ME 766 Course Project

Implementation of SELL-C-Sigma sparse Matrix Vector Multiplication Kernel in CUDA

Vivek Bangera
Mahak Arora

System Specs:

The system used for the experiment have been recorded in Syspec.txt

The CUDA device details are mentioned in CUDAdevice.txt

Code listing

matrix.cu	-	Compressed Row Storage Format
pjds.cu	-	Padded JDS
SELL_1.cu	-	SELL-C-Sigma

Introduction:

The project aims at implementing a SELL-C-Sigma sparse Matrix Vector Multiplication (spMVM) kernel to efficiently multiply very large sparse matrices.

The idea was influenced by [1].

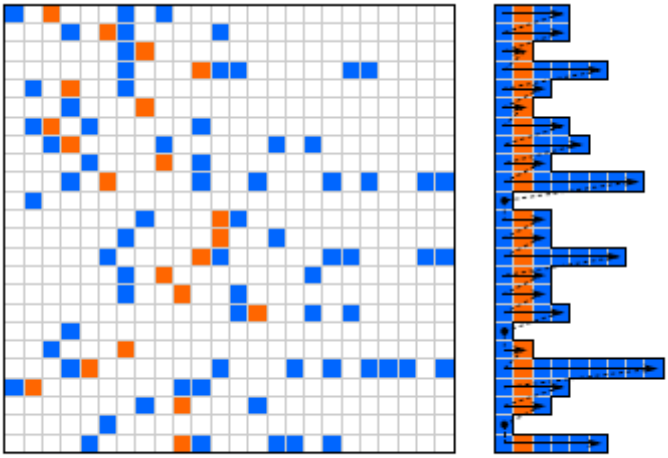
Let us dive a bit into the Sparse Matrix Container problems and discuss how the idea for the SELL-C-Sigma format was inspired.

Sparse Matrices

Sparse matrices are usually large matrices which have very few non-zero elements. These elements may at times be adjacent or at times may be completely randomly placed. Now, to store such a matrix in its entirety seems like a waste of memory (although we have plenty of it now) and a waste of computational power as well as we end up performing a lot of zero multiplications as well. This is counterproductive and very inefficient. The idea then is to store only the non-zero elements and then multiply these non-zero elements with the matching element in the vector. Thus, the first sparse matrix storage format we discuss is Compressed Row Storage format (CRS).

Compressed Row Storage (CRS)

The Compressed Row Storage format uses three one-dimensional arrays to store all the information of the non zero values of a sparse matrix. The three arrays are: row_ptr, cols and val. The array val[] stores the actual non-zero values of the matrix. The array col_ind[] stores the column number of the nonzero elements of each row. These values are used to identify the elements of the vector with which we multiply the non-zero elements of the matrix. The array row_ptr[] is used to store the address of the offset of every row. This is illustrated by the example below.



(a) Source matrix (b) CRS

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}.$$

val	10	-2	3	9	3	7	8	7	3 ··· 9	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	1 ··· 5	6	2	5	6

row_ptr	1	3	6	9	13	17	20

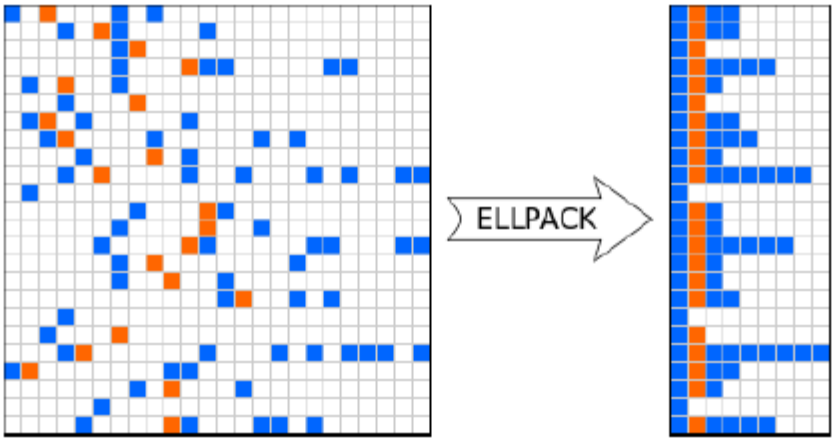
[2]

The col_ind stores the column index of the non-zero values of the sparse matrix and row_ptr stores the offset of the col_ind of start of the each row.

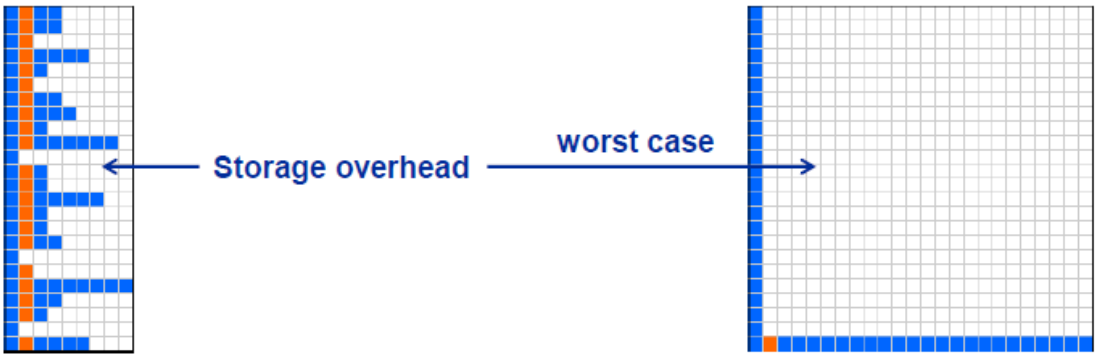
The problem with the CRS format is that it is not SIMD friendly. There is a lot of scope of parallelizing the code but most of the values are not fetched and stored in caches at the same time. The main kernel of the CRS spMVM when observed has significant scope for vectorization (which is shown by loop unrolling in [1]).

ELLPACK

To make the spMVM more SIMD friendly the ELLPACK format was invented. At a marginal cost for storage, the parallelism of spMVM can be exploited very well. This is illustrated by the example below.

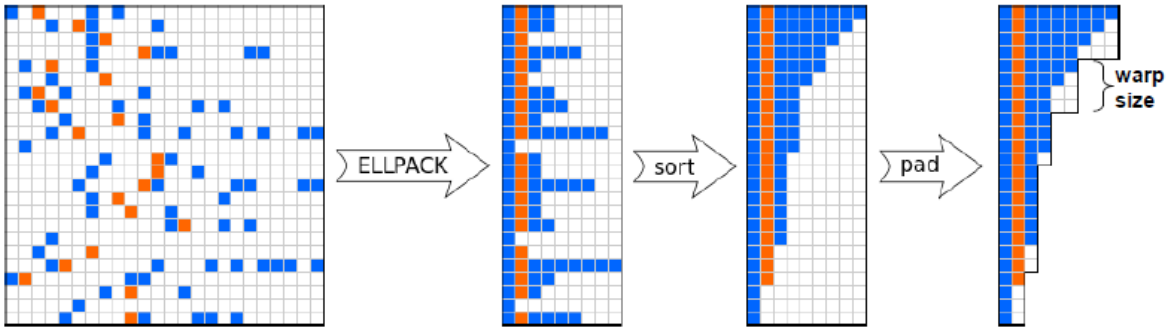


This format needs a lot of padding which increases the storage overhead. In the worst case we could end up padding (N-1)*(N-1) for 2N elements



pJDS (Padded Jagged Diagonal Storage)

In padded Jagged Diagonal Storage format the rows of ELLPACK format are sorted according to the highest number of non-zero values and then padding is done in chunks. A chunk of rows gets padded to the same column width according to the column width of the top row of that chunk. This reduces the overall padding to a large extent keeping the scope of vectorization intact.



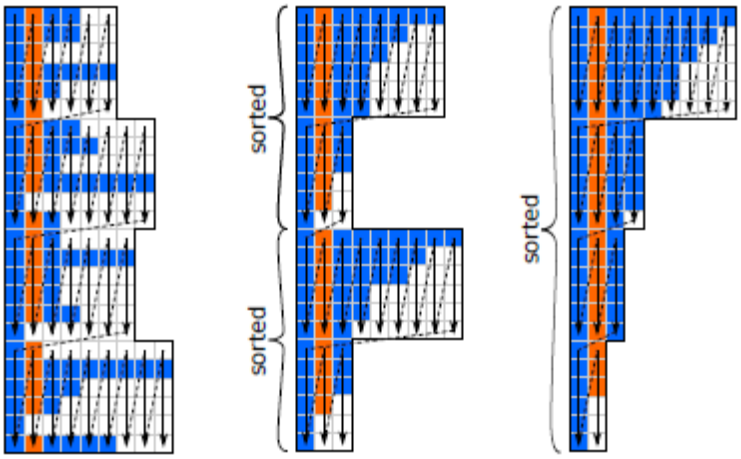
SELL-C-Sigma

The Sliced ELLPACK is a more constrained way of implementing the ELLPACK. Here we wish to gain the performance improvements guaranteed by ELLPACK but still reducing the padding done by ELLPACK.

The SELL-C-Sigma has been implemented by us as follows.

1. Store all the non-zero elements in a 2D matrix.
2. In another 2D matrix, store all the column values of the non-zero elements of the matrix.
3. Divide the N rows of the matrix in blocks of sigma each (i.e. sigma rows in each block).
4. Sort the rows in each block in descending order i.e. the largest rows on the top of the block and the smallest on the bottom.
5. Each block of sigma is then further divided into blocks of size ‘c’.
6. The 2D matrix of nonzero values that we have generated is then flattened out and stored in column major format (both, values and cols)
7. The array cs[] stores the ‘c’ offset i.e. the offset of every c block.
8. The array cols[] stores column information.

The flattened arrays are then passed to the GPU to perform the multiplication.



(a) SELL-6-1, $\beta = 0.51$ (b) SELL-6-12, $\beta = 0.66$ (c) SELL-6-24, $\beta = 0.84$

Sparse Matrices for the experiment.

The sparse matrices for our experiment were generated by us using a python script. We used the randint() function from the random library of python to help generate the matrices. The script is present in the repository as randommat.py. Roughly less than 5% of the matrix has non-zero values while the remaining are all zeros.

Results

The results we have obtained for the multiplication kernels have been noted in the table below.

N	SELL-2-4	SELL -10-20	SELL-100-200	CRS	pJDS
100	0.063232	0.058208	0.052	0.016288	0.062304
200	0.075136	0.064960	0.061	0.03936	0.075424
500	0.162432	0.106240	0.082080	0.20624	0.171488
1000	0.451072	0.267200	0.151968	0.810176	0.502496
5000	10.88029	5.491264	3.098976	19.71008	13.62064
10000	56.26208	21.322111	14.800960	78.33757	67.99927

N is the dimension of the sparse matrix (N*N) that we used.

All times in the table above are in milliseconds

As we can observe from the table, the SELL-C-Sigma format clearly outperforms the other 2 formats by quite some margin. The points to be noted though are:

1. We have only computed the actual kernel multiplication time and not the pre-processing time required to set up the matrix for the multiplication
2. We have not considered data transfer time from the host to the CUDA device and vice versa as Bus speeds may be slow and may not be able to truly reflect the computing performance of the GPU.

The overheads incurred in doing the SELL-C-Sigma format are:

1. Sorting the rows in each block of size sigma.
2. Flattening each 2D array to form respective 1D arrays

References

[1] A UNIFIED SPARSE MATRIX DATA FORMATFOR EFFICIENT GENERAL SPARSE MATRIX-VECTORMULTIPLY ON MODERN PROCESSORS WITH WIDE SIMD UNITS byMORITZ KREUTZER_, GEORG HAGER_, GERHARD WELLEIN_, HOLGER FEHSKEy,
AND ALAN R. BISHOPz

[2] http://netlib.org/linalg/html_templates/node91.html#SECTION00931100000000000000