following flowchart demostrates the basic workflow of the project. The execution of our project starts from main() function and exapands further as shown in the figure below.



Program Structure

1. DataStructure

Matrix Structure

Matrix()	:	Constructor to initialize the variables
~Matrix()	:	Destructor to clear variables after use
Width, Height, Pitch	:	Integer variables determining matrix dimensions
Elements	:	Float array to store matrix elements.

2. Functions

*a) InitializeMatrix(Matrix *mat, int x, int y, float val)*

Input	:	a matrix, x and y cordinates and a value
Job	:	sets matrix[x][y]=value

*b) Matrix_Element_Required(Matrix *mat, int x, int y)*

Input	:	a matrix, x and y cordinates
Job	:	returns matrix[x][y]

*c) AllocateMatrix(Matrix *mat, int height, int width)*

Input	:	Matrix pointer, height and width of the matrix
Job	:	Allocates space for the matrix and initializes all the elements with 0.0.

d) DisplayMatrix(Matrix &mat, bool force = false)

Input	:	a Matrix
Job	:	Print the elements of the Matrix

*e) setup_data (string file_name, Matrix *X, Matrix *y)*

Reads a dataset file and converts it into X and Y matrices. X contains the features whereas the Y contains the labels.

*f) Normalize_Matrix_min_max(Matrix *m)*

It normalizes the input matrix. Follows Min-Max normalization.

Normalized Value = (Old Value – Min Value) / Max Value

g) InitializeRandom

It initializes the Input Matrix with Randmo Values.

h) CheckCudaErrorAux (const char *file, unsigned line, const char *statement, cudaError_t err)
Checks and Displays the caught CUDA errors.

i) #define SAFE_CALL(value) CheckCudaErrorAux(__FILE__,__LINE__, #value, value)
SAFE_CALL is a macro function that wraps CheckCudaErrorAux function.

j) sigmoidKernel(float *r, int m)
function to compute sigmoid

k) matrixAbsErrorKernel(float *p, float *ys, float *r, int rw, int rh)
function to compute Matrix Absolute Error

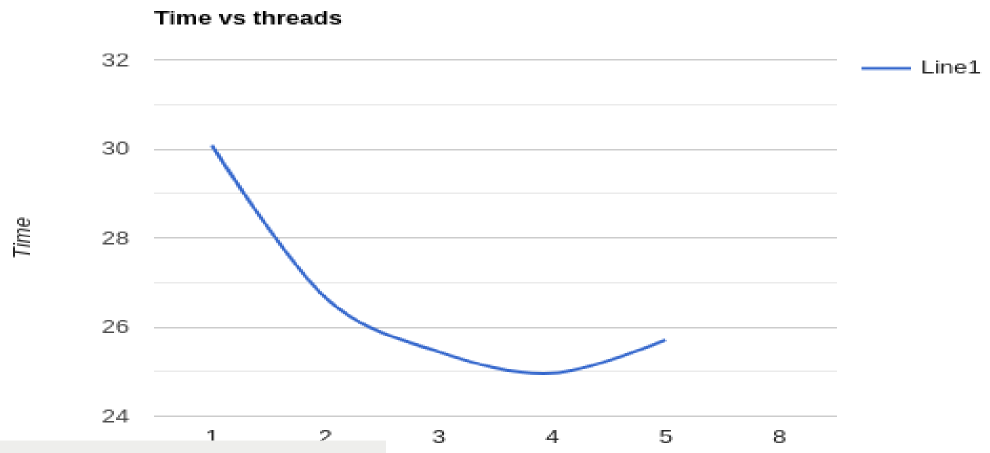
l) absErrorKernel(float *p, float *ys, float *r, int m)
function to compute Absolute Error

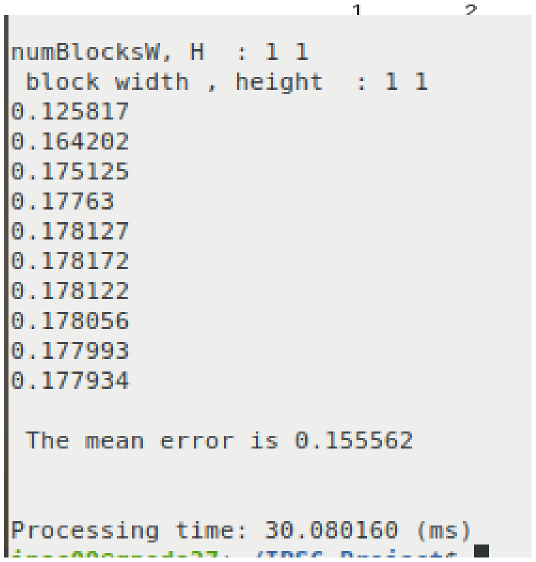
m) updateParamsAbsErrorKernel(float *p, float *ys, float *th, float *xs, int m, float alpha)
function to update the parameters

n) crossEntropyKernel(float *p, float *ys, float *r, int m)
function to determine the cross-entropy

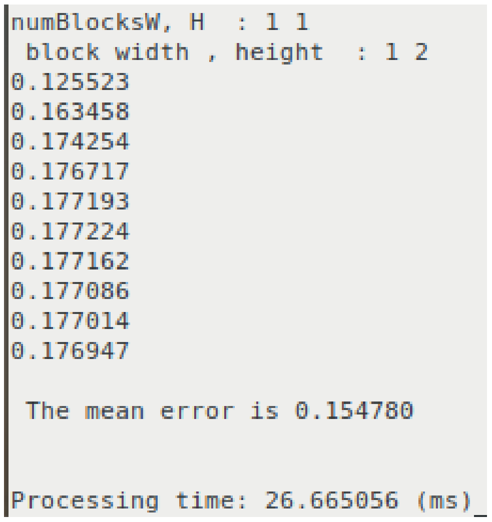
o) Logistic_Regression_CUDA(Matrix *X, Matrix *y, Matrix *Parameters, Matrix *Train_Parameters, int maxIterations, float alpha, vector<float> &cost_function)
The main function of our code that performs the Logistic Regression and calls various other functions specified above.

Results and Observations

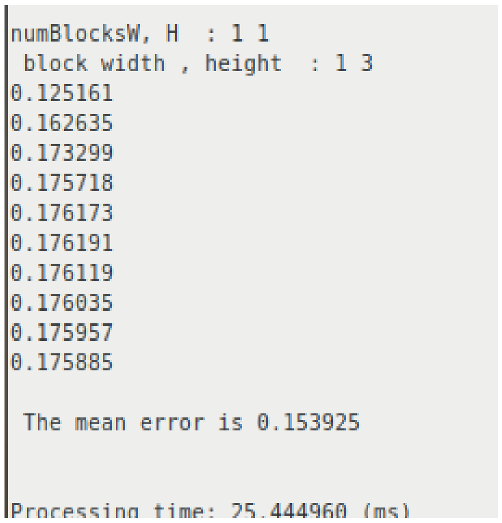




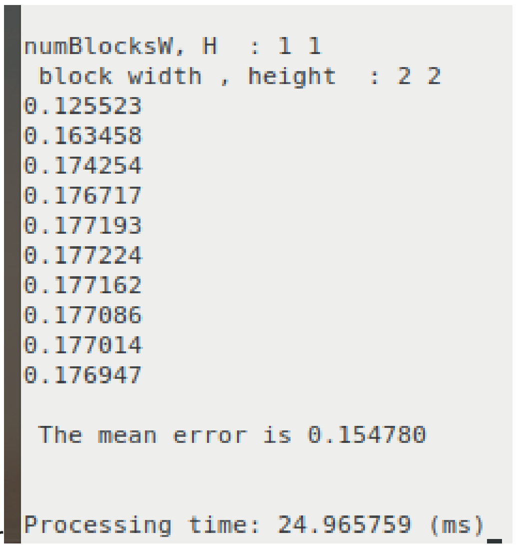
```
numBlocksW, H  : 1 1
 block width , height  : 1 1
0.125817
0.164202
0.175125
0.17763
0.178127
0.178172
0.178122
0.178056
0.177993
0.177934

 The mean error is 0.155562


Processing time: 30.080160 (ms)
```



```
numBlocksW, H  : 1 1
 block width , height  : 1 2
0.125523
0.163458
0.174254
0.176717
0.177193
0.177224
0.177162
0.177086
0.177014
0.176947

 The mean error is 0.154780


Processing time: 26.665056 (ms)
```

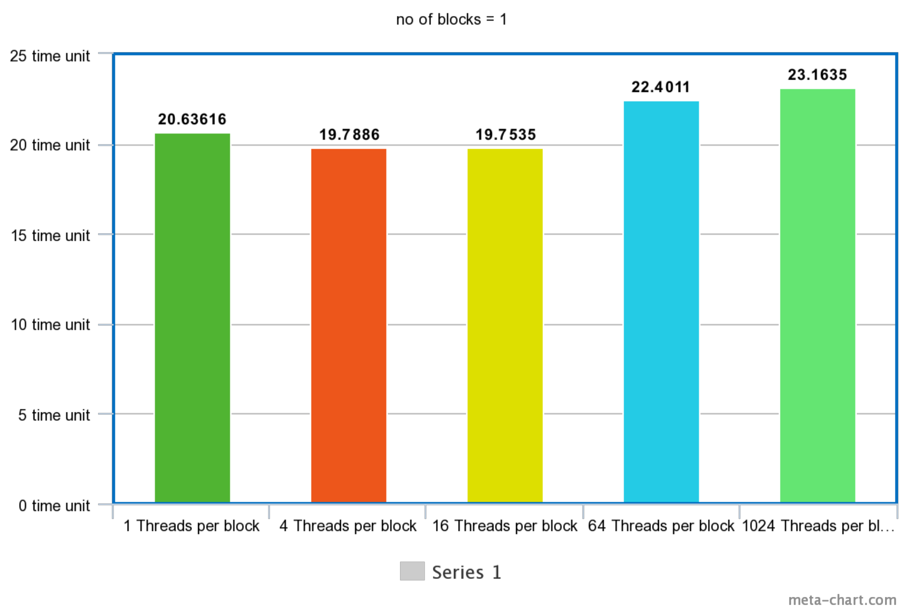


```
numBlocksW, H  : 1 1
 block width , height  : 1 3
0.125161
0.162635
0.173299
0.175718
0.176173
0.176191
0.176119
0.176035
0.175957
0.175885

 The mean error is 0.153925


Processing time: 25.444960 (ms)
```

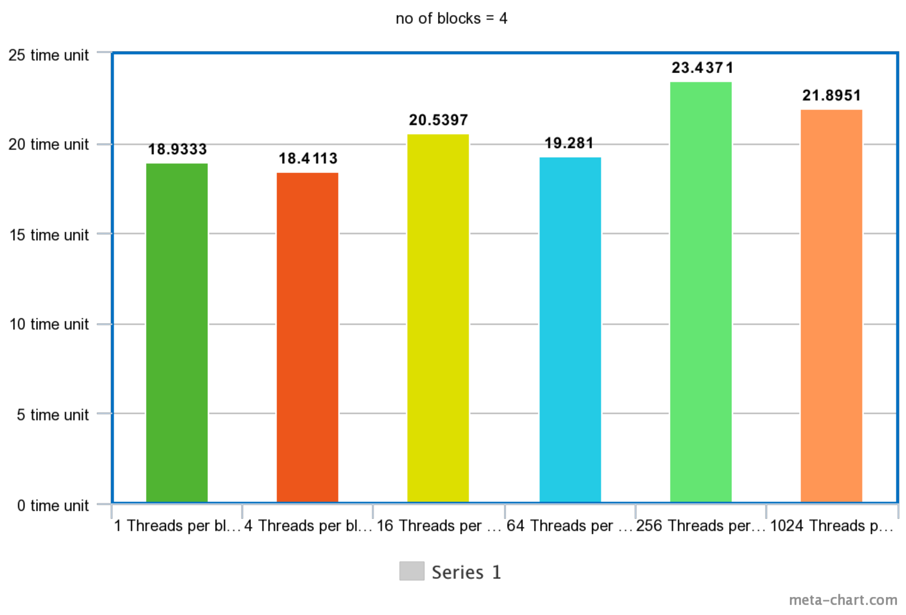


```
numBlocksW, H  : 1 1
 block width , height  : 2 2
0.125523
0.163458
0.174254
0.176717
0.177193
0.177224
0.177162
0.177086
0.177014
0.176947

 The mean error is 0.154780


Processing time: 24.965759 (ms)
```

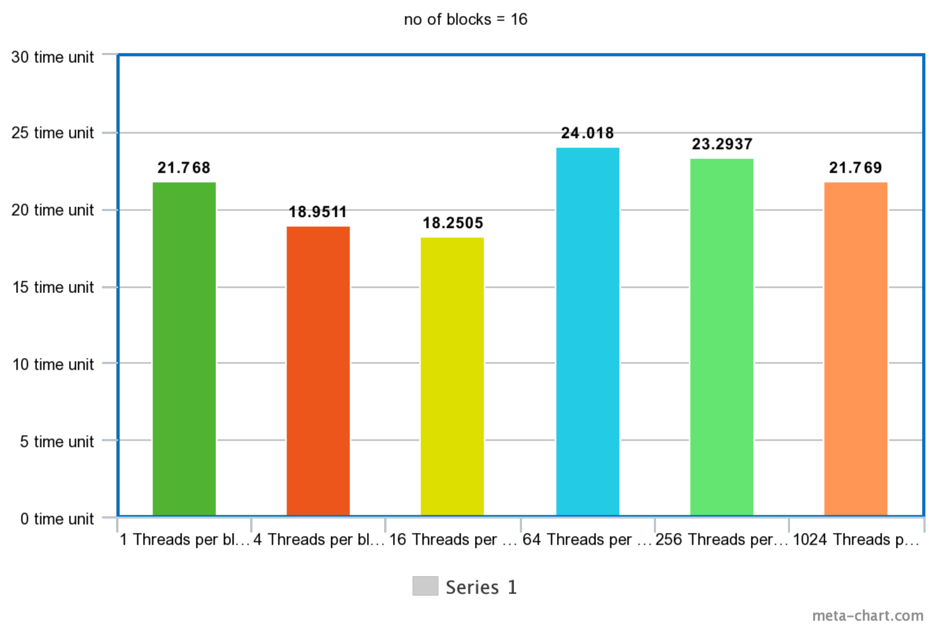
Following are the results obtained after trying various combinations of Number of Blocks and Number of threads per Block.



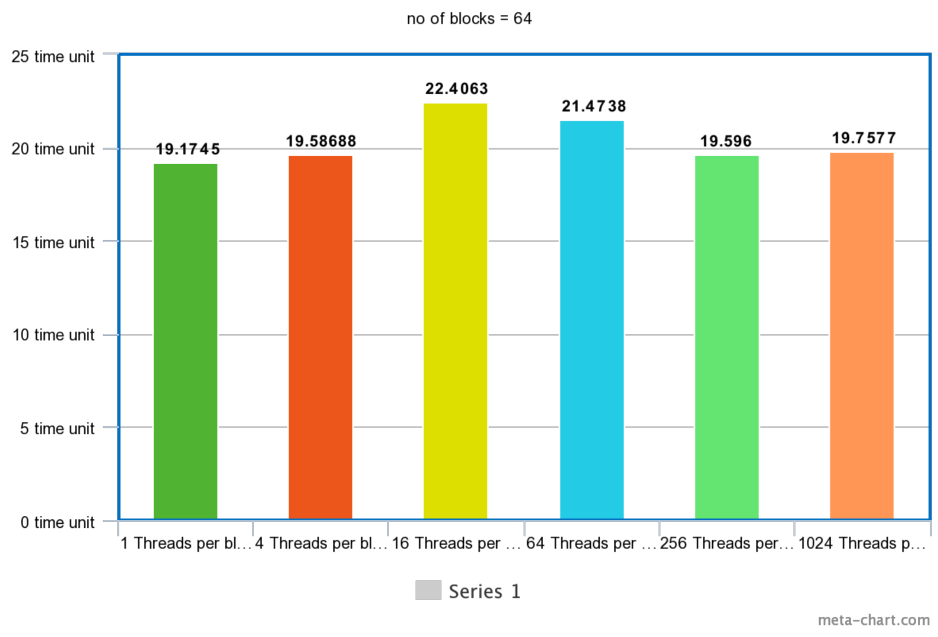
Results when Number of Blocks =1



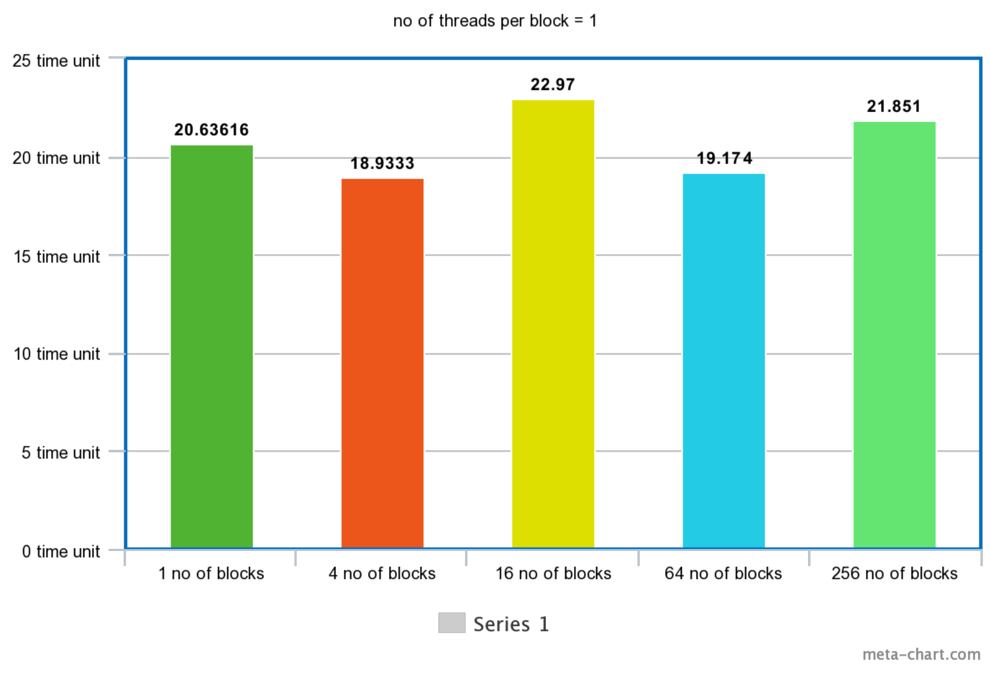
Results when Number of Blocks =4



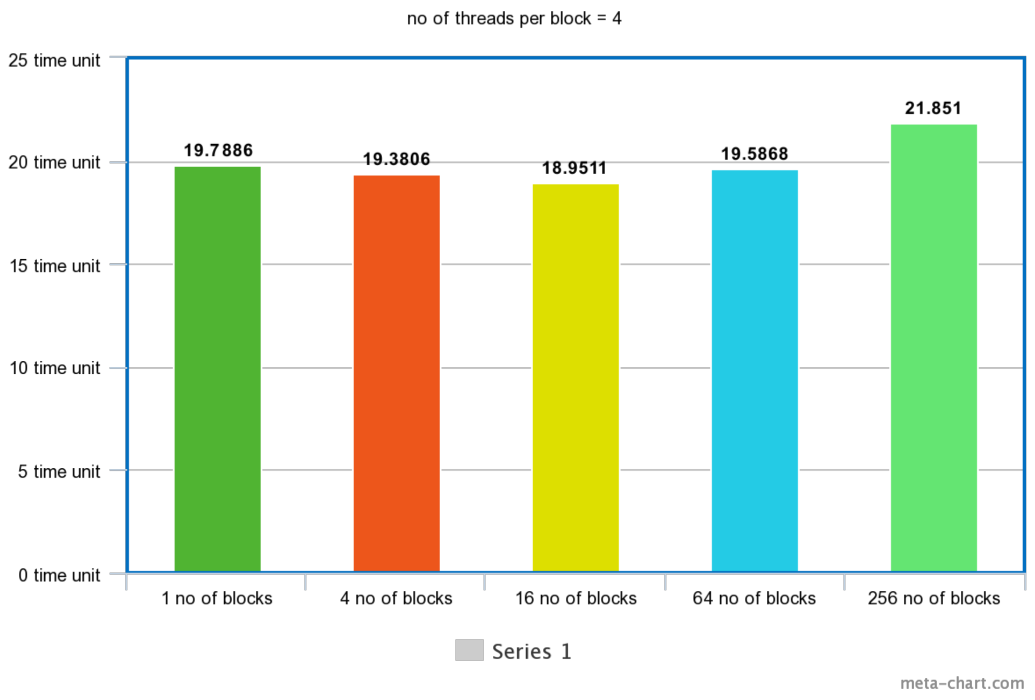
Results when Number of Blocks =16



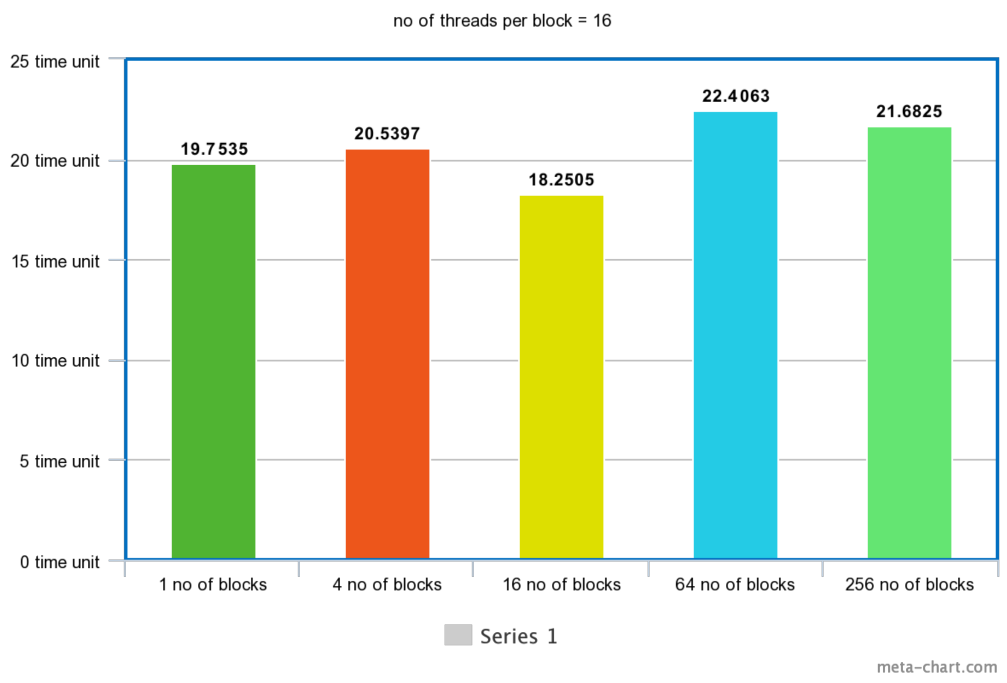
Results when Number of Blocks =64



Results when Number of Threads/Blocks =1



Results when Number of Threads/Blocks = 4



Results when Number of Threads/Blocks =16