Abstract:

Logistic regression is a statistical analysis tool used as a predictive analytic in a variety of disciplines. In this paper, we focus on parallelizing binary logistic regression analysis for predictive analytics in data science for environmental science datasets. Parallelizing logistic regression would improve computation time and decrease memory latency when analyzing large data sets, allowing for more data to be processed faster. In this paper, we compare a sequential implementation of binary logistic regression with a CUDA implementation, a parallelized R implementation and a multiprocessor OpenCV version, looking at the relationship between dataset sizes and time taken to process the data. Our findings point to improvements in computation time and less memory latency for CUDA versions of logistic regression.

Introduction:

Logistic Regression is used as a predictive analytic in many disciplines ranging from biology and conservation to business. It models a binary dependent and one or more binary or nonbinary independent variables. This is useful in cases of observing phenomena that may occur due to a specific event. The purpose of logistic regression is to predict the occurrence of phenomena based on acquired current data. Mathematically, the binary dependent variable is either 0 or 1 and indicates the presence or absence of a certain condition, such as alive/dead or win/lose, that may be related to the independent conditions. In this paper, we are primarily concerned with the applications of binary logistic regression analysis for predictive analytics in data science, particularly for environmental science datasets. For our purpose, logistic regression is used to classify dependent variables into different groups.

Currently, there is an abundance of large datasets that are open sourced and easily accessible to the public. This is especially beneficial for scientific research. However, processing large data sets is time consuming and resource intensive for CPU in terms of memory and computation time. Sequential implementations of logistic regression require a lot of time to process smaller amounts of data and have a high memory latency. Implementation of a parallelized binary logistic regression would allow for an increased amount of data to be processed in less time and with less resource intensive computations. Logistic regression is an excellent analytic to parallelize because it primarily utilizes matrix multiplication, which is easy to convert to parallel code. It also utilizes an inverse function which is a variation of matrix multiplication and determines the natural log of a matrix, a function that is easily supported by parallelism. In this paper, we compare a sequential implementation of binary logistic regression with a CUDA implementation, a parallelized R implementation and a multiprocessor OpenML version, looking at the relationship between dataset sizes and time taken to process the data. Our findings point to improvements in computation time and less memory latency for CUDA versions of logistic regression, as well as parallelized R implementations.

Background:

Logistic regression is a predictive analytic that is used to categorize data into different groups. It can be binomial, ordinal or multinomial and is based on linear

regression; a logistic regression estimates a multiple linear regression function. It is the correct regression to use if there is a binary dependent variable, such as time spent studying versus whether a student passes or fails a course, or other situations depending on independent variables such as win/lose, dead/alive, yes/no, present/absent etc. Binary logistic regression is used to describe the relationship between a binary dependent variable and one or many independent variables. This predictive analytic tool is useful in data science and many other fields for predicting how likely an event will occur when certain independent factors are present. Following are the steps needed to implement a logistic regression.

Linear Regression

In order to do a logistic regression, we must first start with a linear regression.

$$Y = B0 + B1X1 + B2X2 \ldots + BnXn$$

In this linear model, the x's are the predictors/independent variables and the Bn's are the parameters of the model or the coefficients of the independent variables, with B0 being a constant term. The Y value is the outcome /dependent variable and can vary from negative to positive infinity for a linear regression.

Logistic Function

The next step is to turn the linear regression into a sigmoid function, also known as a logistic function.

$$p(x) = E(Y) = 1/1+e^{-x} = e^{x}/1+e^{x}$$
$$= e^{(B0 + B1X1 + B2X2)} / 1+e^{(B0 + B1X1 + B2X2)}$$

P(x) is the probability of the dependent variable equaling a success. E(Y) is the expected value of the binary dependent variable Y. X is the linear regression formula. The function is as such because the range of p(x) should be between 0 to 1, rather than -infinity to +infinity. This function provides a limited range of values for probability, so the odds of the probability must be taken next to fully convert this logistic function into a logistic regression.

Logistic Regression

The logistic regression is the natural log of the inverse of the logistic function. The core of logistic regression is estimating the log odds of an event, also known as the logit of the probability, where the log odds is a prediction of the odds of a dependent variable based on one or more binary or real valued independent predictors/x-values.

The odds are determined by the inverse of the logistic function, where the probability of an outcome's success (Y) is divided by the probability that it will not occur (1-Y). To determine the log-odds of an event occurring, we must take the natural log of this inverse. The purpose of this is to fit the logit of the probability of success with the predictors/x-values. We are then able to obtain a continuous predictor for the odds of an event happening.

$$\text{logit } p(x) = \ln[p(x) / 1\text{-}p(x)]$$

The dependent variable turns into a logit variable (the natural log of the odds of the dependent variable occurring or not). We then estimate the probability of the occurrence of a certain event based on the independent variables. The logit serves as a link between the linear regression and logistic function. The probability of success of obtaining a particular value from a binary dependent is equivalent to the odds of the dependent variable Y equaling a particular case.

$$\ln[Y/(1-Y)] = B0 + B1X1 + B2X2 \ldots + BnXn$$

Logistic regression seeks to find the equation that best predicts the value of Y for each value of X. The Y variable in logistic regression is measured indirectly, where Y is the probability of obtaining a particular value of a binary variable, whereas the Y variable in linear regression is measured directly. B1 to Bn are the regression coefficients associated with the X values, the X values are the prediction variables and B0 is the reference group, which are the individuals presenting the reference level of all the X values **(cite Understanding logistic regression analysis)**.

(insert 3 graphs showing differences between logistic func/reg, linear reg)
The graph for a linear regression: note how it can extend infinitely in both directions. The graph for a logistic function: note how it extends across the y-axis. The graph for a logistic regression: note how the values are between 0 and 1 and are measured continuously.

Literature Survey:

We've looked at a variety of papers regarding parallelizing logistic regression, as well as the applications of parallelized and sequential logistic regressions. The following related works are presented because they either implement a biostatistic(?) model of logistic regression or they compare sequential and parallel versions of logistic regression.

Section A: Biostatistic logistic regression

The algorithm that we have coded sequential and parallel versions of is applicable to biology and environmental studies. The work done by (Deforestation modelling using logistic regression and GIS) uses logistic regression in the manner that we implement and parallelize. In their work, they(NAMES!) look at binary logistic regression, determining whether deforestation is present or not in an area based on slope as an independent variable. The setup of the experiment is similar to ours, and their work is what drove us to parallelize logistic regression for scientific fields. While they analyze a sequential version of the logistic regression, there is no attempt to parallelize it. The data input is also far different than ours, where the focus is to use GIS with logistic regression, making comparison with their work difficult. They(NAMES!) analyze digital thematic-topographic maps with 63 different data layers such as forests, ranges and gardens. Our tests read data from a file containing random input values that

fall within a certain range to represent our x and b values, rather than topographic maps or open source data.

Section B: Parallelizing logistic regression

Others, such as (NAMES liang, choi)(Evaluating Parallel Logistic Regression Models citation!), have parallelized the logistic regression for machine learning. Liang et. all(?) have looked at parallelized regressions in distributed platforms, parallel algorithms and sub-linear approximations. They worked with Hadoop and Spark, two distributed systems used in data science and analytics, to run sequential and parallel versions of code and compare platform/algorithm combinations with sub-linear algorithms specific to machine learning. This is clearly useful for machine learning but is not applicable to other fields that utilize logistic regression. Liang et all focus on parallelizing machine learning algorithms, while we are looking to parallelize logistic regression itself. They used five open datasets related to machine learning to make sure that they could accurately compare their methods whereas our data is randomly generated and doesn't come from open source. Using open datasets are beneficial because the results allow for comparability with the performance of sequential machine learning algorithms that already exist.

The work done by (NAMES) in (Breast Cancer Prediction by Logistic Regression with CUDA Parallel Programming Support)(CITE) focuses on parallelizing logistic regression for the purposes of machine learning, bioinformatics and data analysis. While it is primarily a machine learning approach and our work aims to be more generalized, the parallelization is reached by using CUDA parallel programming support, which we use in our experimental design as well. However, not much detail is gone into about how CUDA is implemented, only that different CUDA software versions are compared in determining the loss function for the machine learning logistic regression. Nevertheless, this work is useful in that it this approach is applicable to bioinformatics and data analysis. The dataset used is a real dataset containing anonymized patient information obtained from the Wisconsin Diagnostic Breast Cancer database. It contains 569 instances, with each instance corresponding to a patient (CITE!!!). This is beneficial in that the results obtained can be tested in real life, and they are. From the estimated Y values obtained per patient, there was a better chance at predicting whether a patient had breast cancer or not.

Section C: Implementations that we've used or referenced

In Evaluating Parallel Logistic Regression Models (CITE), Liang and Choi implement a binary logistic regression, and we reference the parallel algorithm approach, but without the distributed platform or sub-linear approximation. They analyze their experimental results in accuracy, efficiency, scalability, robustness and running time. We analyze our results in terms of accuracy and running time and use this work as a reference on how to proceed with such an analysis.

From (NAME et. all's) work in Deforestation modelling using logistic regression and GIS, we use a similar logistic regression and proceed with the sequential code and then the parallelized code in a step-by-step approach as well. We break down our

implementation into linear regression, then logistic function and finally into logistic regression in order to code logistic regression. (NAMES) use R^2 chi square test(?) to determine the fitness of their model, which is something we consider in our work but did not have enough time to do.

Proposed Solution:

We aim to parallelize the logistic regression as applied to biology and data science by using CUDA. There has been a lot of research on parallelizing logistic regression for machine learning techniques and algorithms, but nothing that we've seen that is general enough to cover parallelized logistic regression outside of machine learning purposes. The biostatistic research that we have come across uses sequential logistic regression with other new technologies but does not attempt to parallelize the logistic regression itself. From our research, we haven't found others who attempt to parallelize a general logistic regression using CUDA. Therefore, we propose to parallelize the logistic regression in order to improve predictive behavior in a number of fields. The purpose of this is to show that logistic regression can be parallelized and used to analyze more data in less time.

Experimental Setup:

ASSUMPTIONS

For our experiment we are implementing a binary logistic regression where our dependent variable is either 0 or 1 while our independent x variables are real numbers ranging from -3 to 3. To make sure our generated data set is valid, we followed the assumptions of data used for logistic regression applications. This involves having a binary dependent variable, not having outliers present in the data or strong correlations between independent data sets, and making sure no x values are below -3.29 or above 3.29. (https://www.statisticssolutions.com/what-is-logistic-regression/). Another consideration we took into account is the model fit. Having more independent x variables increases the amount of variance (R^2) but adding too many x variables will decrease the generalizability of the predictive analytic, thus decreasing its accuracy. During our analysis we made sure not to have a very large number of columns, which represents our x variables, so that we could model prime data sets used for logistic regression. **To analyze the accuracy of our logistic regression, we computed its goodness-of-fit based on the Chi square test, as well as the amount of variance R^2**. (We have not done this…may not have time to do this so remove line if not)

METHODS

We decided to use a matrix of random values for our input data, where the user chooses the number of rows and columns. The rows represent the number of samples that data is acquired for while the columns within each row represent the number of independent x variables. For example, rows could represent a patient and the columns could represent independent factors that may determine whether the patient has a certain illness or not. We chose this presentation of data so that we could calculate matrix multiplication and inversion for sequential and CUDA code.

To code our sequential version of a logistic regression, we broke down the logistic regression as described in the background section. An array of all the x values are read in from the file and transposed into a new array. X' and X are then multiplied together (X'X). We then calculate the inverse of the transposed X which is (-1*X') and then we perform matrix multiplication on (X'X) and (-1*X') . The value obtained is then multiplied with Y values to produce B values. From these B values that are produced, we populate an array of new X values and combine them with the B values to give us a predicted Y value.

For our CUDA code, we used the same logic as the sequential code but we parallelized matrix multiplication, inverse **…etc…**

MACHINE SETUP
We used NYU linux CIMS accounts to run our sequential and parallelized logistic regression code. Before running the sequential and parallelized code, we made sure to generate the random data files that we needed to run our code with. To do this we compiled the random data program and then ran it with our specified number of rows and columns. From here, to run the parallelized code we used the cuda5 machine on the NYU prince cluster and loaded the cuda version 9.1 module. For the sequential code, we used the snappy machine on NYU's prince cluster to obtain faster results for sequential code. We used the time flag to record time taken to compute the logistic regression of our sequential code and CUDA code, and we used nvprof to profile our CUDA code.

Results and Discussion:

Conclusion:

Hi Folks,
The deadline for the project is one week after the last lecture. This means you need to submit the project, through NYU classes, by Dec 19th 11:55pm.
You need to submit two things: (1) The source code with a small readme.txt file to tell us how to compile/execute your code on CIMS machines. (2) The report.
The report needs to be format as a research paper, with the following sections:

Abstract: Summarizing the problem, the importance of the problem, and the findings (1-2 paragraphs).

1. Introduction: More detailed description of the problem and its importance.

2. Background info: Any information you need to include here for us to understand your solution, especially if you are working on something that requires domain knowledge.

3. Literature Survey: What have the others do to solve this problem? What are the pros and cons of this previous work? Are there any interesting papers, not necessarily related to the problem you are trying to solve, but whose technique you will use in your project? If yes, discuss them.

4. Propose Solution: describe how you are solving the problem.

5. Experimental setup: Which machines have you used? What are the problem size(s)?etc.. Simply speaking, if I read this section, I must be able to repeat your experiments without asking you about any other details.

6. Results and Discussion: Show the results you have obtained and analyse them. You can use nvprof or any other tools to get deeper understanding of the results and come up with meaningful conclusion. If all you say is: as we can see x is increasing with y, this is considered a very bad analysis. But, why x is increasing with y? What can we learn from this? Under which conditions x will increase with y, etc.

7. Conclusions: Finally summarize your findings in a list of bullet points.

8. List of references you have used.

ex: \bibitem{notes} John W. Dower {\em Readings compiled for History 21.479.}
to cite, use ….. This is obvious \cite{notes}.

```
This is obvious [2].
```

No restrictions on the length of each section.

1. Peretti A, Amenta F (2016) Breast Cancer Prediction by Logistic Regression with CUDA Parallel Programming Support. Breast Can Curr Res 1: 111.

2. Deforestation modelling using logistic regression and GIS M. Pir Bavaghar

3. Evaluating Parallel Logistic Regression Models

4. Understanding logistic regression analysis —- Sandro Sperandei

5. Use of spatial regression models in the analysis of burnings and deforestation occurrences in forest region, Amazon, Brazil

6. Parallel Large Scale Feature Selection for Logistic Regression ———Sameer Singh,

7. Nonlinearities and heterogeneity in environmental quality: An empirical analysis of deforestation

Phu Nguyen Van

RESULTS PAGE

test time for seq up to matrix of …
test time and nvprov for cuda up to matrix of …

SEQ

row	col
10,000	10
10,000	100
10,000	1,000
10,000	10,000
10,000	100,000
10,000	1,000,000

```
time ./seqreg 100 10
final (X'X)^-1*X'Y ->
-0.144672
0.148559
-0.138328
-0.0806667
0.0609811
-0.137439
0.00861426
0.0418988
-0.0988965
-0.0257875

real 0m7.361s
user 0m7.349s
sys  0m0.004s


vaa238@linax1[CUDAregress]$ time ./seqreg 20 10
final (X'X)^-1*X'Y ->
0.0303712
0.43332
0.356843
```

```
0.136246
0.628563
-0.184302
-0.45731
0.45337
0.353311
0.149812

real 0m7.239s
user 0m7.208s
sys  0m0.021s


[vaa238@crunchy5 CUDAregress]$ time ./seqreg 100 100
exit
exit
exit
^C
real    51m8.926s
user    51m8.720s
sys 0m0.008s
```

CUDA

row	col	
10,000	10	
10,000	100	
10,000	1,000	ERROR
10,000	10,000	ERROR
10,000	100,000	row must be > col
10,000	1,000,000	row must be > col

New CUDA results

row	col	real time
10	10	0m0.803s
100	10	0m1.098s
1000	10	0m0.930s
10000	10	0m1.015s
100000	10	0m1.461s
1000000	10	ERROR

```
[vaa238@cuda5 CUDAregress]$ time ./cudareg 10000 10

final (X'X)^-1*X'Y ->
0.035933
0.00628397
0.000636919
```

```
0.0013617
-0.000546976
-0.00171491
0.011161
-0.0111071
0.0011033
0.00338498

real 0m0.355s
user 0m0.136s
sys  0m0.191s



[vaa238@cuda5 CUDAregress]$ time ./cudareg 10000 100

final (X'X)^-1*X'Y ->
-0.00238813
0.00947119
0.00805639
0.00262715
-0.0138212
0.0182296
0.0136489
-0.0120947
-0.0121761
0.00185726
0.00131094
0.0169589
0.00037683
0.0106875
-0.00534291
0.00349783
0.013537
-0.000767846
-0.0178026
-0.00944503
-0.0121835
-0.0272253
-0.0061366
-0.00680413
0.00931518
-0.0187321
-0.00137014
0.0119504
-0.00766289
-0.00323239
0.0117389
0.00279467
0.00558144
0.00683342
-0.0011481
-0.00647602
-0.0044578
0.00470331
-0.0123707
0.00702254
```

```
-0.00105229
-0.0177669
0.00422775
0.00767015
0.0186549
0.0112003
0.00230715
-0.0154416
-0.00762015
0.0219569
-0.00157796
0.0055059
-0.00255624
0.0119382
0.0238112
0.0138896
-0.00953987
-0.0211317
-0.00609091
-0.0153643
0.0114533
0.0113026
0.0152257
-0.0193705
0.00152938
0.000981795
-0.00712736
-0.00942445
0.00528686
0.0272913
-0.00929488
-0.0120774
0.00497741
-0.0115304
0.0175597
0.0088993
0.0076019
-0.0054308
-0.0155139
0.0156907
0.0120422
3.25533e-05
0.00270377
0.00286546
-0.0182881
-0.0164623
-0.000158567
0.00506452
0.00471663
0.0124728
0.00806342
0.0145796
-0.0203197
-0.00303626
0.00679254
-0.00426496
-0.000834481
-0.0169363
```

```
-0.00701451
0.0172833

real 0m2.393s
user 0m0.434s
sys  0m1.513s


[vaa238@cuda5 CUDAregress]$ time ./cudareg 10000 1000

Segmentation fault (core dumped)

real 0m8.740s
user 0m4.707s
sys  0m2.296s


[vaa238@cuda5 CUDAregress]$ time ./cudareg 10000 10000
Segmentation fault (core dumped)

real 3m21.193s
user 2m4.207s
sys  0m57.416s
```

```
[vaa238@cuda5 CUDAregress]$ time ./cudareg 100000 10

final (X'X)^-1*X'Y ->
0.00318976
0.000914195
0.00199461
-0.000102013
0.00220997
-0.00162936
0.000681088
-0.00232166
0.00011445
-0.00118732

real 0m1.461s
user 0m0.596s
sys  0m0.748s

==12222== Profiling application: ./cudareg 100000 10
==12222== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.27%  242.82ms         3  80.941ms  67.577ms  93.520ms
cudaMatMultiplication(float*, int, int, float*, int, int, float*)
                   1.80%  4.4900ms         7  641.43us  1.4070us  1.5493ms  [CUDA memcpy HtoD]
                   0.91%  2.2621ms         4  565.52us  2.0160us  2.2560ms  [CUDA memcpy DtoH]
                   0.02%  40.736us        10  4.0730us  3.7120us  6.3360us
computeRowsKernel(float*, int, int)
                   0.01%  28.703us        10  2.8700us  2.6870us  4.2880us
computeColsKernel(float*, int, int)
                   0.00%  2.3360us         1  2.3360us  2.3360us  2.3360us
augmentMatrixKernel(float*, float*, int, int)
                   0.00%  2.1440us         1  2.1440us  2.1440us  2.1440us
getInverseMatrixKernel(float*, float*, int, int)
      API calls:   46.68%  267.32ms        12  22.277ms  6.8200us  264.47ms  cudaMalloc
                   45.46%  260.31ms        11  23.665ms  23.412us  95.292ms  cudaMemcpy
```

```
                    6.52%  37.313ms        21  1.7768ms  1.1191ms  2.5089ms
cudaDeviceSynchronize
                    0.56%  3.2315ms       376  8.5940us     153ns  412.74us  cuDeviceGetAttribute
                    0.55%  3.1461ms        12  262.17us  6.2060us  464.49us  cudaFree
                    0.12%  694.43us        25  27.777us  12.318us  309.23us  cudaLaunch
                    0.06%  335.27us         4  83.816us  58.841us  130.79us  cuDeviceTotalMem
                    0.05%  258.17us         4  64.543us  60.619us  66.996us  cuDeviceGetName
                    0.00%  24.188us        89     271ns     140ns  5.2990us  cudaSetupArgument
                    0.00%  12.159us        25     486ns     336ns  1.9830us  cudaConfigureCall
                    0.00%  5.4390us         8     679ns     235ns  2.8270us  cuDeviceGet
                    0.00%  2.1320us         3     710ns     240ns  1.3290us  cuDeviceGetCount
```

```
[vaa238@cuda5 CUDAregress]$ time ./cudareg 10000 10

final (X'X)^-1*X'Y ->
0.035933
0.00628397
0.000636919
0.0013617
-0.000546976
-0.00171491
0.011161
-0.0111071
0.0011033
0.00338498

real  0m1.015s
user  0m0.167s
sys   0m0.693s

==12196== Profiling application: ./cudareg 10000 10
==12196== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.85%  191.25ms         3  63.751ms  52.150ms  81.304ms
cudaMatMultiplication(float*, int, int, float*, int, int, float*)
                    0.09%  172.00us         7  24.571us  1.4400us  40.928us  [CUDA memcpy HtoD]
                    0.02%  42.879us         4  10.719us  1.9200us  36.671us  [CUDA memcpy DtoH]
                    0.02%  42.752us        10  4.2750us  3.9040us  6.2400us
computeRowsKernel(float*, int, int)
                    0.01%  28.000us        10  2.8000us  2.7520us  2.8480us
computeColsKernel(float*, int, int)
                    0.00%  2.4320us         1  2.4320us  2.4320us  2.4320us
augmentMatrixKernel(float*, float*, int, int)
                    0.00%  2.2080us         1  2.2080us  2.2080us  2.2080us
getInverseMatrixKernel(float*, float*, int, int)
      API calls:   53.87%  283.36ms        12  23.613ms  5.8340us  280.84ms  cudaMalloc
                   38.35%  201.74ms        11  18.340ms  16.315us  83.494ms  cudaMemcpy
                    6.65%  34.958ms        21  1.6647ms  199.11us  2.5222ms
cudaDeviceSynchronize
                    0.57%  2.9753ms       376  7.9130us     150ns  322.63us  cuDeviceGetAttribute
                    0.34%  1.7660ms        12  147.17us  7.6080us  257.08us  cudaFree
                    0.11%  582.74us        25  23.309us  11.241us  237.06us  cudaLaunch
                    0.06%  327.70us         4  81.926us  60.360us  125.34us  cuDeviceTotalMem
                    0.05%  260.95us         4  65.237us  60.033us  78.618us  cuDeviceGetName
                    0.00%  22.777us        89     255ns     125ns  5.7300us  cudaSetupArgument
                    0.00%  12.850us        25     514ns     334ns  2.4150us  cudaConfigureCall
                    0.00%  4.8840us         8     610ns     208ns  2.4960us  cuDeviceGet
                    0.00%  2.3890us         3     796ns     191ns  1.6800us  cuDeviceGetCount
```

```
[vaa238@cuda5 CUDAregress]$ time ./cudareg 1000 10

final (X'X)^-1*X'Y ->
0.0130835
0.0604227
0.0368568
```

```
-0.0130747
0.0250712
-0.0673258
-0.0126996
-0.0319951
0.025234
-0.00263982

real  0m0.930s
user  0m0.096s
sys   0m0.637s

==12122== Profiling application: ./cudareg 1000 10
==12122== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.92%  142.27ms         3  47.424ms  22.296ms  66.144ms
cudaMatMultiplication(float*, int, int, float*, int, int, float*)
                   0.02%  33.247us        10  3.3240us  2.0790us  13.856us
computeColsKernel(float*, int, int)
                   0.02%  32.544us         7  4.6490us  1.7280us  6.9120us  [CUDA memcpy HtoD]
                   0.02%  26.591us        10  2.6590us  2.4630us  3.8400us
computeRowsKernel(float*, int, int)
                   0.01%  12.319us         4  3.0790us  2.0480us  5.6950us  [CUDA memcpy DtoH]
                   0.00%  2.3680us         1  2.3680us  2.3680us  2.3680us
augmentMatrixKernel(float*, float*, int, int)
                   0.00%  1.8880us         1  1.8880us  1.8880us  1.8880us
getInverseMatrixKernel(float*, float*, int, int)
      API calls:   60.41%  231.48ms        12  19.290ms  8.3910us  228.91ms  cudaMalloc
                   38.01%  145.63ms        11  13.239ms  14.084us  67.609ms  cudaMemcpy
                    0.80%  3.0470ms       376  8.1030us     154ns  316.77us  cuDeviceGetAttribute
                    0.46%  1.7496ms        12  145.80us  6.2990us  275.02us  cudaFree
                    0.13%  491.56us        25  19.662us  10.266us  171.98us  cudaLaunch
                    0.08%  319.64us         4  79.911us  59.014us  131.23us  cuDeviceTotalMem
                    0.07%  260.92us         4  65.228us  60.671us  71.650us  cuDeviceGetName
                    0.04%  165.15us        21  7.8640us  5.2780us  18.267us
cudaDeviceSynchronize
                    0.01%  21.980us        89     246ns     128ns  5.2660us  cudaSetupArgument
                    0.00%  10.898us        25     435ns     268ns  2.4500us  cudaConfigureCall
                    0.00%  5.3390us         8     667ns     214ns  2.7840us  cuDeviceGet
                    0.00%  2.1140us         3     704ns     190ns  1.3770us  cuDeviceGetCount
```

```
[vaa238@cuda5 CUDAregress]$ time ./cudareg 100 10

final (X'X)^-1*X'Y ->
-0.144672
0.148559
-0.138328
-0.0806667
0.0609811
-0.137439
0.00861428
0.0418988
-0.0988965
-0.0257875

real  0m1.098s
user  0m0.116s
sys   0m0.674s

==12090== Profiling application: ./cudareg 100 10
==12090== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.95%  188.44ms         3  62.813ms  58.663ms  66.223ms
cudaMatMultiplication(float*, int, int, float*, int, int, float*)
                   0.02%  37.503us        10  3.7500us  3.6800us  4.0320us
computeRowsKernel(float*, int, int)
```

```
                    0.01%  28.224us        10  2.8220us  2.7520us  2.9440us
computeColsKernel(float*, int, int)
                    0.01%  11.328us         7  1.6180us  1.4400us  1.7600us  [CUDA memcpy HtoD]
                    0.00%  9.1830us         4  2.2950us  2.0800us  2.4640us  [CUDA memcpy DtoH]
                    0.00%  2.4000us         1  2.4000us  2.4000us  2.4000us
augmentMatrixKernel(float*, float*, int, int)
                    0.00%  2.2080us         1  2.2080us  2.2080us  2.2080us
getInverseMatrixKernel(float*, float*, int, int)
      API calls:   54.17%  282.94ms        12  23.578ms  5.5130us  280.83ms  cudaMalloc
                   38.35%  200.33ms        11  18.211ms  15.178us  68.400ms  cudaMemcpy
                    6.10%  31.837ms        21  1.5161ms  311.87us  2.3572ms
cudaDeviceSynchronize
                    0.58%  3.0408ms       376  8.0870us     150ns  329.24us  cuDeviceGetAttribute
                    0.37%  1.9314ms        25  77.254us  11.179us  1.5764ms  cudaLaunch
                    0.30%  1.5748ms        12  131.23us  7.4050us  328.85us  cudaFree
                    0.07%  347.73us         4  86.931us  63.846us  140.19us  cuDeviceTotalMem
                    0.05%  254.65us         4  63.661us  60.182us  70.061us  cuDeviceGetName
                    0.00%  21.804us        89     244ns     122ns  5.3560us  cudaSetupArgument
                    0.00%  14.552us         8  1.8190us     206ns  12.182us  cuDeviceGet
                    0.00%  11.603us        25     464ns     303ns  1.6700us  cudaConfigureCall
                    0.00%  2.0390us         3     679ns     188ns  1.3420us  cuDeviceGetCount
```

```
[vaa238@cuda5 CUDAregress]$ time ./cudareg 10 10

final (X'X)^-1*X'Y ->
-0.713823
1.59628
-0.0107803
0.776971
0.500111
1.08367
-2.57054
-0.0673254
1.98204
-0.19013

real  0m0.803s
user  0m0.034s
sys   0m0.676s


==12051== Profiling application: ./cudareg 10 10
==12051== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.95%  198.17ms         3  66.058ms  60.958ms  73.733ms
cudaMatMultiplication(float*, int, int, float*, int, int, float*)
                    0.02%  43.935us        10  4.3930us  3.6800us  9.6960us
computeRowsKernel(float*, int, int)
                    0.01%  28.288us        10  2.8280us  2.7840us  2.8480us
computeColsKernel(float*, int, int)
                    0.00%  9.3760us         7  1.3390us  1.1520us  1.4720us  [CUDA memcpy HtoD]
                    0.00%  8.8000us         4  2.2000us  2.1120us  2.2400us  [CUDA memcpy DtoH]
                    0.00%  2.3360us         1  2.3360us  2.3360us  2.3360us
augmentMatrixKernel(float*, float*, int, int)
                    0.00%  2.2080us         1  2.2080us  2.2080us  2.2080us
getInverseMatrixKernel(float*, float*, int, int)
      API calls:   56.67%  334.04ms        12  27.837ms  5.3610us  332.85ms  cudaMalloc
                   35.99%  212.17ms        11  19.288ms  14.290us  75.974ms  cudaMemcpy
                    6.03%  35.522ms        21  1.6915ms  1.1161ms  2.6247ms
cudaDeviceSynchronize
                    0.52%  3.0919ms        25  123.67us  11.577us  2.7195ms  cudaLaunch
                    0.52%  3.0507ms       376  8.1130us     150ns  326.93us  cuDeviceGetAttribute
                    0.17%  985.25us        12  82.104us  6.2370us  244.77us  cudaFree
                    0.06%  329.83us         4  82.456us  57.528us  127.57us  cuDeviceTotalMem
                    0.04%  248.74us         4  62.184us  60.076us  65.005us  cuDeviceGetName
                    0.00%  22.309us        89     250ns     124ns  5.4070us  cudaSetupArgument
                    0.00%  11.636us        25     465ns     299ns  2.0620us  cudaConfigureCall
                    0.00%  5.0710us         8     633ns     196ns  2.8470us  cuDeviceGet
                    0.00%  2.4350us         3     811ns     220ns  1.6790us  cuDeviceGetCount
```