Optimizing Propagation and Sparsity Enforcement for a Fully Connected Scalable Feedforward Sparse Autoencoder

David Klaus Alex Welles

May 6, 2014
Dept. of Electrical and Computer Engineering
Boston University

1 Sparse Autoencoder

An autoencoder is an unsupervised learning algorithm that applies forward and back propagation across a neural network to learn a function identifying a feature of the input data [1]. A neural network is a structure of connected computational nodes (or neurons) that each perform linear regression, connected in such a way that the outputs of one or more nodes serve as the input for one or more nodes (figure 1). The structured layering of a neural network's nodes allows a complex non-linear hypothesis equation (output) to be defined from much simpler components. The layer of the network that accepts data is the input layer and the layer defining the hypothesis function is the output layer. Although only one is shown in the figure below, the intervening layers between the input and output are referred to as hidden layers.

An autoencoder is said to be "sparse" when the rate of activation of nodes within its hidden layers is constrained to reduce the rate of activation. Enforcing sparsity has been shown to be beneficial to extract a variety of features that are beyond the scope of this project but, the process of extracting the feature remains the same. In the case where the hidden layers of an autoencoder are smaller than the input and output layers, the network is learning a compressed representation of the input data. Regardless of the size of the hidden layer or the sparsity of the autoencoder, feature extraction from an autoencoder requires that the network be trained to recognize a feature. The process of training a sparse auto-encoder requires two procedures general to all autoencoders, forward propagation and backward propagation, and a unique procedure called sparsity enforcement. This report discusses various methods of optimizing these procedures assuming a scalable architecture described in the next section.

1.1 Establishing a "Gold Standard"

To establish a baseline against which the timing and results of other implementations could be compared, we used the default values of a feedforward network with input and output layers comprised of 64 nodes and a hidden layer of 25 nodes (figure 1). Here feedforward means that there are no cycles or loops and each of the layers is fully connected to the next. The number of nodes we used for our base implementation were given by the assignment on sparse autoencoders created by Dr. Andrew Ng (Stanford University) [1]. Several of the graphics used in our slides and this report have been taken or modified from that assignment.

Our gold standard is Matlab code we wrote to complete Dr. Ng's assignment. Initialization and input values generated by that code were exported and used as inputs for our various implementations in C and results were matched against those of the Matlab code for the purposes of error checking. The Matlab code was chosen as a "gold standard" as it successfullly extracted edge characteristics from the inputs provided by the assignment. The initialization and input values (all floats) were transferred to the C implementations

by exporting them as CSVs and then reading them in to the various C programs before timing began.

Although our initial code only implemented a modestly sized neural network, later we wrote optimized versions that are capable of creating autoencoders with layers of any number of nodes. However, autoencoders larger than the basic version have not been properly modified to extract features from a particular type of data. However the autoencoder's that we have written can be used for such a purpose given that values for λ, β, and the sparsity constraint ρ are empirically found.

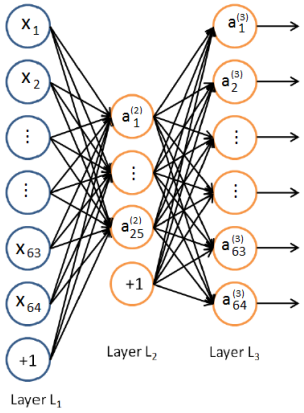

Figure 1: Architecture of an autoencoder. Note: layers are fully connected.

1.2 How a Sparse Autoencoder Works

The pseudocode shown below maps the process of training the sparse autoencoder using forward propagation, back propagation, and sparsity enforcement. Sparsity enforcement is incorporated in the back propagation process. The sub sections below describe these processes in detail while section 1.3 describes how these processes were implemented in code.

Data: Image inputs, initialized weight vectors
Result: One iteration of network training
for i = *0* ***to*** *10000 sample inputs* **do**
  (1) calculate $\hat{\rho}$ (sparsity parameter);
  (2) forward propagate $x^{(1)} \rightarrow a^{(2)}$;
  (3) forward propagate $a^{(2)} \rightarrow a^{(3)}$;
  (4) back propagate (calculate $d^{(3)}$);
  (5) back propagate (calculate $d^{(2)}$);
  (6) enforce sparsity on $d^{(2)}$;
end

Algorithm 1: Pseudocode for training a sparse autoencoder.

1.2.1 Forward Propagation

Forward propagation is the process of calculating the outputs of each layer of the autoencoder given specific input values. The structure of our base case neural network uses two layers of linear equations to learn a compressed representation of an input image. The input layer takes 64 normalized pixel values (floats) from an 8x8 "patch" of an input image. These pixel inputs can be thought of as the input "nodes" of the autoencoder. Layer two, the hidden layer, maps each of the inputs to 25 values via 64 linear equations for each of the 25 internal nodes. This results in 25 linear equations defined by 64 weights contained in the $W^{(1)}$

matrix not including an intercept or bias term ($b^{(1)}$ as defined in the code and the figure above).

Similarly, layer three consists of 64 nodes the output of which is compared to the true pixel values of the images. These 64 output nodes maps from each of the 25 hidden layer nodes via 64 linear equations defined by 25 weights each contained in the $W^{(2)}$ matrix (and a bias vector $b^{(2)}$). This structure is described by the following equations where a, b, x, and z are vectors and W is a matrix. Here the superscript in parentheses maps to the layer of the matrix, the function f() is the sigmoid function, x is an input value, and $h_{w,b}(x)$ is the hypothesis function which produces the output.

$$z^{(2)} = W^{(1)}x + b^{(1)} \tag{1}$$

$$a^{(2)} = f(z^{(2)}) \tag{2}$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{3}$$

$$h_{w,b}(x) = a^{(3)} = f(z^{(3)}) \tag{4}$$

1.2.2 Back Propagation

Backpropagation is performed in order to minimize the one-half squared-error cost function $J(w, b)$ shown in equation 5. This is done by including the error function in the calculation of the output error (by multiplying by the difference between the training output and input value) as shown in equation 6 and then propagating each contributing node and corresponding weights via equation 7. The λ is an empirically determined value which controls the degree to which the regularization term (the second term) is applied to the cost function. This is done to reduce the weights of each linear equation in order to prevent over fitting.

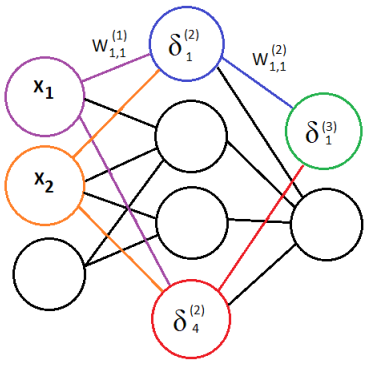

Figure 2: Tracing back propagation of an output node.

Our neural network "learns" or trains by backpropagation using gradient descent. The weights contained in $W^{(1)}$ and $W^{(2)}$ are initialized to random values as determined by sampling from a normal gaussian distribution with a mean of 0 and variation of 0.01 in order to prevent potentially weighting all inputs equally.

Error terms ($\delta^{(2,3)}$) are calculated for each node by taking a weighted average of the partial derivatives of all weights that contributed to that node's output. As an example (figure 2), the error term $\delta_1^{(3)}$ would be calculated by taking a weighted average of the partial derivatives of $W_{1,1}^{(2)}$ (blue) and $W_{1,4}^{(2)}$ (red) and multiplying it by the difference of the output of the batch training input and the expected value (equation 5). The error terms for the blue and red nodes, $\delta_1^{(2)}$ and $\delta_4^{(2)}$ respectively, are calculated by taking the weighted average of the weights contributing to their outputs. In the case illustrated in the figure below, $\delta_1^{(2)}$ and $\delta_4^{(2)}$ would both be calculated from the partial derivatives of the weights from input nodes x_1 and x_2. Specifically, $\delta_1^{(2)}$ would be derived from $W_{1,1}^{(1)}$ and $W_{2,1}^{(1)}$ while $\delta^{(2)_4}$ is derived from $W_{1,4}^{(1)}$ and $W_{2,4}^{(1)}$. The

weights of the linear equations describing the input from one layer of nodes to the next determine, how "responsible" that particular node is in terms of contributing to the overall error of the output node from which back propagation began.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{2m}||h_{w,b}(x^{(i)}) - y^{(i)}||^2\right] + \frac{\lambda}{2}\sum_{l=1}^{2}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2 \tag{5}$$

$$\delta_i^{(3)} = \frac{\partial}{\partial z_i^3}\frac{1}{2}||y - h_{w,b}(x)||^2 = -(y_i - a_i^{(3)}) \cdot f'(z_i^{(3)}) \tag{6}$$

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(2)}\delta_j^{(3)}\right) f'(z_i^{(2)}) \tag{7}$$

Because we do not have a closed form solution, as can be found in the case of a single linear regression, batch gradient descent (or conjugate gradient) is used to locate minima of the squared error function after calling the . Although we used the L-BFGS minFunc Matlab plugin to perform the gradient descent for us, we must first compute the necessary partial derivatives via equations 8 and 9 and sum them over the batch of 10,000 training inputs via equations 10 and 11. Equations 8 and 10 are with respect to the weight matrices while 9 and 11 are with respect to the bias matrices. The values produced by equations 10 and 11 are passed to the minFunc method which then performs gradient descent for 400 iterations.

$$\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b) = \nabla_{W^{(l)}} = (a_j^{(l)})\delta^{(l+1)} \tag{8}$$

$$\frac{\partial}{\partial b_{ij}^{(l)}}J(W,b) = \nabla_{b^{(l)}} = \delta^{(l+1)} \tag{9}$$

$$\nabla_{W^{(l)}} = \nabla_{W^{(l)}} + (a_j^{(l)})\delta^{(l+1)} \tag{10}$$

$$\nabla_{b^{(l)}} = \nabla_{b^{(l)}} + \delta^{(l+1)} \tag{11}$$

1.2.3 Sparsity Enforcement

We imposed a sparsity constraint on the hidden nodes of our neural network to ensure hidden node inactivity roughly 99% of the time. This is done by adding an activity penalty to the error term of our hidden layers (equation 12).

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)}\delta_j^{(3)}\right) + \beta\left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}\right)\right) f'(z_i^{(2)}) \tag{12}$$

This is done using Kullback-Leibler (KL) divergence between the Bernoulli random variable with mean *rho* and another Bernoulli random variable with mean $\hat{\rho}$. Thus, we calculate $\hat{\rho}$ during an initial forward propagation of all sample patches and then apply the KL-divergence penalty on the second (lines 57, 78, and 79 of figure 2). As a node's activation more greatly diverges from 1%, the KL penalty applied to that node increases. However, the term β controls the weight of the sparsity penalty term and an optimal value must be determined empircally.

1.3 Code Optimizations

The sections below present partial results for each implementation of the autoencoder. A full table and charts are available along with a summary in the results section. All data is shown for code run on the High Performance Computational Lab machines in Photonics 207 although some anecdotal results are shared concerning grid ME and BME machines. A Matlab implementation of the sparse autoencoder is used as a comparative baseline and a goal to beat.

1.3.1 Naive Serial (Really Bad Code)

Our naive serial implementation (figure 3 left) was modeled off of our Matlab code and used numerous vector functions (figure 3 right). The use of so many functions created an excessive number of jumps due to function calls and made it impossible to optimize using loop unrolling or openMP. The macro values "VISIBLE_SIZE" and "HIDDEN_SIZE" refer to the sizes of the input and output layers and hidden layer respectively. Data is stored in contiguous one dimensional arrays. This code ran incredibly slowly without optimization flags set. Optimzation flag 1 significantly sped up the runtime of this code. Optimization flags 1 through 3 differed very little in terms of total run time (figure 4). This code served as a first attempt at transforming our Matlab code into C code.

```
for( j = 0;j < VISIBLE_SIZE;j++ )
{
    dotPdt(h_a2,&h_W2[j*HIDDEN_SIZE],&h_z3[j],VISIBLE_SIZE);
}

addVectors(h_z3,h_b2,h_z3,VISIBLE_SIZE);
vectElemSigmoid(h_z3,h_a3,VISIBLE_SIZE);
initializeVector(h_temp1, VISIBLE_SIZE, 1.0);
subVectors(h_temp1,h_a3,h_temp1,VISIBLE_SIZE);
vectElemVectMult(h_a3,h_temp1,h_temp1,VISIBLE_SIZE);
subVectors(h_a3,&h_inputs[i],h_temp2,VISIBLE_SIZE);
vectElemVectMult(h_temp1,h_temp2,h_d3,VISIBLE_SIZE);
```

Forward propagation from $a^{(2)}$ to $a^{(3)}$.

```
void dotPdt(float* src1,float* src2, float *dest, int length){
    float accum = 0;
    int i = 0;
    for( i = 0; i< length;i++){
        accum += src1[i] * src2[i];
    }
    *dest = accum;
}
```

An example vector function.

Figure 3: An example of the naive serial code implementation.

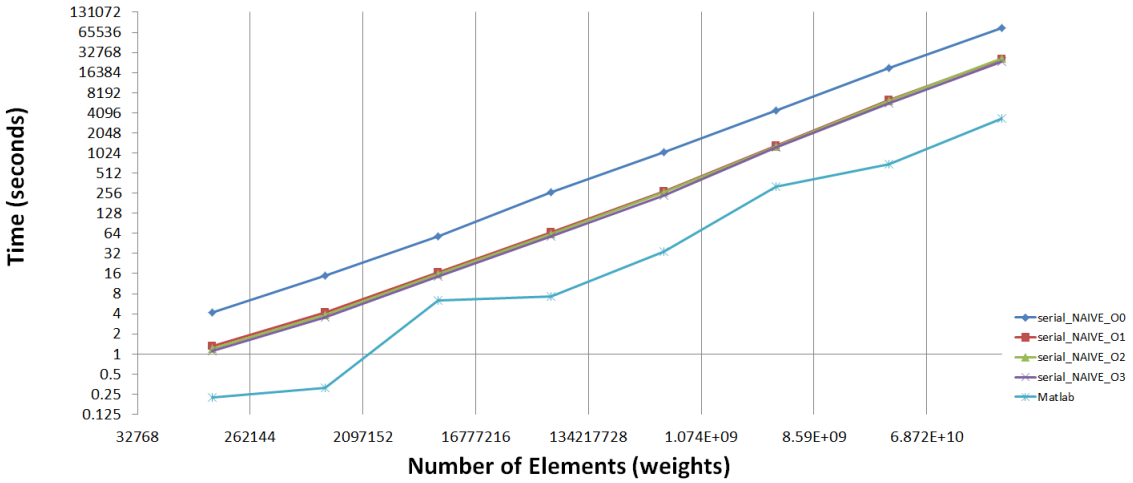

Figure 4: Log log plot of runtime of naive serial sparse autoencoder implementation with different optimization flags.

Code Variant	64x32	128x64	256x128	512x256	1024x512	2048x1024	4096x2048	8192x4096
serial_naive_0	4.13	15.07	58.39	260.96	1045.78	4480.39	18990.84	76136.01
serial_naive_1	1.29	4.14	16.66	65.90	268.30	1321.22	6273.55	25763.80
serial_naive_2	1.20	3.85	15.56	63.41	266.31	1285.16	6242.11	26455.83
serial_naive_3	1.10	3.54	14.33	57.90	239.40	1229.89	5718.97	24152.99
Matlab	0.22	0.31	6.34	7.18	34.30	317.16	692.25	3404.98

Table 1: Timing results in seconds depending on autoencoder dimensions (input nodes x hidden nodes) for the naive serial implementation of the sparse autoencoder. The code variant number corresponds to optimization flag.

1.3.2 Baseline Serial

Although some code motion (use of accumulators and reassociation) was possible within each of the vector functions, the naive serial code was rewritten by "manually inlining" all of the functions into much more condensed code (figure 5). Speedup over the naive implementation was substantial (figure 6) and likely due to the use of accumulators and reduction of the overall number of jumps and branches created by duplicated loops and function calls. Similar to the naive implementation, including any of the compiler optimization flags better than trippled performance but showed no real difference between flags (figure 6). Overall our performance without optimization flags has improved by just over 2x (figures 4, 6).

```
//*************************************
//  FORWARD PROPAGATION x(1) --> a(2)
//*************************************

for( j = 0;j < HIDDEN_SIZE; j++ )
{
    accum1 = 0;
    for( k = 0; k < VISIBLE_SIZE ; k++ )
    {
        accum1 += h_inputs[ i + k ] * h_W1[ j * VISIBLE_SIZE + k ];
    }
    h_a2[j] = (float)(1/(1+exp(-accum1 - h_b1[j])));
}

//*************************************
//  FORWARD PROPAGATION a(2) --> a(3)
//*************************************

for( j = 0;j < VISIBLE_SIZE; j++ )
{
    accum1 = 0;
    for( k = 0; k < HIDDEN_SIZE; k++ )
    {
        accum1 += h_a2[ k ] * h_W2[ j * HIDDEN_SIZE + k ];
    }
    h_a3[j] = (float)(1/(1+exp(-accum1 - h_b2[j])));
}
```

Figure 5: Serial baseline code snippet.

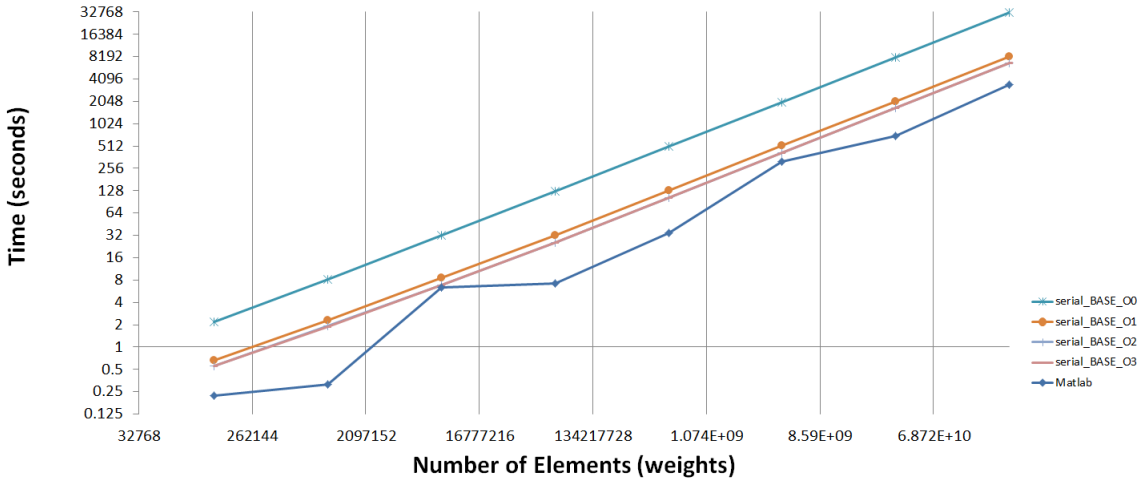

Figure 6: Log log plot of runtime of baseline serial sparse autoencoder implementation with different optimization flags.

Code Variant	64x32	128x64	256x128	512x256	1024x512	2048x1024	4096x2048	8192x4096
serial_base_0	2.17	8.24	31.94	125.71	501.79	2001.83	8003.89	31994.42
serial_base_1	0.66	2.30	8.51	32.24	129.46	516.00	2048.37	8183.33
serial_base_2	0.55	1.91	6.93	25.85	104.13	418.37	1668.43	6706.02
serial_base_3	0.55	1.90	6.91	25.91	104.24	418.25	1665.20	6717.96
Matlab	0.22	0.31	6.34	7.18	34.30	317.16	692.25	3404.98

Table 2: Timing results in seconds depending on autoencoder dimensions (input nodes x hidden nodes) for the baseline serial implementation of the sparse autoencoder. The code variant number corresponds to flag.

1.3.3 Loop Unrolling

To facilitate ease of testing different unrolling factors, C directives were used to allow for easy changes in number of loops unrolled (figure 7). The maximum number of loops unrolled for both J and K loops was 8; however, there is likely still room for some small gains for unrolling factors of 10 or 12. The gains from further loop unrolling are unlikely to result from shortening the critical path as floating point functional unit utilization is generally limited by data dependencies. However, a small reduction in runtime due to a removal of loop branches may be possible.

The non compiler optimized 8x loop unrolled serial code provided roughly a 3x speedup over the corresponding serial baseline implementation. As was the case with the baseline code, optimization flags provided a significant speedup. However also like the baseline speedup, there was little difference between the runtimes of the loop unrolled code using O1, O2, and O3.

```
for( j = 0; j < HIDDEN_SIZE; j++ )
{
    accum1 = 0;
    visByJ = VISIBLE_SIZE * j;
    for( k = 0; k < VISIBLE_SIZE ; k += NUM_K_LOOPS )
    {
        iPlusK = k + i;
        visByJPlusK = k + visByJ;
        #if   NUM_K_LOOPS == 1
            accum1 +=   h_inputs[ iPlusK ] * h_W1[ visByJPlusK ];
        #elif NUM_K_LOOPS == 2
            accum1 +=   h_inputs[ iPlusK ]     * h_W1[ visByJPlusK ]
                    +   h_inputs[ iPlusK + 1 ] * h_W1[ visByJPlusK + 1 ];
        #elif NUM_K_LOOPS == 3
            accum1 +=   h_inputs[ iPlusK ]     * h_W1[ visByJPlusK ]
                    +   h_inputs[ iPlusK + 1 ] * h_W1[ visByJPlusK + 1 ]
                    +   h_inputs[ iPlusK + 2 ] * h_W1[ visByJPlusK + 2 ];
        #elif NUM_K_LOOPS == 4
            accum1 +=   h_inputs[ iPlusK ]     * h_W1[ visByJPlusK ]
                    +   h_inputs[ iPlusK + 1 ] * h_W1[ visByJPlusK + 1 ]
                    +   h_inputs[ iPlusK + 2 ] * h_W1[ visByJPlusK + 2 ]
                    +   h_inputs[ iPlusK + 3 ] * h_W1[ visByJPlusK + 3 ];
```

Figure 7: Loop optimizations for forward propagation from $x^{(1)}$ to $a^{(2)}$.

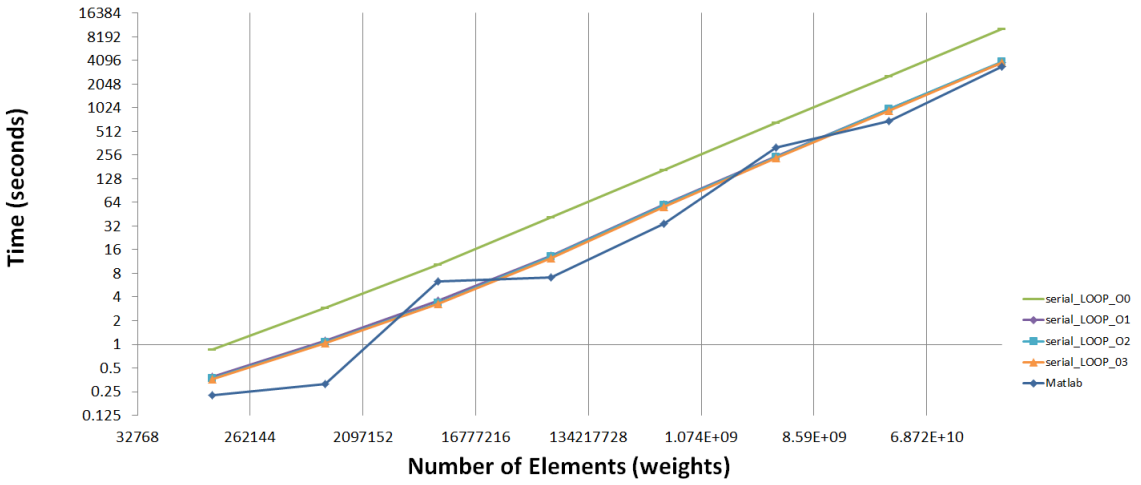

Figure 8: Log log plot of runtime of loop unrolled serial sparse autoencoder implementation with different optimization flags.

Code Variant	64x32	128x64	256x128	512x256	1024x512	2048x1024	4096x2048	8192x4096
serial_loop_0	0.85	2.89	10.37	41.04	165.37	659.59	2599.62	10441.16
serial_loop_1	0.38	1.12	3.58	13.53	60.22	246.56	977.70	3876.70
serial_loop_2	0.37	1.05	3.32	13.21	58.63	243.90	982.09	3991.94
serial_loop_3	0.36	1.04	3.30	12.61	56.52	233.29	932.28	3829.75
Matlab	0.22	0.31	6.34	7.18	34.30	317.16	692.25	3404.98

Table 3: Timing results in seconds depending on autoencoder dimensions (input nodes x hidden nodes) for the 8x loop unrolled serial implementation of the sparse autoencoder. The code variant number corresponds to flag.

1.4 OpenMP

OpenMP was not called around the larger I loop as elements within it required sequential operation and multiple critical sections or barriers. Thread setup and tear down over the 10000 loops in these situations often lead to much worse performance than the serial baseline. However, threading the smaller J loops did provide a modest benefit. Unfortunately, it proved too difficult or inefficient to combine loop unrolling with openMP without introducing errors to the results. It is also likely that such a combination would only be beneficial at extremely large data inputs.

The openMP variant performed roughly 3.7x worse than the loop unrolled variant and slightly worse than the serial baseline. We experimented on both the ME grid and BME grid trying various number of loops but only found improvements in performance over the baseline when the number of threads used was 2. The performance on the HPCL machines was considerably poorer than observed on the ME grid (figure 10). Overall, we were surprised at the negative impact additional threads had on the openMP variant's performance.

```
#pragma omp parallel shared(h_inputs, h_W1, h_b1, h_a2) private(i, j, k, accum1)
{
    #pragma omp for
    for( j = 0;j < HIDDEN_SIZE; j++ )
    {
        accum1 = 0;
        for( k = 0; k < VISIBLE_SIZE ; k++ )
        {
            accum1 += h_inputs[ i + k ] * h_W1[ j * VISIBLE_SIZE + k ];
        }
        h_a2[j] = (float)(1/(1+exp(-accum1 - h_b1[j])));
    }
}
```

Figure 9: OpenMP pragma calls added to forward propagation from $x^{(1)}$ to $a^{(2)}$.

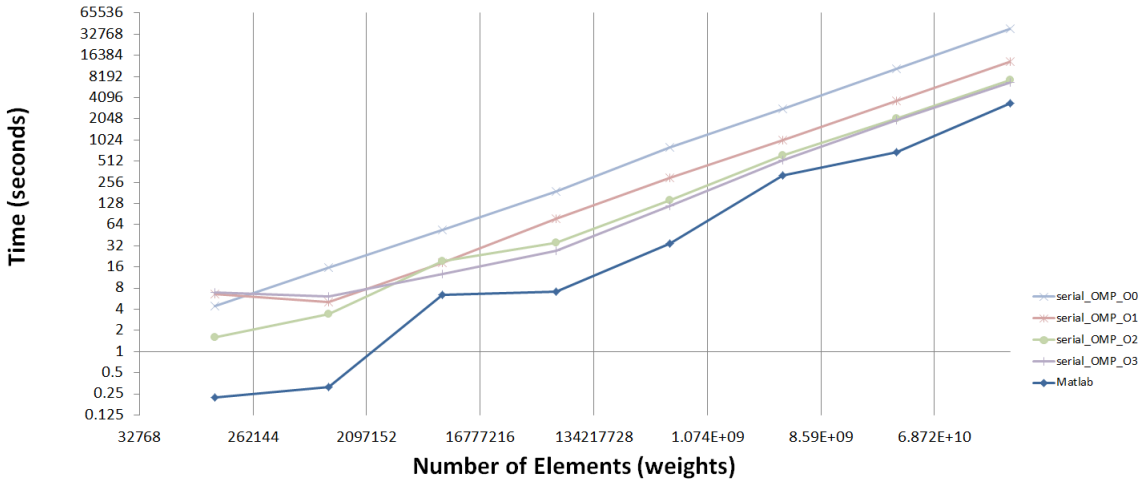

Figure 10: Log log plot of runtime of openMP serial sparse autoencoder implementation with different optimization flags.

Code Variant	64x32	128x64	256x128	512x256	1024x512	2048x1024	4096x2048	8192x4096
serial_omp_0	4.47	15.86	53.25	189.46	801.07	2851.26	10444.64	38913.18
serial_omp_1	6.61	5.01	18.37	78.55	297.12	1010.91	3649.42	13340.75
serial_omp_2	1.60	3.46	19.17	35.42	142.60	626.54	2063.90	7263.56
serial_omp_3	7.04	6.15	12.70	27.52	117.62	525.72	1972.42	6755.93
Matlab	0.22	0.31	6.34	7.18	34.30	317.16	692.25	3404.98

Table 4: Timing results in seconds depending on autoencoder dimensions (input nodes x hidden nodes) for the serial openMP implementation of the sparse autoencoder. The code variant number corresponds to flag.

1.4.1 CUDA

The initial set of CUDA kernels we wrote (sparseAutoencoder_CUDA_NAIVE.cu) utilized shared memory both to take advantage of faster on chip memory access times and to reduce memory efficiently while minimizing divergence (maximize warp utilization). Because the number of blocks in the grid is equal to the number of rows and the number of threads in a block is equal to the number of elements in a row, each thread is responsible for one multiplication only (figure 11). Furthermore, contiguous chunks of data are used allowing each thread to access and store based on its thread ID. This removes bank conflicts and potential serialization slow down.

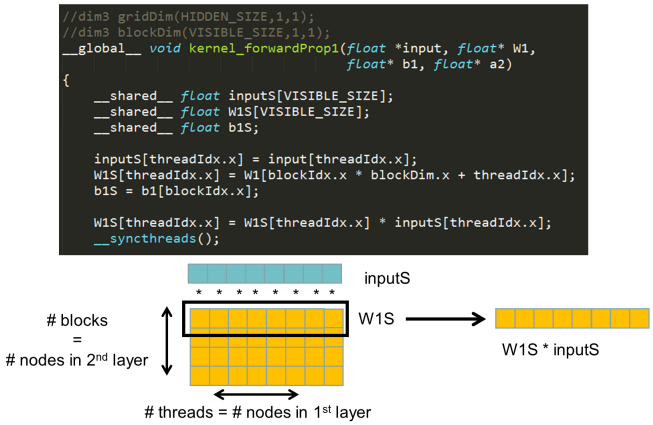

Figure 11: Initial loading and storage of temporary product in shared memory for CUDA forward propagation.

The products of inputS and W1S are summed using a vector reduction method that keeps the memory contiguous and reduces overall divergence (figure 12). By keeping fewer warps doing more work-blocks each block is retired more quickly. This method also prevents bank conflicts from keeping the partial sums stored at regular intervals (rather than contiguously).

```
133     for(int offset = blockDim.x / 2; offset > 0; offset >>= 1)
134     {
135         if(threadIdx.x < offset)
136         {
137             // add a partial sum upstream to our own
138             W1S[threadIdx.x] += W1S[threadIdx.x + offset];
139         }
140
141         // wait until all threads in the block have
142         // updated their partial sums
143         __syncthreads();
144     }
```

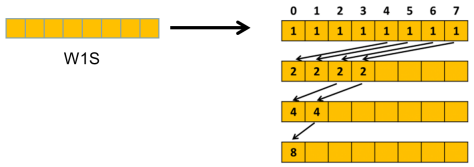

Figure 12: Vector reduction that reduces bank conflicts, divergences, and keeps warps fully active for as long as possible.

Finally, thread 0 is used to sum the overall results and calculate the result of the activation function (figure 13). Although the code and diagram shown below is for forward propagation, the kernels for back propagation follow the same overall form with more complex functions (than simple addition) placed into the thread 0 if statement.

```
//only the 0 thread of each block does this.
if( threadIdx.x == 0 )
{
    //add the bias vector value
    W1S[0] += b1S;

    //apply sigma function and transfer back to global mem
    a2[blockIdx.x] = float(1/(1+exp(-W1S[0])));

    //perform other operations here
}
```

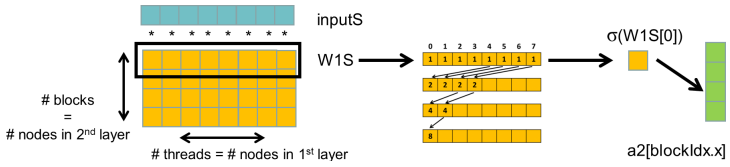

Figure 13: Vector reduction reducing bank conflicts, divergences, and partially empty warps.

We also tried another kernel method (contained in sparseAutoenconder_CUDA_NAIVE.cu) that used an atomic add on data stored in global memory. Instead of breaking up the W1 matrix into rows, each GPU block can map to an entire W1 matrix as well as W1-row-length-number of input vectors (figure 14). The intuition here is that each W1 matrix will have to take the dot product of every row of W1 with every input vector. It is not important in what order the rows of W1 are dotted with the inputs. For example it is not important when the first row of W1 is dotted with the first input vector, just that they are dotted and then the result of this dot is used to "contribute" to the first variable of the $\hat{\rho}$ vector. So each block receives W1 an array of multiple input vectors and computes an offset. Each block uses this information to compute a dot and place the dot in the appropriate temporary vector that will eventually be used in the computation of $\hat{\rho}$.

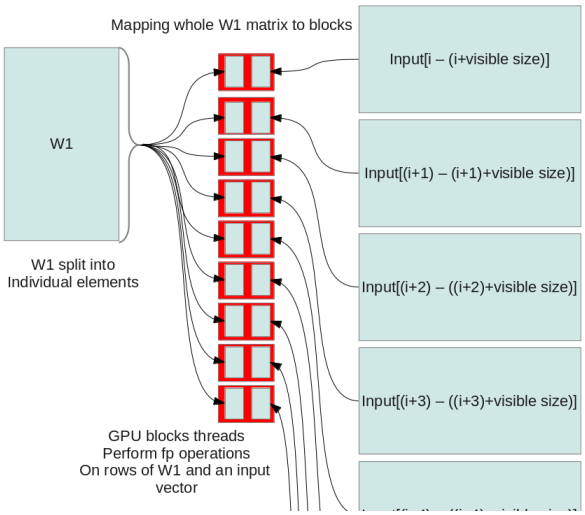

Figure 14: Example of GPU kernel memory mapping of entire W matrix for forward propagation.

This method has the benefit of better load balancing when the size of the visible layer (the number of columns in each row of W1) is small. Memory is also better coalesced because instead of loading each input vector separately, a large contiguous array of this data can be loaded in at once. As the size of the visible layer increases and the memory bus transfer is maximized this effect will decrease, but for neural networks with a small number of nodes in the visible layer the savings from memory coalescence could be significant.

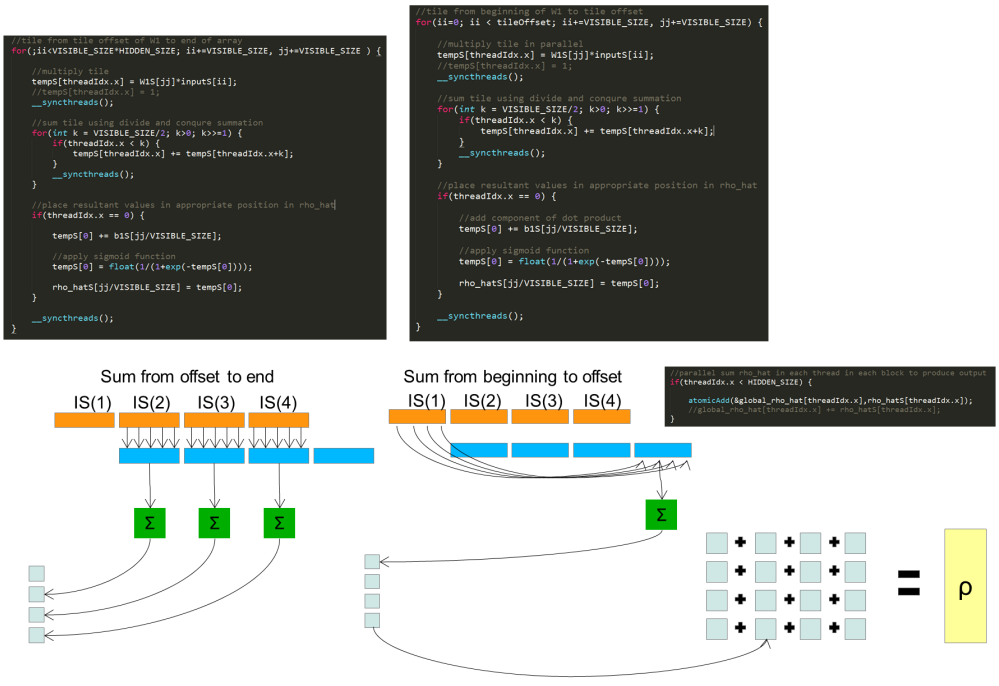

Figure 15: GPU kernel using atomic add.

However, each dot product of this method requires a final atomic operation (each dot's "contribution" to the $\hat{\rho}$ vector) (figure 15). Unfortunately, the time cost of these atomic operations slows the function to such a point that even neural networks with with a small visible layer show no speed up. Despite these

shortcomings we detail this algorithm here because, though it was unable to speed up our code, similar problems that do not require atomic operations would run significantly faster.

The previous kernel code described so far is able to perform a wave of neural network propagation for a very large number of nodes in both the visible and hidden layers on a neural network that is completely connected between each layer. However, this implementation is not infinitely scalable. To provide code that is infinitely scalable it is necessary to break the forwards and back propagation kernel functions as well as the sparsity enforcement kernel into chunks.

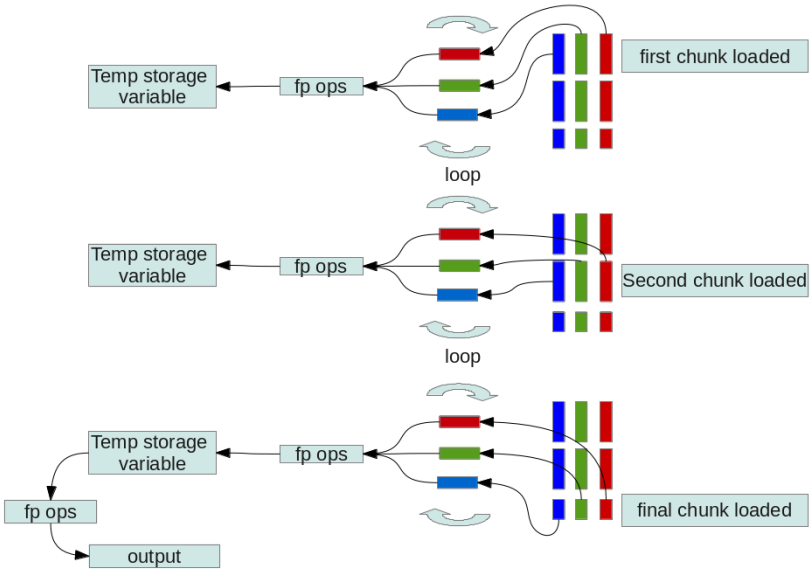

Figure 16: Illustration of the kernel breaking up inputs into "chunks" that fit in GPU shared memory.

Chunks here are defined as a maximum block dimension such that allocating shared memory to each thread based on this dimension does not exhaust the shared memory of the SMs within that block (figure 16). This definition is used because the blocks are allocated such that each thread accesses only one element of each of the chunked blocks. Because many of the operations within the kernels involve repeatedly applying a sequence of floating point operations to two vectors chunking is accomplished by sub-dividing these long vectors into smaller chunks and performing the appropriate fp operations on these smaller ops. As well, often a summation is performed in the functions. This is accomplished by summing the individual chunks and keeping a running sum of all previous chunk sums then adding the running sum. Then once a new sum has been computed in each of the threads that is added to the running sum.

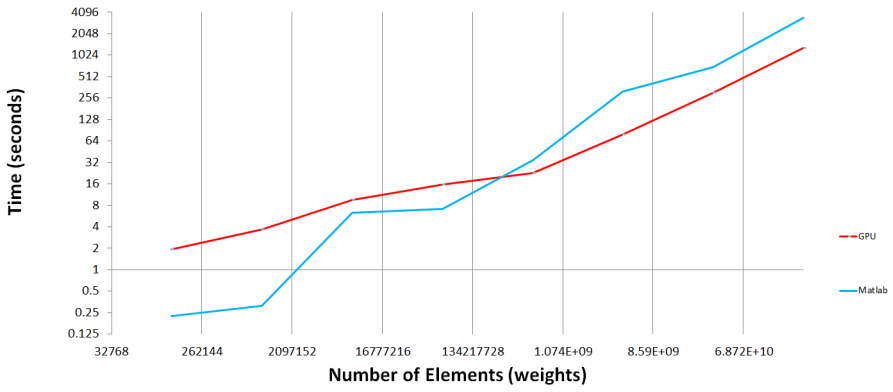

Figure 17: Log log plot of runtime of CUDA implementation of sparse autoencoder.

The CUDA implementation greatly reduced runtime and beat the performance of every other implementation including Matlab. However, there is a break even point in terms of runtime and neural network size that is discussed in the results section.

Code Variant	**64x32**	**128x64**	**256x128**	**512x256**	**1024x512**	**2048x1024**	**4096x2048**	**8192x4096**
Matlab	0.22	0.31	6.34	7.18	34.30	317.16	692.25	3404.98
GPU	1.93	3.66	9.54	15.57	22.94	78.28	306.71	1287.14

Table 5: Timing results in seconds depending on autoencoder dimensions for the CUDA implementation of the sparse autoencoder. The code variant number corresponds to flag.

2 Results and Conclusions

The CUDA implementation of the sparse autoencoder provided a speedup of at least 1.5x over all serial implementations run on network dimensions greater than or equal to 1024 input nodes by 512 hidden nodes (figures 18, 19, 20, and 21; tables 6 and 7). When run on smaller network sizes, the CUDA implementation generally performed worse except when compared with the naive serial implementation compiled with the O0 flag. A range of break even times was to be expected as the overhead of set up and tear down for the GPU can only be offset by an increase in the overall number of operations being performed by a serial implementation. As the number of operations per propagation increases, the efficiency and utilization of the GPU increases. This is in part because the number of threads per block scales with the size of the weight matrices, the dimensions of which are determined by number of nodes in each layer. This can be observed by comparing each of the first eight columns of table 7 against the last.

Of the serial implementations done in C, 8x loop unrolling showed the best performance both with and without optimization flags (figures 20, 21). Matlab did admirably well over a large range of values and our CUDA implementation yielded only a 2.6x speedup for the largest network size tested (table 7). Loop unrolling and optimization flags showed comparable performance to Matlab as network dimensions increased. Within code variants (e.g., serial_loop_O1 as compared to serial_loop_O2, etc.) optimization flags showed relatively little difference in performance (figures 4, 6, 8, and 10).

Overall, we were surprised at just how well Matlab performed in comparison to our CUDA implementation and generally thought that we would see a greater speed up in comparison to all variants. The fact that our CUDA implementation's speedups range from 2.6 to 59.2x for the largest network dimensions we tested indicates one or both of the following (1) that the propagation algorithms of a sparse neural network are not suited for higher degrees of parallelism or (2) we did not make use of all the optimizations available.

Other potential optimizations we did not have time to implement include use of textured memory for storing data that is highly reused without modification, testing a range of block and grid dimensions to find a load balancing "sweet spot" for each network dimension set, or increasing the number of kernels used while decreasing the number of different operations (steps within each propagation step) in order to fit more memory into shared memory at a given time. We were also surprised by how poorly openMP performed even when run on machines with up to 32 cores. This struck us as odd and we cannot help but think that there is a great deal of room for improvement. It would also be interesting to compare Matlab's integrated GPU functions with our own kernels.

Code Variant	**64x32**	**128x64**	**256x128**	**512x256**	**1024x512**	**2048x1024**	**4096x2048**	**8192x4096**
serial_naive_0	4.13	15.07	58.39	260.96	1045.78	4480.39	18990.84	76136.01
serial_naive_1	1.29	4.14	16.66	65.90	268.30	1321.22	6273.55	25763.80
serial_naive_2	1.20	3.85	15.56	63.41	266.31	1285.16	6242.11	26455.83
serial_naive_3	1.10	3.54	14.33	57.90	239.40	1229.89	5718.97	24152.99
serial_base_0	2.17	8.24	31.94	125.71	501.79	2001.83	8003.89	31994.42
serial_base_1	0.66	2.30	8.51	32.24	129.46	516.00	2048.37	8183.33
serial_base_2	0.55	1.91	6.93	25.85	104.13	418.37	1668.43	6706.02
serial_base_3	0.55	1.90	6.91	25.91	104.24	418.25	1665.20	6717.96
serial_loop_0	0.85	2.89	10.37	41.04	165.37	659.59	2599.62	10441.16
serial_loop_1	0.38	1.12	3.58	13.53	60.22	246.56	977.70	3876.70
serial_loop_2	0.37	1.05	3.32	13.21	58.63	243.90	982.09	3991.94
serial_loop_3	0.36	1.04	3.30	12.61	56.52	233.29	932.28	3829.75
serial_omp_0	4.47	15.86	53.25	189.46	801.07	2851.26	10444.64	38913.18
serial_omp_1	6.61	5.01	18.37	78.55	297.12	1010.91	3649.42	13340.75
serial_omp_2	1.60	3.46	19.17	35.42	142.60	626.54	2063.90	7263.56
serial_omp_3	7.04	6.15	12.70	27.52	117.62	525.72	1972.42	6755.93
Matlab	0.22	0.31	6.34	7.18	34.30	317.16	692.25	3404.98
GPU	1.93	3.66	9.54	15.57	22.94	78.28	306.71	1287.14

Table 6: Timing results in seconds for code variants depending on network dimensions (input nodes x hidden nodes).

Code Variant	**64x32**	**128x64**	**256x128**	**512x256**	**1024x512**	**2048x1024**	**4096x2048**	**8192x4096**
serial_naive_0	2.1	4.1	6.1	16.8	45.6	57.2	61.9	59.2
serial_naive_1	0.7	1.1	1.7	4.2	11.7	16.9	20.5	20.0
serial_naive_2	0.6	1.1	1.6	4.1	11.6	16.4	20.4	20.6
serial_naive_3	0.6	1.0	1.5	3.7	10.4	15.7	18.6	18.8
serial_base_0	1.1	2.3	3.3	8.1	21.9	25.6	26.1	24.9
serial_base_1	0.3	0.6	0.9	2.1	5.6	6.6	6.7	6.4
serial_base_2	0.3	0.5	0.7	1.7	4.5	5.3	5.4	5.2
serial_base_3	0.3	0.5	0.7	1.7	4.5	5.3	5.4	5.2
serial_loop_0	0.4	0.8	1.1	2.6	7.2	8.4	8.5	8.1
serial_loop_1	0.2	0.3	0.4	0.9	2.6	3.1	3.2	3.0
serial_loop_2	0.2	0.3	0.3	0.8	2.6	3.1	3.2	3.1
serial_loop_3	0.2	0.3	0.3	0.8	2.5	3.0	3.0	3.0
serial_omp_0	2.3	4.3	5.6	12.2	34.9	36.4	34.1	30.2
serial_omp_1	3.4	1.4	1.9	5.0	13.0	12.9	11.9	10.4
serial_omp_2	0.8	0.9	2.0	2.3	6.2	8.0	6.7	5.6
serial_omp_3	3.6	1.7	1.3	1.8	5.1	6.7	6.4	5.2
Matlab	0.1	0.1	0.7	0.5	1.5	4.1	2.3	2.6
GPU	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 7: Speedup results of CUDA over each implementation of the sparse autoencoder neural network at different network dimensions (input nodes x hidden nodes).

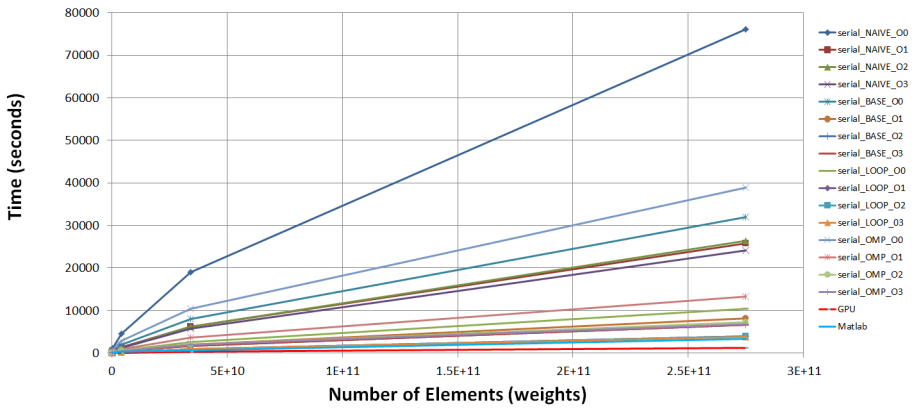

Figure 18: Runtime of all implementations of sparse autoencoder.

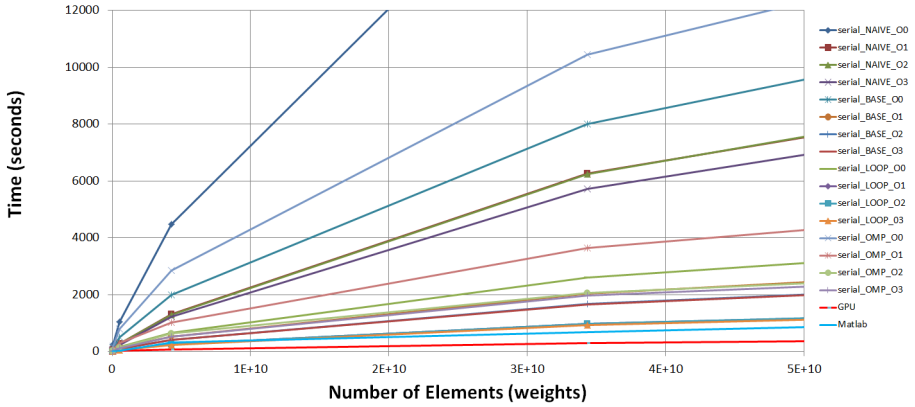

Figure 19: Runtime of all implementations of sparse autoencoder zoomed in.

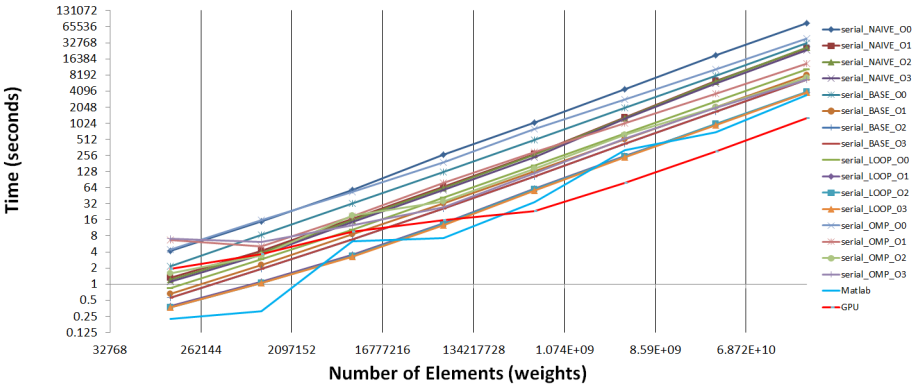

Figure 20: Log log plot of runtime of CUDA implementation of sparse autoencoder.

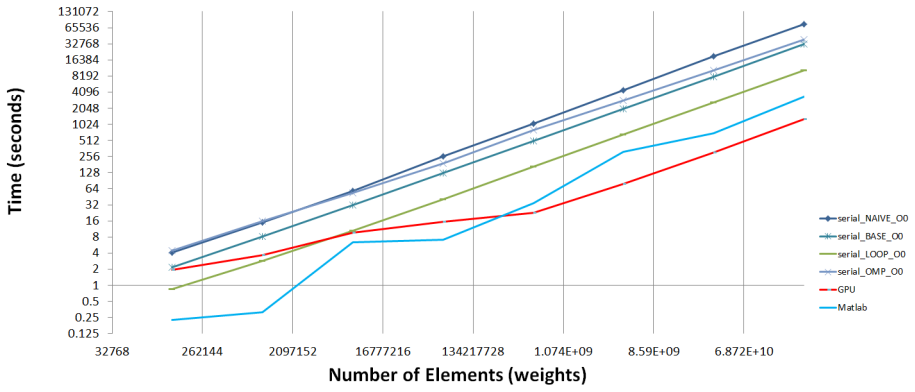

Figure 21: Log log plot of runtime of GPU, Matlab, and -O0 variants of sparse autoencoder.

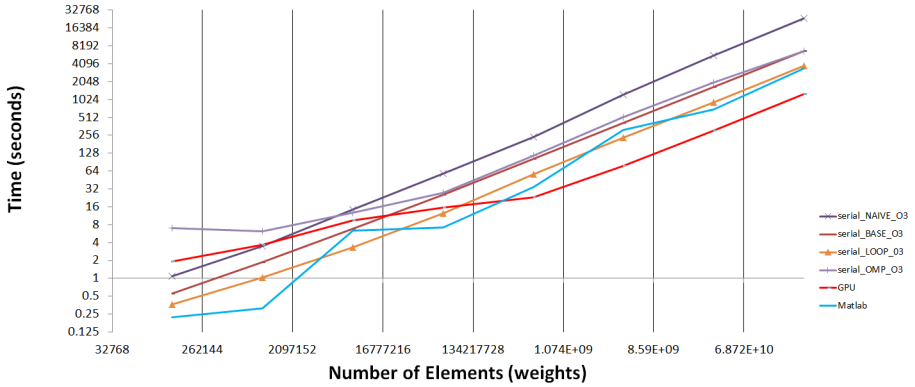

Figure 22: Log log plot of runtime of GPU, Matlab, and -O0 variants of sparse autoencoder.

3 Included files

The following files allow for the running of the C implementation of the sparse autoencoder. Each is a stand alone file with compilation instructions in the comments at the top of the file. Also included is a folder with Matlab code for running the sparse autoencoder. Instructions on running the Matlab code are contained in the train.m file comments but the file names are not enumerated here. The csv files included are calibration files that can be read into all of the C code. Place them in the same folder or specify the correct folder path and file name in the define statements at the top of the code.

- sparseAutoencoder_SERIAL_NAIVE.c
- sparseAutoencoder_SERIAL_BASELINE.c
- sparseAutoencoder_SERIAL_LOOP.c
- sparseAutoencoder_SERIAL_OMP.c
- sparseAutoencoder_CUDA_NAIVE.cu
- sparseAutoencoder_CUDA_CHUNK.cu
- W1.csv

- W2.csv
- c_patches.csv
- awelles_klausd_ec527_final_project_analysis.xlsx

4 References

1. Dr. Andrew Ng's assignment for the sparse autoencoder is available here: http://www.stanford.edu/class/archive/cs/cs294a/cs294a.1104/sparseAutoencoder.pdf