Parallel Algorithm for Quasi-Band Matrix-Matrix Multiplication

Dharma Teja Vooturi[1] and Kishore Kothapalli[2]

International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India, 500032.
[1]dharmateja.vooturi@research.iiit.ac.in, [2]kkishore@iiit.ac.in

Abstract. Sparse matrices arise in many practical scenarios. As a result, support for efficient operations such as multiplication of sparse matrices (`spmm`) is considered to be an important research area. Often, sparse matrices also exhibit particular characteristics that can be used towards better parallel algorithmics. In this paper, we focus on quasi-band sparse matrices that have a large majority of the non-zeros along the diagonals. We design and implement an efficient algorithm for multiplying two such matrices on a many-core architecture such as a GPU.
Our implementation outperforms the corresponding library implementation by a factor of 2x on average over a wide variety of quasi-band matrices from standard datasets. We analyze our performance over synthetic quasi-band matrices.

Keywords: band · quasi-band · spmm · real-world · synthetic

1 Introduction

Multiplying two sparse matrices, denoted `spmm`, is an important and challenging problem in parallel computing with applications to a wide variety of disciplines including climate modeling, computational fluid dynamics, and molecular dynamics [10]. Due to the importance of `spmm`, a number of works aimed at efficient algorithms and their implementations on a variety of architectures are reported in the literature [2, 6, 7].

A current trend in parallel algorithm engineering is to focus on customizing algorithms based on input characteristics. Such a customization allows the algorithms to benefit from the properties of the input. Recent examples include finding the strongly connected components of real-world graphs by Hong et al. [4], mapping graph traversals to a CPU+GPU heterogeneous platform by Gharibieh et al. [3], sparse matrix-vector multiplication and matrix-matrix multiplication of scale free matrices by Indarapu et al. [5] and Ramamoorthy et al. [7].

In this context, we note that applications such as aerodynamics and computational fluid dynamics [10] produce sparse matrices called as *quasi-band* matrices that exhibit a near-diagonal nature of sparsity. As noted by Yang et al. [11], many sparse matrices also can be reordered or divided into a near-diagonal form.

In this paper, we focus on quasi-band matrices and design a GPU algorithm to multiply two such matrices. Our work extends the work of Yang et al. [11] who design a GPU algorithm for multiplying a sparse quasi-band matrix with a dense vector. Our algorithm starts by separating the input quasi-band sparse matrices into a diagonal part and the rest as a sparse part. Once such a separation is achieved, we introduce specific optimizations to perform the four multiplications: the diagonal/sparse part with the diagonal/sparse part.

Our main technical contributions can be summarized as follows.

– We propose an algorithm (see Section 3) for multiplying two sparse quasi-band matrices. Our algorithm identifies, to a reasonable extent, the indices of the output matrix that will have nonzero entries and uses this information to manage the space required and an estimate of the work required.
– An implementation of our algorithm on an Nvidia K40 GPU achieves a speedup of 5x and 2x on average over a collection of band and quasi-band matrices, respectively, taken from the University of Florida dataset [10]. (See Section 4).
– We also perform experiments on synthetic quasi-band matrices to understand the effect of the nature of the matrix on the speedup. (See Section 4).

2 Preliminaries

We start with a few definitions. If all the nonzero elements in a matrix are in a single diagonal then that matrix is called a *uni-diagonal* matrix. If there exists a diagonal index pair (i, j) such that $i \leq j$ and all the non-zero elements of a matrix are present between diagonals d_i (leftOffset), d_j (rightOffset) then such a matrix is said to be a *uni-band* matrix with a bandwidth of $(j - i + 1)$. If multiple such disjoint pairs of indices exist then it is called a *multi-band matrix.* Uni-band and multi-band matrices are commonly referred to as *band* matrices. A band matrix along with some non-zero elements in the non-band diagonals is called a *quasi-band* matrix. We also use the terms defined in Table 1 that indicate some of the properties of quasi-band matrices.

Table 1. Glossary of Terms

Term	Description
nnz	Number of nonzero elements in a matrix.
nnzPercentage	Percentage of NNZ to all elements in matrix.
bandPercentage	Percentage of NNZ in band part to NNZ in matrix.
bandOccupancy	Percentage of NNZ in band part to all elements in band part.
bandCount	Number of bands present in the matrix.
diagonalCount	Total number of diagonals across all the bands in the matrix.

2.1 Matrix Representation

In our work, we make use of several representations to store sparse matrices. These are described below in brief.

DIA Format: Each diagonal in the matrix with at least one nonzero is stored as a column array of length equal to the number of rows in the matrix. Diagonals are numbered starting with zero for the principal diagonal and -1 and +1 for the diagonals to the left and right of the principal diagonal, and so on. These numbers, called as *diagonal offsets*, are stored in a separate array called offsetArray.

COOSR (COOSC) Format: These can be thought of as a mix of the COO and the CSR (CSC) formats reported in [1]. Each non-zero element in a matrix can be mapped to a triplet (value, row, column). In this format we store three arrays *data, rowIndex and colIndex* each of length nnz(number of non-zero elements in the matrix). Elements are ordered row (`resp.` column) wise in the arrays. An array rowPtr (colPtr) of length $[rows + 1]([columns + 1])$ is also stored. An index i in rowPtr (colPtr) array maps to the index value of first element of row (column) i in *value* array.

3 Our Algorithm

One of the prime difficulties of sparse matrix multiplication include the possible lack of any relation between the nature and degree of sparsity of the input matrices and their product matrix. On multi- and many-core architectures, other difficulties such as load imbalance imply that efficient sparse matrix multiplication is often challenging. One way of addressing this difficulty is to look for particular properties of the input matrices and their impact on the product matrix. To this end, focusing on quasi-band matrices, we start with the following observations.

Observation 1 *Multiplying two uni-diagonal matrices A and B with diagonal offsets a_o and b_o respectively results in another uni-diagonal matrix with diagonal offset $(a_o + b_o)$.*

Observation 2 *When a matrix A having a single non-zero element a_{ij} is multiplied with a uni-band matrix B defined by diagonal offsets $(left_o, right_o)$, in the product matrix $C = A \times B$, only elements in row i with column indices between $[max(0, j + left_o) : min(B_{cols} - 1, j + right_o)]$ are effected.*

To make use of the above observations, in the matrix product $C = A \times B$, we start by partitioning quasi-band matrices A and B into their band and non-band components denoted A_{bands} and A_{sparse}, B_{bands} and B_{sparse} respectively. We then compute four matrix products $C_{\text{bb}} = A_{\text{bands}} \times B_{\text{bands}}$, $C_{\text{sb}} = A_{\text{sparse}} \times B_{\text{bands}}$, $C_{\text{bs}} = A_{\text{bands}} \times B_{\text{sparse}}$ and $C_{\text{ss}} = A_{\text{sparse}} \times B_{\text{sparse}}$. The matrix C

is the result of adding the matrices $C_{\mathrm{bb}}, C_{\mathrm{sb}}, C_{\mathrm{bs}}$, and C_{SS}. An outline of the above approach is shown in Algorithm 1.

The computation of each of the four matrix products along with how to achieve the required partitioning of A and B is described in subsections 3.1–3.3. For computing C_{SS} we use the `spmm` kernel from NVIDIA's `cusparse` library [8].

Algorithm 1 Algorithm for multiplying quasi-band matrices A and B into C.

```
1: $A = A_{bands} + A_{sparse}$ ($A_{bands}$ is $A^1_{band} + A^2_{band} + ..... + A^m_{band}$)
2: $B = B_{bands} + B_{sparse}$ ($B_{bands}$ is $B^1_{band} + B^2_{band} + ..... + B^n_{band}$)
3: $C_{bb} = A_{bands} \times B_{bands}$
4: $C_{sb} = A_{sparse} \times B_{bands}$
5: $C_{bs} = A_{bands} \times B_{sparse}$
6: $C_{ss} = A_{sparse} \times B_{sparse}$
7: $C = MERGE(C_{bb}, C_{sb}, C_{bs}, C_{ss})$
```

3.1 Matrix Partition

We first calculate diaOccupancy for each diagonal in the matrix. `DiaOccupancy` for a diagonal is defined as the percentage of non-zero elements in that diagonal to the rows of the matrix. We filter all the diagonals which have diaOccupancy greater than a threshold *diaOcc* to a set S. Diagonals in set S are potential pivots of bands. We start by identifying the band surrounding the diagonal with the largest diaOccupancy in S. This is done by including this diagonal and the diagonals to its left and right so long as their bandOccupancy is more than a threshold *bandOcc*. These diagonals are then recognized as one band and the diagonals from S which intersect with the band are removed from S. The process is repeated to locate other bands of diagonals until S is empty. The diagonals that are chosen as part of some band are arranged in the DIA format. The left over elements are arranged in the COOSR format for A_{sparse} and COOSC format for B_{sparse}. As threshold *diaOcc* is used for finding pivot diagonals of bands, we keep the value at 50%. We also set *bandOcc* to be 40%, to ensure that bands are nearly half dense.

3.2 Multiplying A_{bands} with B_{bands}

In this section we present optimizations that can be applied when we multiply two band matrices. We perform the multiplication in two steps: a preprocessing step and a computation step.

In the preprocessing step, we use Observation 1 to allocate the correct space for the matrix C_{bb} which we store in the DIA format. Each diagonal from A_{bands} having offset a_o can then be multiplied with each diagonal in B_{bands} having offset b_o in parallel. Furthermore each row element r in the resultant diagonal can be updated atomically in parallel using equation $C_{bb}(a_o+b_o, r) += A_{bands}(a_o, r) \times B_{bands}(b_o, r + a_o)$ provided diagonal offset $a_o + b_o$ and row index $r + a_o$ are valid . (For a matrix M represented in the DIA format, $M(o, r)$

refers to the element with diagonal offset o and row number r.). An outline of computation step is shown in Algorithm 2.

Algorithm 2 Multiplying A_{bands} with B_{bands}.

```
1: for $a_o$ in $A_{bands}.offsetArray$ do
2:     for $b_o$ in $B_{bands}.offsetArray$ do
3:         for $r$ in $[0 : A_{rows})$ do
4:             if $(-A_{rows} < a_o + b_o < B_{cols})$ and $(0 <= r + a_o < A_{rows})$ then
5:                 $C_{bb}(a_o + b_o, r) = A_{bands}(a_o, r) * B_{bands}(b_o, r + a_o)$
6:             end if
7:         end for
8:     end for
9: end for
```

3.3 Multiplying A_{sparse} with B_{bands}

We now use Observation 2 in this computation. We start with a preprocessing step where we find all the column segments effected by $A_{\text{sparse}} \times B^i_{bands}$ for any i. These segments may overlap with each other. We proceed by merging these segments so that we can then allocate necessary storage for the matrix C_{sb} in the COOSR format.

In the actual computation step, we note from Observation 2 that an element e from A_{sparse} at row r and column c can be multiplied with each diagonal in B_{bands} having offset b_o in parallel. Such a computation effects at most one element in C_{sb} with row r and column $(c + b_o)$. As C_{sb} is stored in the COOSR format, this element needs to be introduced in to row r of C_{sb} by doing binary search on column indices of row r. An outline of computation step is shown in Algo 3. Using the COOSR format instead of CSR format increases the efficiency of our algorithm as we have simultaneous access to the row and the column indices of elements. This can be observed in steps 3 & 4 in Algorithm 3.

The multiplication of A_{bands} with B_{sparse} is similar to that of $A_{\text{sparse}} \times B_{\text{bands}}$. In this case row segments will be effected, instead of column segments. Another change to note is that we store the matrix C_{bs} in the COOSC format.

3.4 Merging

In the merging step, an element e at row r and column c in the matrix C has to be accumulated from corresponding elements in the four sub-products $C_{\text{bb}}, C_{\text{sb}}, C_{\text{bs}}$ and C_{ss}. This is done in four steps. In the first step, for every element with row index r and column index c in the matrix C_{bs} we check if there is a corresponding element in any of the matrices $C_{\text{sb}}, C_{\text{ss}}$, and C_{bb}. If it exists, we consolidate the contribution of the element (r, c) in C_{bs} with one of the other three matrices. In the second step, we consolidate overlapping elements in C_{ss} with elements in matrices C_{sb} and C_{bb}. In the third step, we consolidate

Algorithm 3 Multiplying A_{sparse} with B_{sparse}.

```
1: for $i$ in $[0 : A_{sparse}.NNZ)$ do
2:   for $b_o$ in $B_{bands}.offsetArray$ do
3:     $r = A_{sparse}.rowIndex[i]$
4:     $c = A_{sparse}.colIndex[i] + b_o$
5:     if $0 \leq c < B.cols$ then
6:       index $= SEARCH(c, C_{sb}.colIndex[C_{sb}.rowPtr[r]{:}C_{sb}.rowPtr[r+1]-1])$
7:       $C_{sb}.data[index] += A_{sparse}.data[i] * B_{bands}(b_o, c)$
8:     end if
9:   end for
10: end for
```

overlapping elements in C_{sb} with elements in matrix C_{bb}. Having easy access to both the row and column indices via the COOSR/COOSC formats makes the merge process efficient compared to using CSR/CSC formats.

We now have all the four subproducts that do not have any overlapping elements, but they all are not in the same format. In the fourth step, we convert matrices C_{bs} and C_{bb} into the COOSR format leaving us with four non-overlapping matrices in the COOSR format. It is now easy to combine these four matrices to a single matrix C in the COOSR format. (See also [7].)

Table 2. Properties Of Band Matrices. Letter K stands for a thousand, and letter M stands for a million.

Matrix	Rows	NNZ	Band-Count	Diagonal-Count	Band-Occupancy	Band-Spread
Bai/af23560	24K	484K	5	33	62.5	12.9
Boeing/crystm03	25K	583K	27	39	88.8	689.21
Castrillon/denormal	89K	1156K	5	13	99.75	6.67
Averous/epb1	15K	95K	3	11	58.96	3.77
Norris/fv1	10K	87K	3	9	99.32	4.08
Nasa/nasa2146	2K	72K	3	45	76.63	13.79
Boeing/pcrystk03	25K	1751K	9	99	72.66	76.53
Oberwolfach/windscreen	22K	1482K	9	99	79.36	909.68
Nemeth/nemeth21	10K	1173K	1	169	73.39	0

4 Experimental Results and Analysis

4.1 Datasets

We experiment with real-world band and quasi-band matrices from the University of Florida sparse matrix collection [10]. The matrices we use and some of the properties are shown in Tables 2 and 3. We also experiment with a variety of synthetic datasets that help us understand the impact of the nature of the matrix on our algorithm. These are described in Section 4.4.

Table 3. Properties of Quasi-Band Matrices.

Matrix	Rows	NNZ	Nonband-Elements	Band-Count	Diagonal-Count	Band-Percentage	Band-Occupancy
Schenk_IBMSDS/matrix_9	103K	2M	9.5K	5	31	99.55	66.73
Fluorem/PR02R	161K	8.2M	61K	5	108	99.25	47
Boeing/pwtk	218K	11M	2.1M	3	71	81.46	61.31
Simon/raefsky3	21K	1.5M	34K	3	93	97.71	75.48
Schenk_AFE/af_shell9	505K	18M	10M	1	25	42.93	59.83
Norris/heart1	3557	1.4M	0.95M	1	239	31.51	52.32
DNVS/trdheim	22K	1.9M	1.4M	1	35	25.92	64.88
HB/cegb2802	2802	277K	156K	1	67	43.73	65
MathWorks/Sieber	2290	15K	8K	1	5	46.18	60.02
Muite/Chebyshev3	4101	37K	16K	1	9	55.59	44.43

4.2 Experimental Platform

We use the K40 GPU from the NVidia Tesla series in our experiments. The host on which K40 is mounted is an Intel i7-4790K CPU with 32GB of global memory. To program the GPU we used CUDA API Version 6.5.

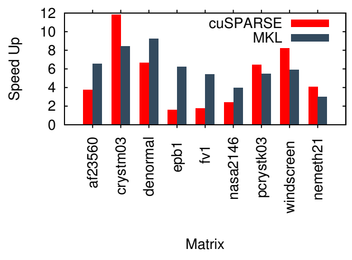
4.3 Results on Real-world Datasets

We show results as speedup when compared to `spmm` kernel(cusparseScsrgemm) in NVIDIA's `cusparse` library [8] and `spmm` kernel(mkl_scsrmultcsr) from theIntel `MKL` library [9] . The computation is done in single precision.

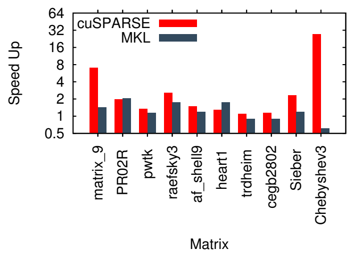
Band Matrices: Figure 1(a) shows the speedup achieved by our algorithm on band matrices from Table 2. In our algorithm, only steps partitioning and computing $A_{bands} \times B_{bands}$ need to be executed as the input matrices are band matrices. The variation in the speedup across the matrices can be partly explained as follows.

Firstly, notice that for matrices with more bandOccupancy, the number of unproductive computations in our algorithm reduce. Hence, for our algorithm, a high bandOccupancy is helpful. On the other hand, given that the input and the output matrices are band matrices, the number of nonzeros in the output matrix depends on how bands are spread in the input matrices. We capture the above via parameter `bandSpread` which is calculated as follows. Take the middle diagonal in each band to be a pivot diagonal. Calculate the distance (absolute difference of the diagonal offset values) between all pairs of pivot diagonals. Divide sum by dimension and multiply it by 100 to arrive at `bandSpread` of the matrix. Table 2 shows the bandSpread value for all matrices considered. It can now be noticed that the performance of the `spmm` routine from `cusparse` degrades as bandSpread increases. Hence, we expect that a high bandSpread usually results in a bigger speedup keeping other parameters fixed.

Following the above, consider matrices `af23560, crystm03, pcrystk03,` and `windscreen` that have near equal size. Matrix `af23560` has less bandOccupancy and less bandSpread when compared to matrix `crystm03`. Hence the former has less speedup. Matrix `windscreen` has high bandOccupancy and high bandSpread compared to that of matrix `pcrystk03` resulting in a better speedup. Matrices `epb1`, `fv1`, and `nasa2146` have very small size and NNZ compared to others. To offset the cost of partitioning and pre-processing, a reasonable size and NNZ are desirable. Hence these matrices show a lesser speedup. Among matrices `epb1` and `fv1`, `epb1` has low bandOccupancy and low bandSpread resulting in low speedup.



(a) Band × Band



(b) Quasi-Band × Quasi-Band

Fig. 1. Results on real world datasets from [10].

Quasi-band Matrices: Figure 1(b) shows the speedup achieved by our algorithm on quasi-band matrices from Table 3. For quasi-band matrices, the performance of our algorithm and also that of the `spmm` library routine from `cusparse` depends on various factors such as nnzPercentage, bandPercentage, bandOccupancy, bandCount, diagonalCount, bandSpread, and the spatial distribution of non-band elements. Because of various dependencies among these factors, it is in general not possible to explain the speedup achieved.

On matrix_9, the high speedup achieved is due to its high bandPercentage. On the other hand, though the matrix `PR02R` has a bandPercentage comparable to that of matrix_9, the speedup is not as high due to poor bandOccupancy. For matrix `trdheim`, the low speedup can be attributed to the fact that all the nonzero elements are present in a small band of diagonal indices (-531 to 531). This ensures that also the `cusparse` library routine performs well. The matrix `Chebyshev3` has its nonzero elements outside of the bands cohesively within the first four rows. Such a structure is likely to create load imbalance in the cusparse library routine for `spmm` which results in a high speedup for our algorithm. (See also Experiment 3 in Section 4.4).

Profile: In Figure 2, we show the time taken by various steps of our algorithm on matrices from Table 3 as a percentage of the overall time. As we use `cusparse` library for computing C_{ss}, this suggests that indeed multiplying sparse matrices is always difficult and focusing on the nature of sparsity is usually beneficial.

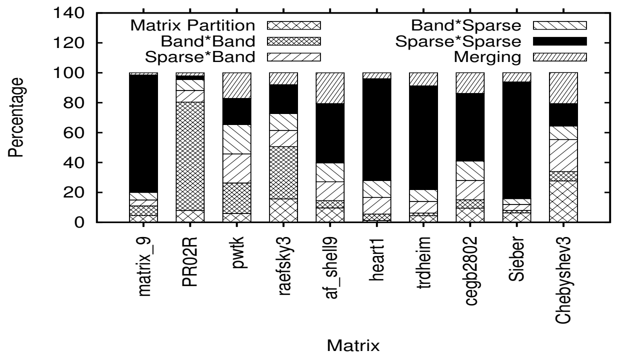


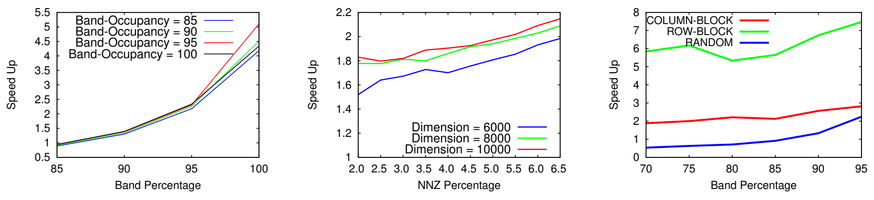
Fig. 2. Percentage of time taken across the various steps of our algorithm on quasi-band matrices from Table 3.

4.4 Synthetic Datasets and Results

We conduct three experiments with synthetic quasi-band matrices to understand the factors that influence the speedup and runtime of our algorithm. We consider synthetic matrices with 16,000 rows, an nnzPercentage of 0.5 and a bandPercentage of 95%. The number of bands and the position of the bands are chosen uniformly at random.

In Experiment 1, we study the impact of bandPercentage and bandOccupancy. The results of this study, shown in Figure 3(a), indicate that a high bandPercentage is favorable to our algorithm compared to bandOccupancy. In Experiment 2 we study the impact of nnzPercentage. The results of this study, shown in Figure 3(b), indicate that a high nnzPercentage improves the performance of our algorithm. This can be understood from the highly parallel strucutre of the algorithm.

In Experiment 3, we consider three data layouts i.e, RANDOM,ROW-BLOCK and COLUMN-BLOCK for the non-band elements in matrix. In case of RANDOM, the non-band elements are spread uniformly at random in non-band part. In case of ROW-BLOCK and COLUMN-BLOCK, the non-band elements are filled in contiguous rows and columns respectively. Figure 3(c) shows the speedup as we vary the bandPercentage for a given data layout. As can be observed, our algorithm too suffers in the RANDOM distribution model but is faster compared to the library routine from `cusparse` once the bandPercentage crosses 85%. Further, the speedup on COLUMN-BLOCK distribution is lower compared to that of ROW-BLOCK. The reason for this can be attributed to the fact that that the library routine from `cusparse` does 5X times better on COLUMN-BLOCK compared to ROW-BLOCK as there is lesser chance of load imbalance in the COLUMN-BLOCK.



(a) bandPercentage (b) nnzPercentage (c) sparsity

Fig. 3. Studies on quasi-band matrices with varying bandPercentage, nnzPercentage and spatial distribution of non-band Elements.

5 Conclusions and Future Work

In this paper, we demonstrated that paying attention to the nature of sparsity will be beneficial when performing operations on sparse matrices on architectures such as the GPU. In future, we would like to focus on other operations such as matrix inversion while focusing on quasi-band matrices.

References

1. N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proc. SuperComputing (SC), 1-11, 2009.
2. A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In Proc. Intl. Conf. on Parallel Processing, 503-510, 2008.
3. A. Gharaibeh, B. Costa, E. Santos-Neto, and M. Ripeanu. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In Proc. International Parallel & Distributed Processing Symposium (IPDPS), 851-862, 2013.
4. Hong, S., Rodia, N. C., and Olukotun, K. On Fast Parallel Detection of Strongly Connected Components in Small-World Graphs. In Proc. SC, Article No.92, 2013.
5. S. Indarapu, M. Maramreddy, and K. Kothapalli. Architecture- and Workload-aware algorithms for Spare Matrix- Vector Multiplication. In Proc. of ACM India Computing Conference, Article No.3, 2014.
6. W. Liu, B. Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In Proc. of IPDPS, 370-381, 2014.
7. K. R. Ramamoorthy, D. S. Banerjee, K. Srinathan and K. Kothapalli A Novel Heterogeneous Algorithm for Multiplying Scale-Free Sparse Matrices, in Proc. of IPDPS Workshops, 637-646, 2015.
8. Nvidia sparse matrix library(cuSPARSE), http://developer.nvidia.com/cusparse.
9. Intel Math Kernel Library, https://software.intel.com/en-us/articles/intel-mkl/.
10. University of Florida (2011) UF sparse matrix collection. Available at: http://www.cise.ufl.edu/research/sparse/matrices/groups.html.
11. W. Yang, K. Li, Y. Liu, L. Shi and L. Wan. Optimization of quasi-diagonal matrix-vector multiplication on GPU. International Journal On High Performance Computing Applications, Vol. 28(2) 183-195, 2014 .