ADVANCED BIG DATA ANLAYICS

Assignment 3 Write-up

By

Ashish Nanda
Uni: an2706

INSTALLATION:

The first requirement of the assignment was to install PyCuda/PyOpenCl either on AWS or on our own laptops. I decided to install PyOpenCl on my Macbook, which also has a dedicated GPU: The Nvidia GeForce GT 750M.

In order to complete the installation, I performed the following steps as per the instructions on the documentation page:

```
pip install numpy # Installs version 1.9
git clone http://git.tiker.net/trees/pyopencl.git
cd pyopencl
git submodule init
git submodule update
python configure.py
python setup.py build
make
python setup.py install
```

Once this was competed, the PyOpenCl installation was successful.

GPU PROGRAMMING AND LINEAR REGRESSION MATRIX OPERATIONS:

Linear regression is one of the most popular and effective prediction techniques, and was also one of the learning methods I had used in assignment 2 for prediction of stock prices. Since PyOpenCl is used largely for computing matrix transformations, we aim to compute some of the transformations involved in solving the linear regression problem.
The derivation for the ordinary least squares linear regression estimators β is given below.

Input: We observe pairs $(X_1, Y_1), \ldots (X_n, Y_n)$.

- $X_i \in \mathrm{R}^p$ is called a feature or covariate or predictor
- $Y_i \in \mathrm{R}$ is called the response.
- The set $\{(X_1, Y_1), \ldots (X_n, Y_n)\}$ is called the training data.

Task:

- Given a new observation X, predict its response Y.
- Understand which covariates are most important in explaining Y.

Example: Predicting blood pressure based on gene expression data

- Predictors $X_i = X_i^1, \ldots, X_i^p$ is a vector of gene expression levels for patient i.
- Response Y_i is his/her blood pressure.

- Seeks predictor such that for $x \in \mathrm{R}^p$

$$m(x) = \beta_0 + \beta_1 x_1 + \ldots \beta_p x_p.$$

- When p is small, a way to find a "good" predictor is by picking β that minimizes the ℓ_2-error on the training data:

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p}\beta_j X_i^j)^2 = \frac{1}{n}\|\boldsymbol{Y} - \boldsymbol{X}\beta\|^2.$$

  The minimizer β_{OLS} of the ℓ_2 error is called the **Ordinary Least Squares estimator (OLS).**

- The OLS estimator has a closed-form solution

$$\beta_{OLS} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y},$$

  where $[\boldsymbol{X}]_{i,j} = X_i^j$ and $[\boldsymbol{Y}]_i = Y_i$.

Thus we can see that to calculate β from the input matrix X and response matrix Y, we need to be able to compute the transpose, inverse and multiplication of matrices.

Since inverse would be very complicated to compute, we decide to focus on computing the other two matrix transformations- **multiplication** and **transpose** on the GPU by implementing different algorithms in PyOpenCl.

The following sections in the report describe the different algorithms and kernels used for each of the transformations, and also compare the time taken by the CPU vs the GPU to compute the operation for different sizes of randomly generated matrices.

MATRIX TRANSPOSE:

Matrix Transpose is one of the transformations required for computing linear regression coefficients, and I have implemented two different kernels to compute the same. One is a naïve implementation and the other is a row optimized implementation for computing the transpose using PyOpenCl. These are given below:

1) Naïve Transpose:

This is the simplest implementation for a Kernel that performs the Transpose computation. Here there are two threads used and one element of the Transpose matrix is computed each time. A snapshot of the kernel code and some important functions are given below:

```

###func0: Naive Implementation ###

func0= cl.Program(ctx,"""
#pragma OPENCL EXTENSION cl_khr_fp64: enable
__kernel void mat_transpose(__global float* A, __global float *A_trans, unsigned int H_A, unsigned int W_A) {
        unsigned int i = get_global_id(0);
        unsigned int j = get_global_id(1);
    A_trans[i*H_A+j]=0;
        A_trans[i*H_A + j]= A[j*W_A +i];
}
""").build().mat_transpose                      #KERNEL

""" """
func0.set_scalar_arg_dtypes([None, None, np.uint32, np.uint32])

def trans_naive(a_buf, atrans_buf, H_A, W_A):
    start = time.time()
    func0(queue, (W_A, H_A), None, a_buf, atrans_buf, np.uint32(H_A), np.uint32(W_A))
        return time.time()-start

def cl_naive_trans(a,a_trans,HA,WA):
    a_buf, atrans0_buf = mem_alloc(a, a_trans)
        t=trans_naive(a_buf,atrans0_buf, HA, WA)
        a_trans0=mem_transfer(a_trans,atrans0_buf)
    return t, a_trans0
```

2) Row Optimization Transpose:

This is a somewhat optimized kernel implementation for computing the transpose. Instead of using two threads like in the previous kernel, we are using only a single thread for a complete row operation. Thus this remains a parallelized operation, but has reduced latency. A snapshot of the code is given below:

```
###func1: Row Optimisation (All Global)###

func1= cl.Program(ctx,"""
#pragma OPENCL EXTENSION cl_khr_fp64: enable
__kernel void mat_transpose(__global float* A, __global float *A_trans, unsigned int H_A, unsigned int W_A) {
        unsigned int i = get_global_id(0);
        unsigned int j;

    for (j=0;j<H_A;j++) {
            A_trans[i*H_A + j]=0.0;
        }
    for (j=0;j<H_A;j++) {
            A_trans[i*H_A + j]= A[j*W_A +i];
    }
}
""").build().mat_transpose                                          #KERNEL
"""" """"
func1.set_scalar_arg_dtypes([None, None, np.uint32, np.uint32])

def trans_row_opt(a_buf, atrans_buf, H_A, W_A):
    start = time.time()
    func1(queue, (W_A, ), None, a_buf, atrans_buf, np.uint32(H_A), np.uint32(W_A))
    return time.time()-start

def cl_row_opt_trans(a,a_trans,HA,WA):
        a_buf, atrans1_buf = mem_alloc(a, a_trans)
        t=trans_row_opt(a_buf,atrans1_buf, HA, WA)
        a_trans1=mem_transfer(a_trans,atrans1_buf)
    return t, a_trans1
```

Output of Transpose code:

The code first checks whether the output value for the transpose of a matrix computed through our PyOpenCL algorithms are in fact correct. In order to do this we compare the output obtained from each kernel implementation with the output from the regular python-numpy computation using the CPU. Once this is 'True', we then compare the times for the two kernel implementations for different sizes of the input matrix.

It is important to note here that the matrix transpose computed by numpy using the CPU will be faster always because numpy's transpose is optimized such that it only changes the strides (successive memory locations) and doesn't touch the actual array, while our kernel implementations actually manipulate array values. Thus a speedup is not achieved unlike in the case of multiplication where the GPU computation is a lot faster.

The output is given below:

```
##############################################
Matrix Transpose

Output for Python-CPU and Naive-Kernel-GPU are equal:   True
Output for Python-CPU and RowOpt-Kernel-GPU are equal:  True

Dim      Python_time     Naive_transpose Row Optimisation
( 6 , 8 )       1.54972076416e-06       4.30345535278e-05       6.15119934082e-05
( 12 , 16 )     9.53674316406e-07       3.79681587219e-05       3.54647636414e-05
( 18 , 24 )     1.07288360596e-06       3.27229499817e-05       2.97427177429e-05
( 24 , 32 )     7.7486038208e-07        3.18288803101e-05       3.17096710205e-05
( 30 , 40 )     1.01327896118e-06       5.57899475098e-05       7.31945037842e-05
( 36 , 48 )     1.31130218506e-06       6.72936439514e-05       3.8206577301e-05
( 42 , 56 )     1.25169754028e-06       3.61800193787e-05       3.37362289429e-05
( 48 , 64 )     1.31130218506e-06       4.39882278442e-05       3.74317169189e-05
( 54 , 72 )     1.49011611938e-06       6.31809234619e-05       7.24792480469e-05
( 60 , 80 )     1.54972076416e-06       7.45058059692e-05       6.37173652649e-05
( 66 , 88 )     1.31130218506e-06       5.63263893127e-05       8.07046890259e-05
( 72 , 96 )     1.49011611938e-06       5.13195991516e-05       6.72936439514e-05
( 78 , 104 )    1.49011611938e-06       4.48226928711e-05       4.27961349487e-05
( 84 , 112 )    1.31130218506e-06       4.13060188293e-05       4.02331352234e-05
( 90 , 120 )    1.01327896118e-06       4.64916229248e-05       5.84721565247e-05
( 96 , 128 )    1.96695327759e-06       4.87565994263e-05       4.50611114502e-05
( 102 , 136 )   1.49011611938e-06       4.20212745667e-05       4.24981117249e-05
( 108 , 144 )   1.49011611938e-06       4.04715538025e-05       4.16040420532e-05
( 114 , 152 )   1.49011611938e-06       4.20212745667e-05       4.24385070801e-05
( 120 , 160 )   1.19209289551e-06       4.24981117249e-05       4.54783439636e-05
( 126 , 168 )   1.54972076416e-06       4.50015068054e-05       4.66704368591e-05
( 132 , 176 )   1.49011611938e-06       6.0498714447e-05        4.47034835815e-05
( 138 , 184 )   1.54972076416e-06       4.42266464233e-05       4.47630882263e-05
( 144 , 192 )   1.49011611938e-06       4.48226928711e-05       6.55651092529e-05
( 150 , 200 )   1.72853469849e-06       5.55515289307e-05       6.67572021484e-05
( 156 , 208 )   1.78813934326e-06       5.50150871277e-05       7.2717666626e-05
( 162 , 216 )   1.78813934326e-06       5.72800636292e-05       6.6876411438e-05
( 168 , 224 )   1.54972076416e-06       5.82337379456e-05       8.1479549408e-05
```

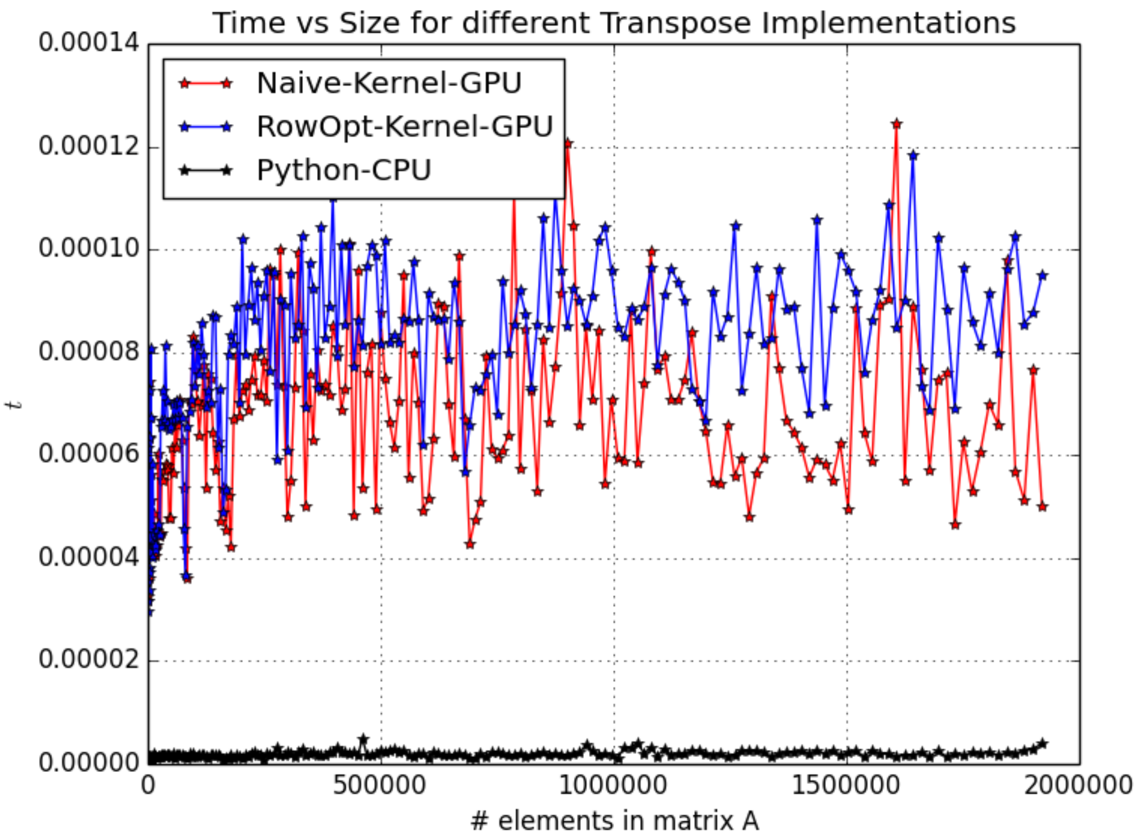
Now we also wish to visualize the performance of the two kernel implementations with respect to the dimensions of the matrix. Thus we check at which point one of the implementations become faster, and also plot the graph for Time vs Size. A snapshot of the code and output are given below:

```
### Comparing differnet pyopenCL kernel timings ###

python_times=[]
pyopencl_naive_times=[]
pyopencl_row_opt_times=[]


param=np.arange(1,201,1).astype(np.int32)

for i in param:
    python_times.append(py_time(i*H_A,i*W_A,4))
    pyopencl_naive_times.append(cl_naive_time(i*H_A,i*W_A,4))
    pyopencl_row_opt_times.append(cl_row_opt_time(i*H_A,i*W_A,4))

print "\nDim\t", "Python_time\t", "Naive_transpose\t", "Row Optimisation\t"
for i in param:
    print "(",i*H_A, ",",i*W_A,")\t", python_times[i-1],"\t", pyopencl_naive_times[i-1], "\t", pyopencl_row_opt_times[i-1], "\t"

for i in param:
    if pyopencl_row_opt_times[i-1] < pyopencl_naive_times[i-1]:
        print "\nAt a dimension size of (", i*H_A, ",", i*W_A, "), Row Optimization beats Naive Transpose implementations"
        break




plt.clf()
plt.plot(param*H_A*param*W_A, pyopencl_naive_times, 'r*-',
         param*H_A*param*W_A, pyopencl_row_opt_times, 'b*-',
         param*H_A*param*W_A, python_times, 'k*-')

plt.xlabel('# elements in matrix A')
plt.ylabel('$t$')
plt.title('Time vs Size for different Transpose Implementations')
plt.legend(('Naive-Kernel-GPU', 'RowOpt-Kernel-GPU', 'Python-CPU'), loc='upper left')
plt.grid(True)
#plt.draw()
plt.savefig('Transpose_scaling.png')

```

After all the timings are printed, we also print the first input matrix dimensions for which Row optimization beats Naïve Transpose implementation:

```
( 1152 , 1536 ) 2.26497650146e-06       5.30481338501e-05       8.60691070557e-05
( 1158 , 1544 ) 2.02655792236e-06       6.07967376709e-05       8.12411308289e-05
( 1164 , 1552 ) 2.26497650146e-06       6.99758529663e-05       9.14931297302e-05
( 1170 , 1560 ) 1.72853469849e-06       6.60419464111e-05       7.99894332886e-05
( 1176 , 1568 ) 2.20537185669e-06       9.79900360107e-05       9.62018966675e-05
( 1182 , 1576 ) 2.02655792236e-06       5.69820404053e-05       0.000102698802948
( 1188 , 1584 ) 2.74181365967e-06       5.12599945068e-05       8.55326652527e-05
( 1194 , 1592 ) 2.92062759399e-06       7.65919685364e-05       8.77380371094e-05
( 1200 , 1600 ) 3.99351119995e-06       5.01871109009e-05       9.50694084167e-05

At a dimension size of ( 12 , 16 ), Row Optimization beats Naive Transpose implementations
Ashishs-MacBook-Pro:AdvBigData AshishNanda$
```

The code also generates a plot for the Time vs Size for different Transpose Implementations:



MATRIX MULTIPLICATION:

Matrix Multiplication is also one of the main transformations required for computing linear regression coefficients, and I have implemented two different kernels to compute the same. One is a Naïve implementation and the other is a Tiled implementation for computing the matrix product using PyOpenCl. In both these implementations there is a major improvement in speed using the GPU as compared to the CPU. The two approaches are given below:

1) Naïve Multiplication:

This is the most basic implementation of a kernel for Multiplication. We use two threads, and each time compute the value of a single element in the resultant matrix by operating on the corresponding row and column of the input matrices. Below is a snapshot of the code for the Naïve Kernel and some related functions:

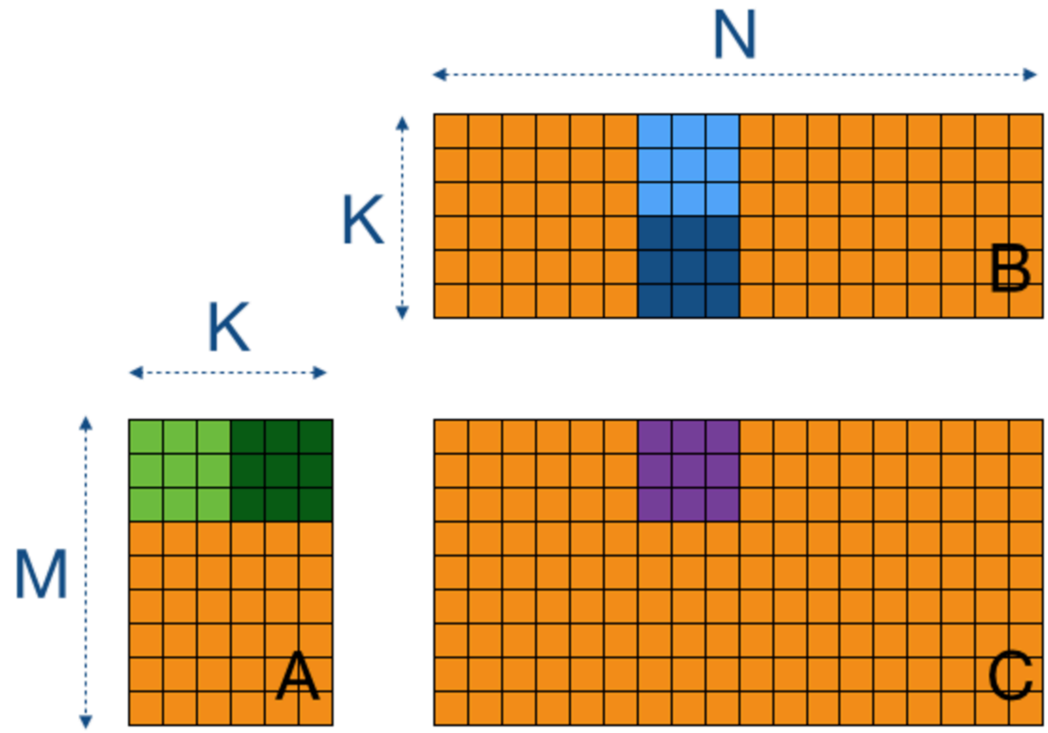
```
#### Defining the Naive Kernel ####

func_mult_naive= cl.Program(ctx,"""
#pragma OPENCL EXTENSION cl_khr_fp64: enable
__kernel void mat_mult(__global float* A, __global float* B, __global float* D, unsigned int SIZE) {
        unsigned int i = get_global_id(0);
        unsigned int j = get_global_id(1);
    unsigned int k;
    float temp=0.0;
    for (k=0; k<SIZE; k++) {

        temp += (A[i*SIZE + k] * B[k*SIZE + j]);
    }
    D[i*SIZE + j] = temp;
}
""").build().mat_mult                       #KERNEL
func_mult_naive.set_scalar_arg_dtypes([None, None, None, np.uint32])

def mult_op_naive(a_buf, b_buf, d_buf, siz):
    start = time.time()
    func_mult_naive(queue, (siz,siz), None, a_buf, b_buf, d_buf, np.uint32(siz))
    return time.time()-start


def cl_op_mult_naive(a,b,d,siz):
        a_buf,b_buf,d_buf = mem_alloc(a,b,d)
        t=mult_op_naive(a_buf,b_buf,d_buf, siz)
        d=mem_transfer(d,d_buf)
        return t, d
```

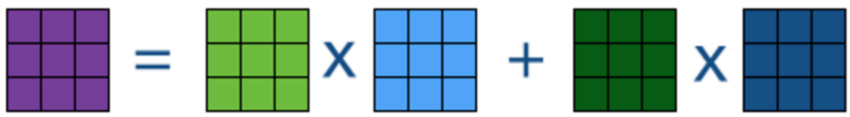
2) Tiled Multiplication:

One of the popular optimizations in matrix operations using kernels is to use Tiling. In order to achieve the optimization, typically a naïve implementation of an algorithm that refers to individual elements is replaced by one that operates on subarrays of data, which are called blocks (or tiles) in the matrix computing field. The operations on subarrays can be expressed in the usual way. The advantage of this approach is that the small blocks can be moved into the fast local memory and their elements can then be repeatedly used.

The following diagram and explanation can help illustrate this more clearly:

To compute a sub-block Csub of C (purple tile in the image below), we need A's corresponding rows (in green) and B's corresponding columns (in blue). Now, if we also divide A and B in sub-blocks Asub and Bsub, we can iteratively update the values in Csub by summing up the results of multiplications of Asub times Bsub.





This is useful because if we take a closer look at the computation of a single element (in the image below), we see that there is lots of data re-use within a tile. For example, in the 3x3 tiles of the image below, all elements on the same row of the purple tile (Csub) are computed using the same data of the green tiles (Asub).

Based on the approach above, I have written a kernel in PyOpenCl that computes matrix multiplication using tiling. A snapshot of the code for the kernel and some related functions can be seen below:

```
#### Defining the Tiled Kernel ####

func_mult_tiling= cl.Program(ctx,"""
#pragma OPENCL EXTENSION cl_khr_fp64: enable
__kernel void mat_mult(__global float* A, __global float* B, __global float* D, unsigned int SIZE, unsigned int m, unsigned int n, unsigned int p) {
    __local float AS[1024];
    __local float BCS[1024];
    int i = get_global_id(1);
    int j = get_global_id(0);

    int bx = get_group_id(0);
    int by = get_group_id(1);
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    int aBegin = n* SIZE * by;
    int aEnd   = aBegin + n - 1;
    int aStep  = SIZE;
    int bBegin = SIZE * bx;
    int bStep  = SIZE * p;
    float temp = 0.0f;
    for (int a = aBegin, b = bBegin; a <= aEnd;a += aStep, b += bStep)
    {
        AS[tx + ty*SIZE] = A[a + n * ty + tx];
        BCS[tx + ty*SIZE] = B[b + p*ty + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k = 0; k < SIZE; ++k)
            temp += AS[ty*SIZE + k] * BCS[k*SIZE + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    D[i * p + j] = temp;
}
""").build().mat_mult                                               #KERNEL
func_mult_tiling.set_scalar_arg_dtypes([None, None, None, np.uint32, np.uint32, np.uint32, np.uint32])

def mult_op_tiling(a_buf, b_buf, d_buf, siz, m, n, p):
    start = time.time()
    func_mult_tiling(queue, (m,p), (siz,siz), a_buf, b_buf, d_buf, np.uint32(siz), np.uint32(m), np.uint32(n), np.uint32(p))
    return time.time()-start


def cl_op_mult_tiling(a,b,d,siz,m,n,p):
        a_buf,b_buf,d_buf = mem_alloc(a,b,d)
        t=mult_op_tiling(a_buf,b_buf,d_buf, siz, m,n,p)
        d=mem_transfer(d,d_buf)
        return t, d
```

Output of Multiplication code:

The code first checks whether the output value for the product of two input matrices computed through our PyOpenCL algorithms are in fact correct. In order to do this we compare the output obtained from each kernel implementation with the output from the regular python-numpy computation using the CPU. Once this is 'True', we then compare the times for the two kernel implementations for different sizes of the input matrix.

In the case of Matrix Multiplication, the GPU based computations are a lot faster as compared to the regular python-numpy computation using the CPU. The speedup is around 150-200 times for inputs involving square matrices of around a size of 5Mb. Thus we print the speedup factor, compare the times and even make a plot of the Time vs Size to get a better understanding of the improvement

The output is given below:

```
Output for Python-CPU and Naive-Kernel-GPU are equal:   True
Output for Python-CPU and Tiling-Kernel-GPU are  equal: True

Dim              Python_time            Naive_time       Tiling_time
( 32 , 32 )      2.37226486206e-05      5.07235527039e-05      6.17504119873e-05
( 64 , 64 )      2.72989273071e-05      3.99947166443e-05      4.4047832489e-05
( 96 , 96 )      4.97698783875e-05      3.34978103638e-05      5.96642494202e-05
( 128 , 128 )    0.000160992145538      6.54458999634e-05      0.000101208686829
( 160 , 160 )    0.000217020511627      7.35521316528e-05      9.74535942078e-05
( 192 , 192 )    0.00027722120285       8.89897346497e-05      8.29100608826e-05
( 224 , 224 )    0.000314295291901      8.71419906616e-05      0.000101983547211
( 256 , 256 )    0.000389456748962      0.000100493431091      9.80496406555e-05
( 288 , 288 )    0.000437796115875      8.42809677124e-05      8.35061073303e-05
( 320 , 320 )    0.00068473815918       8.38041305542e-05      9.72151756287e-05
( 352 , 352 )    0.000640392303467      8.27312469482e-05      0.000103890895844
( 384 , 384 )    0.000773310661316      8.85128974915e-05      9.97185707092e-05
( 416 , 416 )    0.000898957252502      8.64267349243e-05      8.49962234497e-05
( 448 , 448 )    0.00130522251129       8.79764556885e-05      9.9778175354e-05
( 480 , 480 )    0.00113350152969       7.45058059692e-05      8.42809677124e-05
( 512 , 512 )    0.00154680013657       8.38041305542e-05      0.000117719173431
( 544 , 544 )    0.00169318914413       7.82012939453e-05      0.000101983547211
( 576 , 576 )    0.00196695327759       9.5009803772e-05       0.00010222196579
( 608 , 608 )    0.0023946762085        9.69767570496e-05      0.00010222196579
( 640 , 640 )    0.00250351428986       8.1479549408e-05       0.00012594461441
( 672 , 672 )    0.00294721126556       8.63075256348e-05      8.67247581482e-05
( 704 , 704 )    0.00402879714966       8.44597816467e-05      9.00626182556e-05
( 736 , 736 )    0.00343728065491       8.59498977661e-05      0.000108420848846
( 768 , 768 )    0.00459796190262       9.31620597839e-05      0.000100493431091
( 800 , 800 )    0.00431150197983       9.07778739929e-05      0.00010347366333
( 832 , 832 )    0.00460696220398       8.55326652527e-05      9.60230827332e-05
( 864 , 864 )    0.00499552488327       9.29832458496e-05      0.000102281570435
( 896 , 896 )    0.0061132311821        0.000135958194733      0.000100314617157
( 928 , 928 )    0.00693893432617       0.000101923942566      9.49501991272e-05
( 960 , 960 )    0.00754100084305       8.35061073303e-05      9.75131988525e-05
( 992 , 992 )    0.00824499130249       8.32676887512e-05      8.74996185303e-05
( 1024 , 1024 ) 0.00912719964981       9.27448272705e-05      0.000101447105408
( 1056 , 1056 ) 0.00951153039932       9.97185707092e-05      0.000103533267975
( 1088 , 1088 ) 0.0104519724846        8.82148742676e-05      9.54866409302e-05
( 1120 , 1120 ) 0.0103297233582        9.11951065063e-05      0.000102698802948
( 1152 , 1152 ) 0.0117694735527        8.52346420288e-05      0.000189542770386
( 1184 , 1184 ) 0.0180364251137        9.57846641541e-05      9.03010368347e-05
( 1216 , 1216 ) 0.0187122225761        9.42349433899e-05      0.000101149082184
( 1248 , 1248 ) 0.0158684849739        8.1479549408e-05       8.07642936707e-05

After ( 128 , 128 ) pyopenCL Tiling is faster than python.
After ( 96 , 96 ) pyopenCL Naive is faster than python.
Avg speedup factor for multiplication using GPU is: 195.61658654
```

Thus as we can see above the output values produced by the two kernels using PyOpenCl and the GPU match the output for matrix product produced by Python and numpy using the CPU. Also we can after what input size of the square matrices the PyOpenCL implementations for Naïve Kernel and Tiling Kernel are faster than the regular Python-numpy implementations. We also can see a **very large speedup factor** for case of the largest input, which for this run of the algorithm was **195.61**

Now we also wish to visualize the performance of the two kernel implementations with respect to the dimensions of the matrix. Thus we plot the graph for Time vs Size for different inputs. A snapshot of the code and output are given below:

```python
### Comparing python & pyopenCL timings ###

python_times=[]
pyopencl_op_naive_times=[]
pyopencl_op_tiling_times=[]

param=np.arange(1,40,1).astype(np.int32)

for i in param:
        python_times.append(py_calc_time(i*SIZE,4))
        pyopencl_op_naive_times.append(cl_op_naive_time(i*SIZE,4))
        pyopencl_op_tiling_times.append(cl_op_tiling_time(SIZE,i*SIZE,i*SIZE,i*SIZE,4))

l_index=len(python_times)-1
naive_speedup=python_times[l_index]/pyopencl_op_naive_times[l_index]
tiling_speedup=python_times[l_index]/pyopencl_op_tiling_times[l_index]


print "\nDim\t", "\tPython_time\t", "\tNaive_time\t", "Tiling_time\t"
for i in param:
        print "(",i*SIZE, ",",i*SIZE,")\t", python_times[i-1],"\t", pyopencl_op_naive_times[i-1], "\t", pyopencl_op_tiling_times[i-1], "\t"

for i in param:
    if pyopencl_op_tiling_times[i-1]<python_times[i-1]:
        print "\nAfter (", i*SIZE, ",",i*SIZE, ") pyopenCL Tiling is faster than python."
        break
for i in param:
    if pyopencl_op_naive_times[i-1]<python_times[i-1]:
                print "After (", i*SIZE, ",", i*SIZE,") pyopenCL Naive is faster than python."
                break

print "Avg speedup factor for multiplication is:", (tiling_speedup +naive_speedup)/2

plt.clf()
plt.plot(param*SIZE, python_times, 'bo-',
         param*SIZE, pyopencl_op_naive_times, 'r*-',
         param*SIZE, pyopencl_op_tiling_times, 'go-')

plt.xlabel('elements in square matrix A,B')
plt.ylabel('$t$')
plt.title('Time vs Size for different Multiplication Implementations')
plt.legend(('Python-CPU', 'Naive-Kernel-GPU', 'Tiling-Kernel-GPU'), loc='upper left')
plt.grid(True)
plt.gca().set_xlim((min(param*SIZE), max(param*SIZE)))
plt.gca().set_ylim((0, 1.2*max(python_times)))
#plt.draw()
plt.savefig('Multiplication_scaling.png')
```

The code also generates a plot for the Time vs Size for different Multiplication Implementations. A snapshot of the plot is given below: