Sparse DNN Results with the MATLAB interface to SuiteSparse:GraphBLAS

Tim Davis

Sept 2, 2019

The tables below report the results for the 12 sparse deep neural network problems. Problems 1-3 use 1024 neurons, 4-6 use 4,096 neurons, 7-9 use 16K neurons, and the last use 64K neurons. Each group of three uses 120, 480, and 1920 layers, respectively.

1 The MATLAB code

The MATLAB code (with GraphBLAS) is very simple, even simpler than the MATLAB reference implentation. All MATLAB variables are GraphBLAS `gb` objects. In this case, they represent sparse matrices with the `GrB_FP32` floating point type, stored in CSR format (by row).

```
function Y = dnn_gb (W, bias, Y0)
Y = Y0 ;
for i=1:length(W)
    Y = gb.select (’>0’, gb.mxm (’+.+’, Y * W {i}, bias {i})) ;
    M = Y > 32 ;
    if (nnz (M) > 0)
        Y (M) = 32 ;
    end
end
```

For comparison, here is the MATLAB reference implmentation at http://graphchallenge.org. It is about 60x to 70x slower than the two methods using GraphBLAS. Applying the bias is more complex than the GraphBLAS code:

```
function Y = dnn_matlab (W, bias, Y0)
Y = Y0 ;
for i=1:length(W)
    % Propagate through layer.
    Z = Y * W {i} ;
    % Apply bias to non-zero entries.
    Y = Z + (double(logical(Z)) .* bias {i}) ;
    % Threshold negative values.
    Y (Y < 0) = 0 ;
    % Threshold maximum values.
    Y (Y > 32) = 32 ;
end
```

The code to convert from MATLAB sparse matrices to GraphBLAS `gb` objects is shown below. This time is not included since the problem could have been read in as GraphBLAS matrices to begin with. In any case, the conversion time is trivial. Problem 12 is converted from MATLAB to GraphBLAS in 30.4 seconds, or about 1% of the time to solve the DNN with 40 threads. Problem 1 is converted in 0.3 seconds.

```
function [W, bias, Y0] = dnn_mat2gb (W, bias, Y0)
n = size (Y0, 2) ;
Y0 = gb (Y0, 'single') ;
for i=1:length(W)
    W {i} = gb (W {i}, 'single') ;
    bias {i} = gb.build (1:n, 1:n, bias {i}, n, n, '+', 'single') ;
end
```

2 The C code

The C version in LAGraph, minus error checking and with a few other trivial simplifications, is shown below. Is straight-forward but more complex than either the pure MATLAB version (`dnn_matlab.m`) or the MATLAB+GraphBLAS version (`dnn_gb.m`).

```c
#include "LAGraph.h"
void ymax32 (float *z, const float *x)
{
    (*z) = fminf ((*x), (float) 32.0) ;
}
GrB_Info LAGraph_dnn    // returns GrB_SUCCESS if successful
(
    GrB_Matrix *Yhandle,// Y, created on output
    GrB_Matrix *W,      // W [0..nlayers-1], each nneurons-by-nneurons
    GrB_Matrix *Bias,   // Bias [0..nlayers-1], diagonal nneurons-by-nneurons
    int nlayers,        // # of layers
    GrB_Matrix Y0       // input features: nfeatures-by-nneurons
)
{
    GrB_Matrix Y = NULL, M = NULL ;
    GrB_Index nfeatures, nneurons ;
    GrB_Matrix_nrows (&nfeatures, Y0) ;
    GrB_Matrix_ncols (&nneurons,  Y0) ;
    GrB_Matrix_new (&Y, type, nfeatures, nneurons) ;
    GrB_Matrix_new (&M, GrB_BOOL, nfeatures, nneurons) ;
    GrB_UnaryOp Ymax32 ;
    GrB_UnaryOp_new (&Ymax32, ymax32, GrB_FP32, GrB_FP32) ;
    for (int layer = 0 ; layer < nlayers ; layer++)
    {
        // Y = Y * W [layer], using the conventional PLUS_TIMES semiring
        GrB_mxm (Y, NULL, NULL, GxB__PLUS_TIMES_FP32,
            ((layer == 0) ? Y0 : Y), W [layer], NULL) ;
        // Y = Y * Bias [layer], using the PLUS_PLUS semiring.
        GrB_mxm (Y, NULL, NULL, GxB__PLUS_PLUS_FP32, Y, Bias [layer], NULL) ;
        // delete entries from Y: keep only those entries greater than zero
        GxB_select (Y, NULL, NULL, GxB_GT_ZERO, Y, NULL, NULL) ;
        // threshold maximum values: Y (Y > 32) = 32
        GrB_apply (Y, NULL, NULL, Ymax32, Y, NULL) ;
    }
    GrB_free (&M) ;
    (*Yhandle) = Y ;
    return (GrB_SUCCESS) ;
}
```

3 Run time results

Run time in seconds on an Intel Xeon E5-2698v4 @ 2.2GHz, with 20 hardware cores and 256GB of RAM, using the GCC 5.4.0 compiler, and Ubuntu 16.04. Note that the icc compiler generates faster code, but it's not compatible with MATLAB, so the gcc compiler on the system was used instead. MATLAB R2018a was used.

The fastest time is shown in bold, for one and 40 threads. Lower is better.

	one thread			40 threads		
Prob	MATLAB	LAGraph	M/L	MATLAB	LAGraph	M/L
1	**24**	24	0.97	3	**2**	1.17
2	**68**	68	0.99	9	**5**	2.07
3	**242**	243	1.00	34	**16**	2.09
4	**98**	108	0.90	10	**9**	1.07
5	**293**	330	0.89	**31**	31	1.00
6	**1076**	1222	0.88	**117**	118	0.99
7	766	**741**	1.03	58	**51**	1.15
8	2684	**2552**	1.05	201	**175**	1.15
9	10381	**9783**	1.06	783	**690**	1.13
10	**3777**	4536	0.83	254	**245**	1.04
11	**13817**	16447	0.84	971	**926**	1.05
12	**54701**	65492	0.84	3829	**3743**	1.02

4 Rate results

The rate is equal to the number of edges in the DNN, times the number of features (60,000 for all cases), divided by the run time. Rate is reported in terms of billions of edges/sec. Best rate shown in bold; higher is better.

	one thread			40 threads		
Prob	MATLAB	LAGraph	M/L	MATLAB	LAGraph	M/L
1	**10.0**	9.7	1.03	85.8	**100.4**	0.85
2	**14.0**	13.8	1.01	101.3	**209.2**	0.48
3	**15.6**	15.5	1.00	110.1	**230.2**	0.48
4	**9.7**	8.7	1.11	94.6	**101.1**	0.93
5	**12.9**	11.4	1.13	**123.3**	122.9	1.00
6	**14.0**	12.4	1.14	**129.1**	128.4	1.01
7	4.9	**5.1**	0.97	64.6	**74.0**	0.87
8	5.6	**5.9**	0.95	75.2	**86.3**	0.87
9	5.8	**6.2**	0.94	77.2	**87.5**	0.88
10	**4.0**	3.3	1.20	59.4	**61.5**	0.96
11	**4.4**	3.7	1.19	62.2	**65.2**	0.95
12	**4.4**	3.7	1.20	63.1	**64.5**	0.98

5 Comparison

When using 40 threads, the performance of the two methods is almost identical, except for problems 2 and 3, where LAGraph is about twice as fast as the MATLAB `dnn_gb.m`. The two codes differ in how the max threshold of 32 is implemented. The MATLAB interface doesn't allow for user-defined operators, so a mask `M` is used for the `dnn_gb` function. This actually seems to be faster in many cases when using a single thread, as compared to the method used in `LAGraph_dnn`. The latter uses a user-defined operator, `ymax32` and `GrB_apply`. With 40 threads, the `GrB_apply` is faster.

It appears that very little is lost, if any, in the MATLAB interface. To be certain of this, the `LAGraph_dnn.c` function would need to be modified to use the masked assignment method used in `dnn_gb.m`. However, each function was written using the most natural approach available, and since the MATLAB interface does not allow for user-defined operators, it was most natural to write the max32 threshold as masked assigment. In that sense, this is a fair comparison between MATLAB+GraphBLAS and LAGRAPH+GraphBLAS.