Real-Time Learning of Material Constitutive Models Using Convolutional Neural Networks

By
Joseph Hanson
Raghav Sharma
Sanket Ugile

MAE 598: Design Optimization
Final Project Report
December 2nd, 2018

Abstract

This project investigates the use and optimization of using convolutional neural networks running on central processing units (CPU's) and graphical processing units (GPU's) for accelerating the process of parameter estimation of a material sample under thermal loading.

Table Of Contents

Introduction

The goal of this project was to optimize the prediction of the thermal conductivity of a uniform, isotropic material under 2D heat conduction using the Finite Element Method (FEM) and Convoluted Neural Networks (CNN) implemented in Python with the use of Tensorflow.

The 2D case was first used to study the effect of the problem scaling (dimensionality) on computation time for different implementations of the conjugate gradient (CG) method. The different CG implementations included using Numpy (1), Tensorflow using the tf.spare_tensor_matmul operation (2) and Tensorflow using the tf.nn.conv2d operation (3).

To go the other way and try to predict the conductivity of the sample for given a temperature field and force-boundary condition, a convolution matrix filter was defined and used to generate the loss function. This was then fed into the tf.AdamOptimizer (4) to create our CNN.

The results are discussed and possible reasons for errors and improvements that could be made are suggested.

Nomenclature

k: Heat conduction coefficient of the material
K: Global Conductivity Matrix
e: Error
f: Global Flux Vector
d: Nodal Temperature Vector

Mathematical Model

Problem Definition

We consider the following 2D heat conduction FEA problem shown in Figure 1 below.

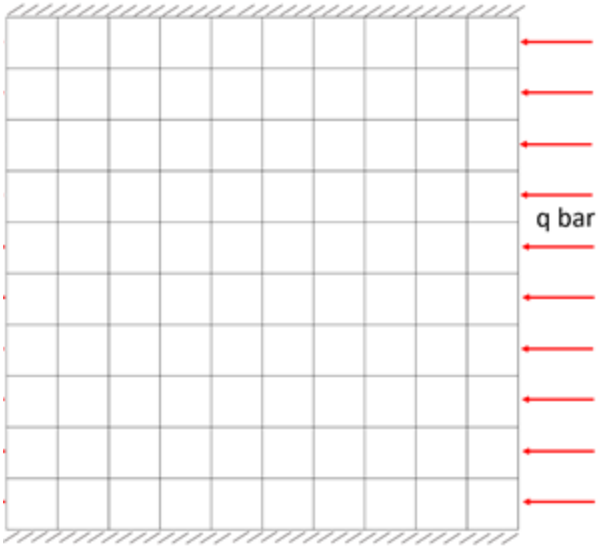

Figure 1: 2D plate with a fixed temperature on the left, adiabatic conditions on top and bottom with a heat flux entering from the right.

We consider a square plate with a constant heat flux of q bar = 10,000 W/m^2 entering from the right side, and which has a constant temperature, T = 0 K maintained on the left side of the plate. The top and bottom sides are assumed to be adiabatic. The material is given a thermal conductivity of 16 W/m.

Numerical Method: Finite Element Analysis (MATLAB)

An FEM MATLAB code was written and used to solve the described problem and get a temperature distribution across the domain. The code is found at https://github.com/hope-yao/MG_net/tree/MAE-598-Student-Project/MAE598_Project/data under the respectvie dimensionality folder.

The code was defined as follows:

Element Type: Linear square elements were used to discretize the domain.

Memory Allocation: The **K** matrix was defined as a sparse matrix so as to save on memory and computation time.

Procedure:

1. Discretization- The domain was divided into square elements.
2. Nodal Arrangement of Equations: We define the nodal conductivity matrix for each pair as well as the nodal flux vector for each arrangement.
3. Assembly: The nodal conductivity matrices and nodal flux vectors are arranged into global conductivity and global flux vectors as per the arrangement of the elements in the domain.
4. Correlation: The global flux vector, the global conductivity matrix, and the unknown temperature nodal values are related by the Fourier's law of heat conduction, as **f=Kd**

5.Solution: To invert the stiffness matrix in the MATLAB solution, in order to solve for temperature, ‘matrix left division’ was used. This is invoked by the line of code:

d=K\f;

Matrix Left Division:
We have the matrix equation:

f=Kd

To find d, we need to solve $\mathbf{d=(K^{-1})f.}$ This is equation was solved in MATLAB using the command **d=K\f;**. This means that $\mathbf{K^{-1}}$ comes on the left of **f**, hence the name. By this command, $\mathbf{K^{-1}}$ does not have to be computed separately, hence saving computation time. Figure 2 below illustrates the MATLAB FEM solution for the 10x10 element case and depicts the temperature distribution across the plate.

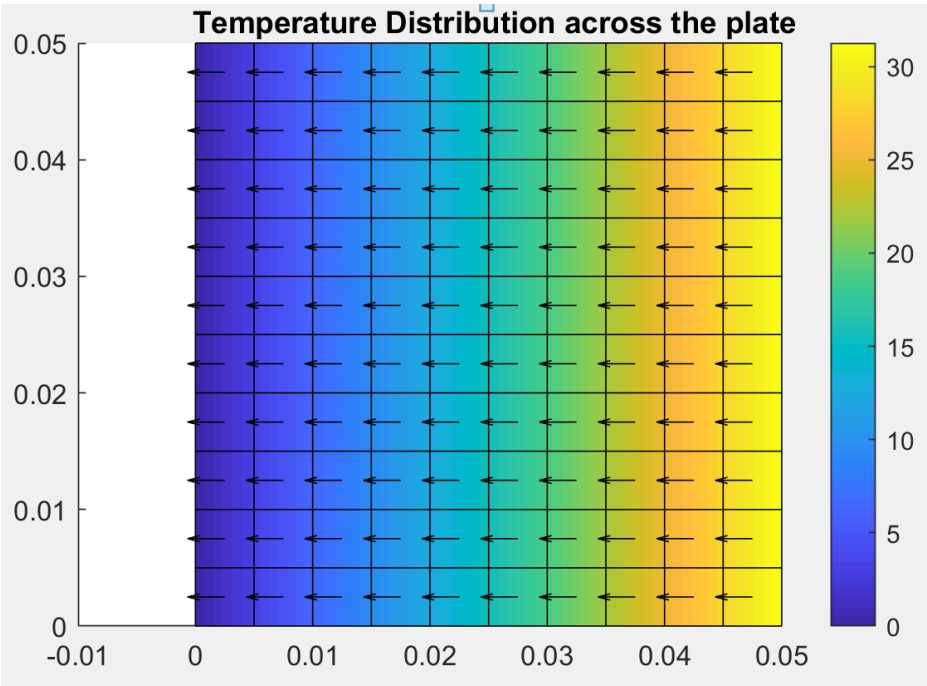

Figure 2: The temperature distribution for the 10x10 element MATLAB FEM solution.

Numerical Method: Conjugate Gradient Method (Python)

The conjugate gradient (CG) method is an algorithm used for solving large systems of linear equations and is best suited for use on matrices that are symmetric and positive semi-definite. It is implemented in an iterative fashion and is able to handle large, sparse matrices that cannot be handled by other direct methods. This method was chosen given

the expected behavior of 2D conduction problems which will have symmetric and positive semi-definite global conductivity matrices.

There are many variations of the CG method. Jonathan R Shewchuk's implementation of the CG algorithm is shown below in Figure 3 (5). Given the inputs A, b, a starting value x, a maximum number of iterations i_{max}, and an error tolerance $\varepsilon<1$:

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If i is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$

Figure 3: Jonathan R Shewchuk's implementation of the CG algorithm.

The method used as the base four our python code was based on the MATLAB example on Wikipedia as shown below in Figure 4 (6) and was modified to run in python.

```
function [x] = conjgrad(A, b, x)
    r = b - A * x;
    p = r;
    rsold = r' * r;

    for i = 1:length(b)
        Ap = A * p;
        alpha = rsold / (p' * Ap);
        x = x + alpha * p;
        r = r - alpha * Ap;
        rsnew = r' * r;
        if sqrt(rsnew) < 1e-10
              break;
        end
        p = r + (rsnew / rsold) * p;
        rsold = rsnew;
    end
end
```

Figure 4: MATLAB example implementation of the CG method found on Wikipedia, was used as basis for our code.

```python
def conjgrad(A, b, tol, x):
    n = len(b)
    r = b - A.dot(x)
    p = r
    rsold = np.dot(r.T, r)
    for i in range(n):
        Ap = A.dot(p)
        alpha = rsold / np.dot(p.T, Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = np.dot(r.T, r)
        if np.sqrt(rsnew) < tol:
            print('Itr:', i)
            break
        p = r + (rsnew / rsold) * p
        rsold = rsnew
    return x
```

Figure 5: Final Python implementation of the CG method.

The strictly Python implementation of the CG method is shown in Figure 5 above was validated using a toy matrix to confirm accuracy and the code can be found at https://github.com/hope-yao/MG_net/blob/MAE-598-Student-Project/MAE598_Project/2D_conduction_CG_py.py. This code imports the global conductivity matrix formulated with MATLAB and solves it with the CG method shown above. It runs the following dimensionality cases; 10x10 elements, 100x100 elements and 1000x1000 elements (shown in Table 1 below) and reports out the number of iterations to convergence and the time taken to solve. The time taken to solve as a function of the dimensionality is shown below in Figure 6, for the case of using a laptop CPU and the lab workstation CPU. The WS CPU shows a faster solve time which is expected given that the WS CPU has more cores and can run at higher frequencies than the laptop CPU.

Sr No.	Dimension of stifness matrix	No. of Nodes	No.of Elements
1	10 x 10	110 x 110	100
2	100 x 100	10100 x 10100	10000
3	1000 x 1000	1001000 x1001000	1000000

Table 1: Shows the different dimensionality cases and their definitions.

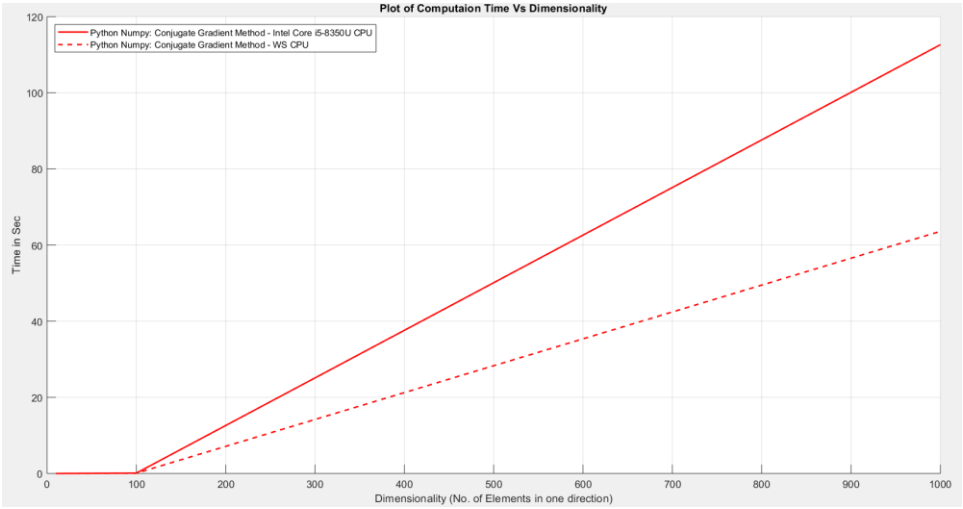

Figure 6: Shows the comparison of time vs dimensionality when running the strictly Python CG method on two different machines.

The baseline Python implementation of the CG method was modified to run using tf.sparse_tensor_dense_matmul Tensorflow operations in hopes of speeding up the solution time and is shown in Figure 7 below and can be found at [2D_conduction_CG_tf.py].

```
def conjgrad_tf(A_tf, b, x, n):
    #r = b - A.dot(x)
    r = b - tf.sparse_tensor_dense_matmul(A_tf, x, adjoint_a=False, adjoint_b=False, name=None)
    p = r
    #rsold = np.dot(r.T, r)
    rsold = tf.matmul(tf.transpose(r), r)
    for i in range(n):
        #Ap = A.dot(p)
        Ap = tf.sparse_tensor_dense_matmul(A_tf, p, adjoint_a=False, adjoint_b=False, name=None)
        #alpha = rsold / np.dot(p.T, Ap)
        alpha = rsold / tf.matmul(tf.transpose(p), Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        #rsnew = np.dot(r.T, r)
        rsnew = tf.matmul(tf.transpose(r), r)
        #print('Itr:', i)
        p = r + (rsnew / rsold) * p
        rsold = rsnew
    return x
```

Figure 7: Tensorflow implementation of CG method using tf.sparse_tensor_dense_matmul operations.

This implementation of the code also runs the different dimensionality cases with the number of CG loop iterations based on the convergence iteration number observed in the strictly Python version of the method. The time taken to solve as a function of the dimensionality is shown below in Figure 8, for the case of using a laptop CPU and the lab workstation CPU and GPU. Again the WS CPU outperforms the laptop CPU as would be expected. It was also expected that the GPU would run faster for higher dimensionality cases but as is shown, was not the case.

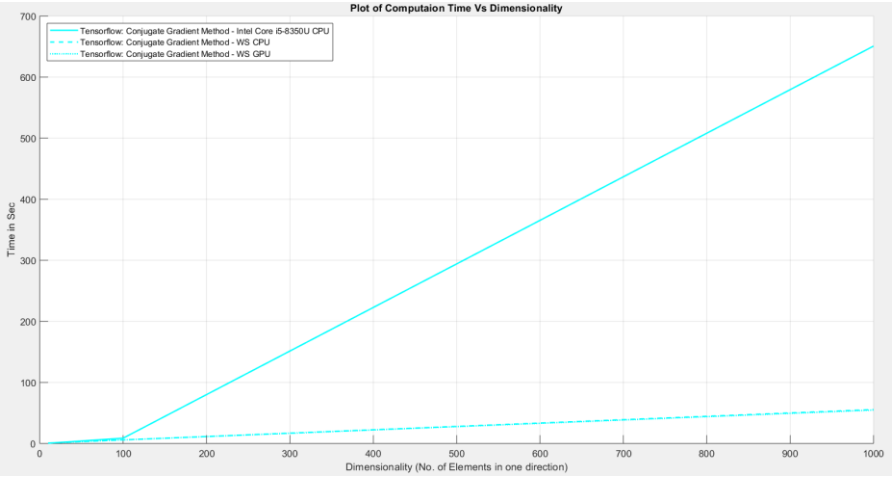

Figure 8: Shows the comparison of time vs dimensionality when running the Tensorflow implementation of the CG method using tf.sparse_tensor_dense_matmul operations on two different machines.

Convolution Matrix Filter

In an effort to further explore the speeding up the solve time and to open up the possibility of using convolution to solve and later predict, a 3X3 convolution matrix filter (shown in Figure 9 below) was defined and used on the generated stiffness matrix **K** so as to accelerate the time required to calculate the solution. The ‘tf.nn.conv2d’ command was used to implement the filter using the tensorflow library. The command is of the format:

```
tf.nn.conv2d(
    input,
    filter,
    strides,
    padding,
    use_cudnn_on_gpu=True, (Default=True)
    data_format='NHWC', (Optional)
    dilations=[1, 1, 1, 1], (optional)
    name=None (Optional)
)
```

Here, input is the matrix after padding, filter is the weights of the filter matrix, strides=[1,1,1,1] and padding='VALID'

```
# Filter
filter = np.asarray([[-5.3333, -5.3333, -5.3333], [-5.3333, 42.6667, -5.3333], [-5.3333, -5.3333, -5.3333]])
A_weights = np.reshape(filter, (3, 3, 1, 1))
```

Figure 9: 3x3 convolution filter used in the Tensorflow CG method using the tf.nn.conv2d operation.

The Tensorflow implementation of CG method using tf.sparse_tensor_dense_matmul operations was modified to run using the tf.nn.conv2d operation and is shown in Figure 10 below. The 10x10 case can be found at CG_tf_convolution_10.py.The other dimensions are also on the github.

```
def conjgrad_tf(A_weights, b, x, n):
    #r = b - A.dot(x)          # python method
    padded_x = tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]], "SYMMETRIC")
        # reshape
    A_dotx_conv = tf.nn.conv2d(input=padded_x, filter=A_weights, strides=[1, 1, 1, 1], padding='VALID')

    A_dotx_conv = tf.reshape(A_dotx_conv, (110,1))
    r = b - A_dotx_conv
    p = r
    #rsold = np.dot(r.T, r)      # python method
    rsold = tf.matmul(tf.transpose(r), r)
    for i in range(n):
        #Ap = A.dot(p)          # python method
        padded_p = tf.pad(tf.reshape(p, (1, 10, 11, 1)), [[0, 0], [1, 1], [1, 1], [0, 0]], "SYMMETRIC")
        Ap_c = tf.nn.conv2d(input=padded_p, filter=A_weights, strides=[1, 1, 1, 1], padding='VALID')
        Ap = tf.reshape(Ap_c, (110,1))
        # Ap = Ap_c[0, 0, :, :]
        #alpha = rsold / np.dot(p.T, Ap)          # python method
        alpha = rsold / tf.matmul(tf.transpose(p), Ap)
        x = tf.reshape(x, (110, 1))
        x = x + alpha * p
        r = r - alpha * Ap
        #rsnew = np.dot(r.T, r)      # python method
        rsnew = tf.matmul(tf.transpose(r), r)
        p = r + (rsnew / rsold) * p
        rsold = rsnew
        #print('Itr:', i)
    return x
```

Figure 10: Tensorflow implementation of CG method using the tf.nn.conv2d operations.

This implementation of the code only runs for one of the different dimensionality cases at a time. The time taken to solve as a function of the dimensionality is shown below in Figure 11, for the case of using a laptop CPU and the lab workstation CPU and GPU. Again the WS CPU outperforms the laptop CPU as would be expected. It was also expected that the GPU would run faster for higher dimensionality cases but again as is shown, that was not the case.

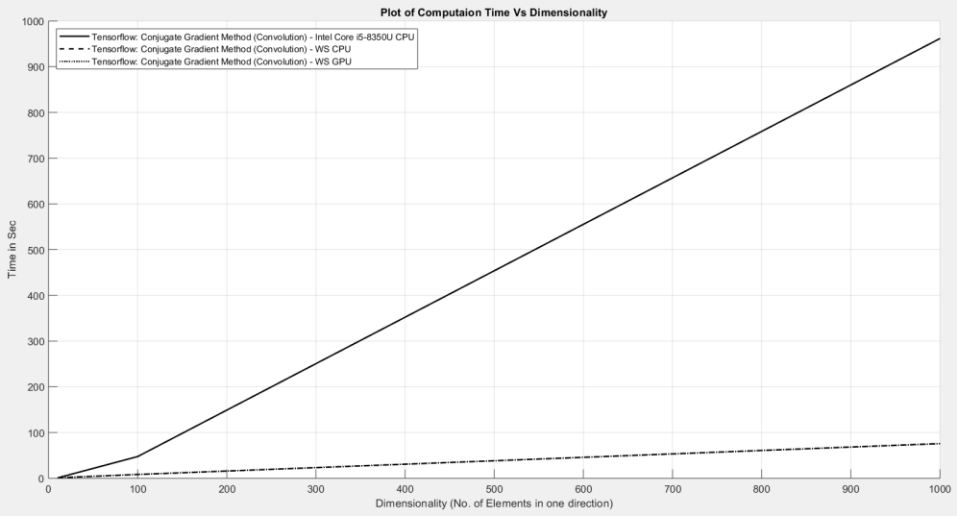

Figure 11: Shows the comparison of time vs dimensionality when running the Tensorflow implementation of the CG method using the tf.nn.conv2d operations on two different machines.

The comparison of the solve time vs dimensionality for all the different cases is shown below in Figure 12 and shows difference for the laptop CPU between the cases but not when run on the lab WS.

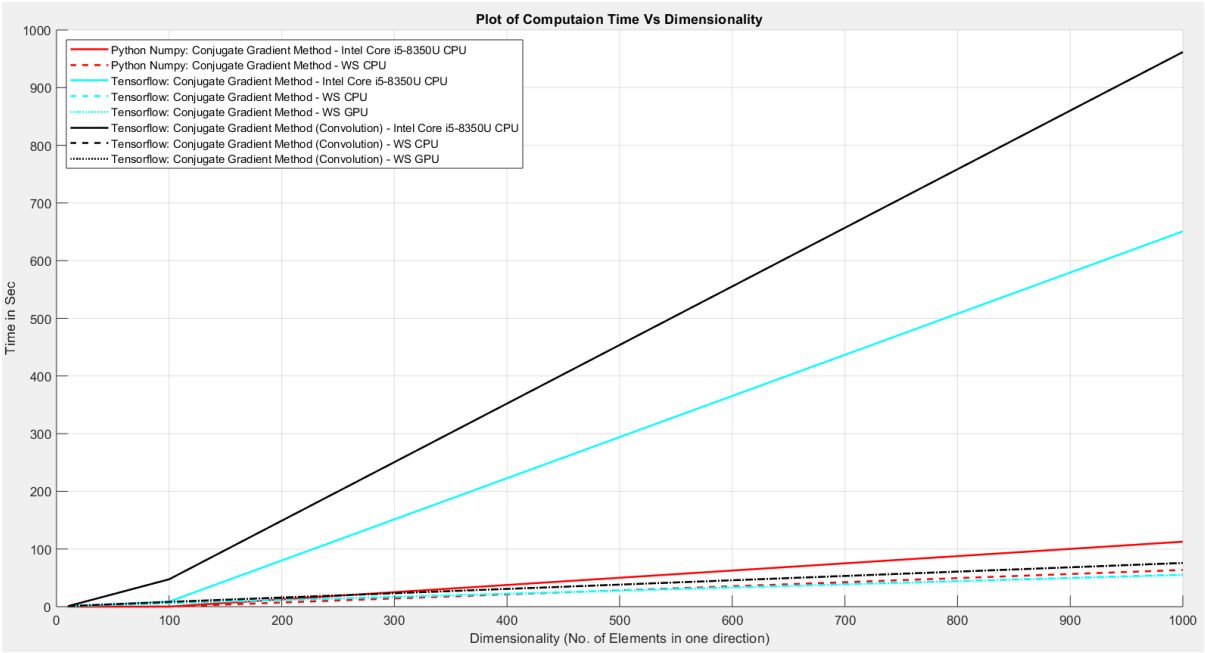

Figure 12: Shows the comparison of time vs dimensionality when running all of the implementations of the CG method on two different machines.

Sr No.	Dimension of stifness matrix	Method Used for Solving	Machine Used for solving	Computation time (sec)
1	10 x 10	**Python**: Conjugate Gradient	Intel Core i5-8350U **CPU**	**0.00074**
2	10 x 10	**Python**: Conjugate Gradient	Lab Work Staion-**CPU**	**0.00084**
3	10 x 10	**Tensorflow**:Conjugate Gradient	Intel Core i5-8350U **CPU**	**0.47984**
4	10 x 10	**Tensorflow**:Conjugate Gradient	Lab Work Staion-**CPU**	**0.72283**
5	10 x 10	**Tensorflow**:Conjugate Gradient	Lab Work Staion-**GPU**	**0.73323**
6	10 x 10	**Tensorflow**:Conjugate Gradient (**Convolution**)	Intel Core i5-8350U **CPU**	**0.72087**
7	10 x 10	**Tensorflow**:Conjugate Gradient (**Convolution**)	Lab Work Staion-**CPU**	**0.99609**
8	10 x 10	**Tensorflow**:Conjugate Gradient (**Convolution**)	Lab Work Staion-**GPU**	**1.01424**
9	100 x 100	**Python**: Conjugate Gradient	Intel Core i5-8350U **CPU**	**0.10062**
10	100 x 100	**Python**: Conjugate Gradient	Lab Work Staion-**CPU**	**0.06895**
11	100 x 100	**Tensorflow**:Conjugate Gradient	Intel Core i5-8350U **CPU**	**8.74552**
12	100 x 100	**Tensorflow**:Conjugate Gradient	Lab Work Staion-**CPU**	**5.92201**
13	100 x 100	**Tensorflow**:Conjugate Gradient	Lab Work Staion-**GPU**	**5.92948**
14	100 x 100	**Tensorflow**:Conjugate Gradient (**Convolution**)	Intel Core i5-8350U **CPU**	**47.44112**
15	100 x 100	**Tensorflow**:Conjugate Gradient (**Convolution**)	Lab Work Staion-**CPU**	**8.19984**
16	100 x 100	**Tensorflow**:Conjugate Gradient (**Convolution**)	Lab Work Staion-**GPU**	**8.34345**
17	1000 x 1000	**Python**: Conjugate Gradient	Intel Core i5-8350U **CPU**	**112.65326**
18	1000 x 1000	**Python**: Conjugate Gradient	Lab Work Staion-**CPU**	**63.62284**
19	1000 x 1000	**Tensorflow**:Conjugate Gradient	Intel Core i5-8350U **CPU**	**650.71783**
20	1000 x 1000	**Tensorflow**:Conjugate Gradient	Lab Work Staion-**CPU**	**55.55385**
21	1000 x 1000	**Tensorflow**:Conjugate Gradient	Lab Work Staion-**GPU**	**54.82987**
22	1000 x 1000	**Tensorflow**:Conjugate Gradient (**Convolution**)	Intel Core i5-8350U **CPU**	**961.71353**
23	1000 x 1000	**Tensorflow**:Conjugate Gradient (**Convolution**)	Lab Work Staion-**CPU**	**75.8212**
24	1000 x 1000	**Tensorflow**:Conjugate Gradient (**Convolution**)	Lab Work Staion-**GPU**	**75.6294**

Table 2: Is a summary of all of the test cases run for the different implementations and machines used.

Optimization Study

Material Property Prediction

Our main objective was to minimize the following function and show it is possible to predict material properties for a given temperature field, u, and boundary force, f, where A is the global conductivity matrix.

$$f = Au$$
$$A(\theta)u = f$$
$$error\ (e) = ||(A(\theta)u = f)||$$
$$\mathbf{min_\theta}\ ||e||^2$$

The prediction code was based off of example code given to us by Houpu Yao and our Tensorflow implementation of the CG method using the tf.nn.conv2d operations. His convolution filter definition (shown in Figure 13 below) was adapted for our prediction code as shown.

```
conductivity = tf.Variable(1., tf.float32)
# Filter
filter = np.asarray([[1, 1, 1], [1, 0, 1], [1, 1, 1]])
A_weights = np.reshape(filter, (3, 3, 1, 1))* conductivity
```

Figure 13: The 3x3 convolution filter used in the CNN prediction code.

The tf.train.AdamOptimizer was used to setup the CNN prediction code and the loss function was defined as shown in figure 14 below.

```
CGpy_result = conjgrad_tf(A_weights, b, x, n)
x = tf.reshape(x, (110, 1))
# optimizer
CGpy_result['loss'] = loss = tf.reduce_mean(tf.abs(CGpy_result['final'] - x))
lr = 1
learning_rate = tf.Variable(lr)  # learning rate for optimizer
optimizer = tf.train.AdamOptimizer(learning_rate)  #
grads = optimizer.compute_gradients(loss)
train_op = optimizer.apply_gradients(grads)
```

Figure 14: Shows the loss function definition for the CNN prediction code.

The 10x10 case for the CNN prediction code can be found at [tf_2D_ConductionMatirx_prediction_10.py](). The other dimensions are also on the github. The training loop was defined as shown in Figure 15.

```
test_loss_hist = []
train_loss_hist = []
k_value_hist = []
for itr in range(500):
    for i in range(1):
        x_input = x
        b_input = b10
        feed_dict_train = {b: b_input, x_input_pl: x_input}
        _, loss_value, k_value = sess.run([train_op, loss, conductivity], feed_dict_train)

        print("iter:{}  train_cost: {}  k_value: {}".format(itr, np.mean(loss_value), k_value))
```

Figure 15: Shows the CNN prediction training loop used.

The CNN prediction code is run for each case and the results are shown in Table 3 below. It is seen that the conductivity predictions are not accurate with large error in both directions. Possible causes are discussed in the next section.

Sr No.	Dimension of stifness matrix	Predicted Conductivity (W/mK)
1	10 x 10	28.3405
2	100 x 100	-5.5589
3	1000 x 1000	48.6953

Table 3: Results of the CNN prediction code using the convolution filter defined above.

Discussion of Results

After implementing the different CG methods using Python and different Tensorflow techniques, there appear to be some possible roadblocks as currently applied. It was expected that there would be measurable difference between the CPU and GPU machines, especially using the convolution method but very little if any difference was observed. Figure 16 below is the comparison of time vs dimensionality for the Tensorflow implementation of the CG method using tf.sparse_tensor_dense_matmul operations vs using the tf.nn.conv2d operations on the lab WS using CPU and GPU and clearly shows the improvement in solving time when using convolution as the dimensionality scales.

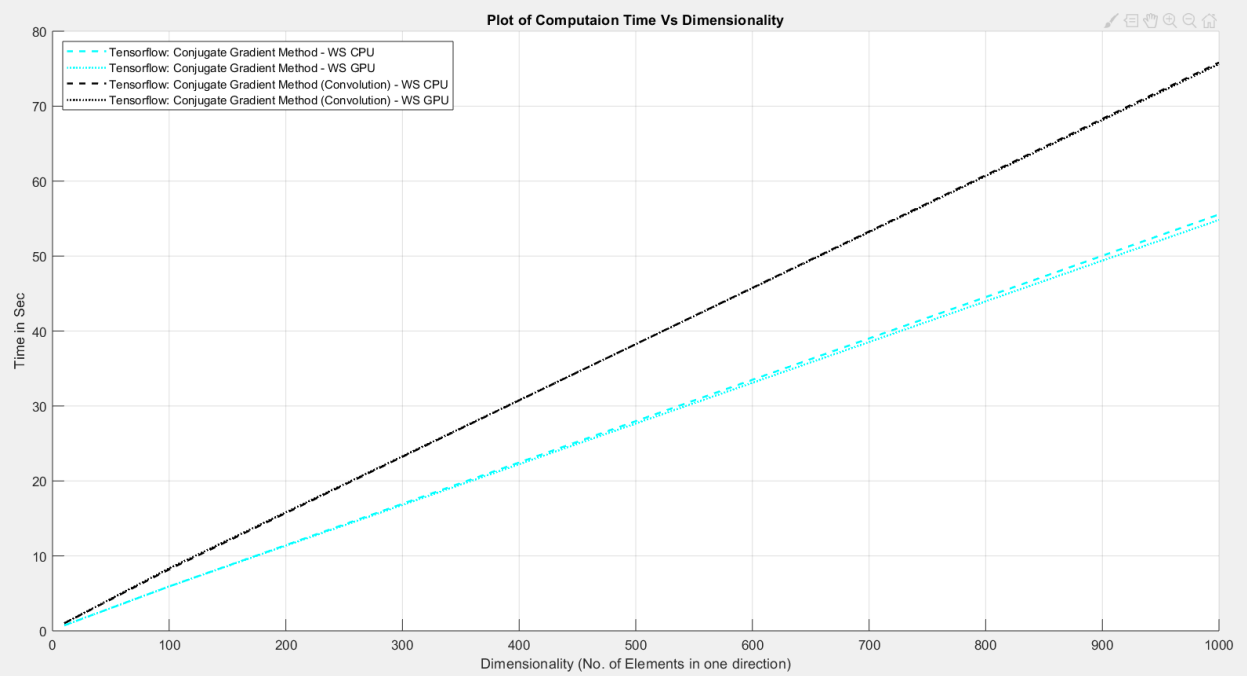

Figure 16: Shows the comparison of time vs dimensionality for the Tensorflow implementation of the CG method using tf.sparse_tensor_dense_matmul operations vs using the tf.nn.conv2d operations on the lab WS using CPU and GPU.

The CNN prediction code was very unstable as seen by the results in Table 3. Some trials were run to see how the prediction varied as the number of iterations (n) was changed but no predictable/understandable pattern/improvement was seen. Changes from advise given by Hope was also implemented as seen in the code here, https://github.com/hope-yao/MG_net/blob/MAE-598-Student-Project/MAE598_Project/tf_2D_ConductionMatirx_prediction_hope_changes.py, without any improvement seen.

It is possible that the convolution method as currently implemented is causing the issue in inaccurate predictions. We were unable to get a multiple matrices filter working and as such expected to see some small error at the boundaries of the problem. More work would need to be done to implement the convolution filter more accurately.

It was also attempted to implement a prediction using the tf.sparse_tensor_dense_matmul implantation but due to the way the global conductivity matrix is imported and not a variable, the gradient cannot be taken. Code can be found here, tf_2D_Conductionmatrix_prediction_pythonCG.py. To alleviate this, the FEM code would need to be implemented in Python so as to have the conductivity as a variable.

References

1. NumPy. (2018). Retrieved October 9, 2018, from http://www.numpy.org/
2. Tf.sparse.matmul | TensorFlow. (n.d.). Retrieved October 14, 2018, from https://www.tensorflow.org/api_docs/python/tf/sparse/matmul
3. Tf.nn.conv2d | TensorFlow. (n.d.). Retrieved October 14, 2018, from https://www.tensorflow.org/api_docs/python/tf/nn/conv2d
4. Tf.train.AdamOptimizer | TensorFlow. (n.d.). Retrieved October 16, 2018, from https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer
5. Jonathan R Shewchuk. 1994. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical Report. Carnegie Mellon Univ., Pittsburgh, PA, USA. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf
6. Conjugate gradient method. (2018, November 29). Retrieved October 14, 2018, from https://en.wikipedia.org/wiki/Conjugate_gradient_method

Appendices

Note: All codes have been updated on the Github link mentioned above.