GPU-Accelerated Optimization Of Block Lanczos Solver For Sparse Linear System

Prashant Verma, Kapil Sharma

Abstract: Solving large and sparse system of linear equations has been extensively used for several crypt-analytic techniques. Block Lanczos and Block Wiedemann algorithms are well known for solving large sparse systems. However, the time complexity of such popular method makes it reluctant and hence, the concept of parallelism is made compulsory for such methods. This paper introduced an optimization of Block Lanczos method over finite field using accelerated GPUs. Here we consider GF (2) finite field. The parallel solver for optimization of Block Lanczos is implemented using NVIDIA Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI) to take advantage of multi-level parallelism on multi-node and multi-GPU systems. CUDA-aware MPI has been extensively used to leverage GPU-Direct Remote Direct Memory Access (RDMA) and GPU-Direct Point to Point (P2P) for optimized inter and intra node communication. The proposed solver for optimization of Block Lanczos explored the bandwidth of memory bandwidth on a single Tesla, multi Tesla K40 and multi Tesla P100 GPU nodes. The parallel efficiency is also achieved on DGX system with 8 P100 and 1 V100 Tesla GPU 's respectively.

Index Terms: block lanczos, cryptography, Graphics Processing Unit, GPU-Direct P2P, MIMD, parallel-processing, RDMA.

1. INTRODUCTION

In today's world, the growth of digital information has been increased rapidly therefore information security are imperative for the security requirement of the digital world. There are various crypt-analytic techniques where solving a large sparse system of linear equations over finite field become a challenge due to high computation. For instance, the integer factorization problem using number field sieve (NFS) [1] and symmetric ciphers cryptanalysis involves solving large sparse linear systems over finite field. It may be used in the last phase of the such problem. Block Lanczos [2],[11]and Block Wiedemann [3],[4],[5] are the popular methods to solve such compute intensive problems but the time complexity of such systems is cubic, therefore such system is computationally slow and practically not feasible. As the Block Wiedemann method has been well parallelized therefore we focus on optimizing the Block Lanczos method. To solve compute intensive problems in reasonable amount of time, accelerated units such as general-purpose graphics processing units (GPGPUs) are accomplished. Now there is an increasing trend to create supercomputer clusters where each node hosts multiple GPUs. If we see the top 500 supercomputer list [6] we found that most of them are GPUs based. For instance, the summit supercomputer created by the Oak Ridge National Laboratory (ORNL) [7] hosts up to 6 Volta V100 NVidia GPUs on each node of the cluster and it is 10 times more powerful than Titan which stands in fifth place of top 500 supercomputer list. Thus, it is necessary to develop applications in a way that it can be efficiently scaled over multiple GPGPUs and nodes. The original method for Block Lanczos algorithm [8][9] is roughly split in three steps. The steps are pre-processing, Lanczos iterations and post-processing. At higher densities (>10%), the Block Lanczos is quite costly in terms of performance. This paper describes the optimization exercise carried out on an existing GPU enabled code for Block Lanczos algorithm. The optimization exercise started with understanding, performance profiling of the existing Block Lanczos method. For benchmarking the performance of the original as well as optimized method, linear systems with up to 30,000 unknowns and densities (number of non-zero elements) from 0.01% to 16% are considered. In the proposed work, we optimized a GPU-accelerated Block Lanczos parallel solver for sparse linear systems over binary galois field i.e. GF (2). The optimized Block Lanczos solver is implemented using CUDA [27] version 7.5, CUDA driver version 352.55, 4 Tesla K40 and multi Tesla P100 GPU nodes, an architecture that explore the parallel compute capability of a GPU for general purpose processing. The parallel solver for optimization of Block Lanczos is implemented using NVIDIA Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI) [28] to take advantage of multi-level parallelism on multi-node and multi-GPU systems. CUDA-aware MPI has been extensively used to leverage GPU-Direct Remote Direct Memory Access (RDMA) and GPU-Direct Point to Point (P2P) [29] for optimized inter and intra node communication. The proposed solver for optimization of Block Lanczos explored the bandwidth of memory bandwidth on a single Tesla, multi Tesla K40 and multi Tesla P100 GPU nodes [30]. The optimized solver is constructed in a way that load balancing and optimized communication can be achieved easily. The rest of the paper is formulated as follows. The next section gives the details of the work related to area for solving large sparse system of linear equations over GF (2). Section III gives an overview of Block Lanczos over GF (2) [10][11]. The proposed methodology for optimization of Block Lanczos solver parallelly in multiple hardware platforms is explained in Section IV. The next Section V shows the experiments results over three different hardware platform for performance and scalability and finally Section VI concludes the paper.

2 RELATED WORK

In order to solve dense and sparse system of linear equations over GF(2) the methods that are available was implemented serially [12][13]. The parallel implementation is also available, but they are not optimized with latest hardware platforms available and hence not fully utilized the available hardware resource of latest existing technology. Nvidia introduces series of accelerating cards for researchers to make their application parallel and solve bigger problems in a reasonable amount of time. "Fig. 1" shows how thread blocks in a grid is arranged in a GPU. The existing hardware platform makes the application efficient and scalable. For solving large system of dense and sparse linear equations Gaussian elimination is a well-researched topic and its parallel version over GF (2) is relatively less studied.

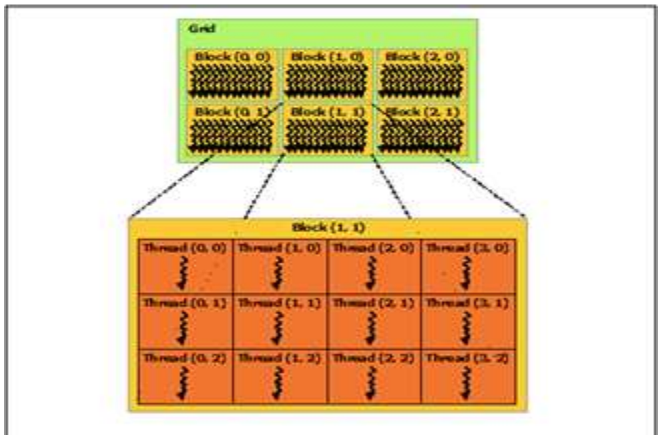


Fig. 1. Grid of Threads Blocks in GPU.

Koc and Arachchige [14] proposed Gaussian Elimination algorithm over finite field GF (2) and implemented the same on the Geometric Arithmetic Parallel Processor known as GAPP. Parkinson and Wunderlich [15] proposed the parallel Gaussian Elimination for finite field GF (2) and the same was deployed on the parallel array processor named as ICL-DAP. Bogdanov et. al. [16] used a hardware that is parallel in architecture to solve Gaussian Elimination over finite field GF (2) quickly. This architecture was implemented on a Field Processor gate array (FPGA). In addition to this the author also evaluates for a possible implementation based on ASIC architecture. All these solutions can solve only small systems of either dense or sparse linear equations over finite field GF (2) and are very costly using special kind of hardware platforms. Albrecht and Pernet [17] proposed the solution of dense system of linear equations over finite field GF (2). The solution used multicore architectures and are very efficient and part of the M4RI (Method of four Russians) library [18]. This solution shows the performance results for 64X64K linear systems of equations and presented that their method is as good as to the implementation by Allan Steel [19] for solving Gaussian Elimination over GF (2) using MAGMA library. This is the first work to solve Gaussian Elimination over GF (2) in a general-purpose processor. This solution shows the-performance results for 64X64K linear systems of equations and presented that their method is as good as to the implementation by Allan Steel [19] for solving Gaussian Elimination over GF (2) using MAGMA library. This is the first work to solve Gaussian Elimination over GF (2) in a general-purpose processor.

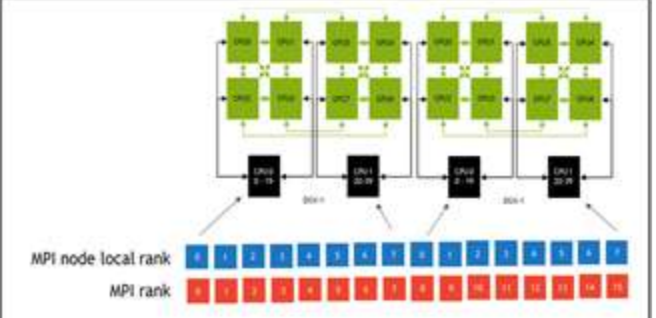


Fig. 2. Handling multi-GPU nodes using MPI

The challenge with the sparse matrix is to reduce the substantial memory requirements by accumulating the only nonzero elements. Depending on the sparsity factor different data structures can be used to save huge amount of memory. Formats to save only non-zero elements can be divided in to mainly two groups. The first groups are those that support modification efficiently. The first groups are those that support modification efficiently. For instance, Dictionary of Keys (DOK), List of lists (LOL), or Coordinate list (COO) comes under this category and typically used for constructing the matrices. The second groups that helps efficient access and matrix operations, such as CSC (Compressed Sparse Column) or CSR (Compressed Sparse Row) [20]. In this paper, we proposed GPU-accelerated optimized Block Lanczos solver for very large sparse system of linear equations. The system has the order of hundreds of thousands of unknowns [21] [22]. It is hard to exploit the larger degree of parallelism to solve such large sparse system of linear equations in a reasonable amount of time. In addition to this the amount of memory required for such systems would not sustain in a single node memory. Therefore, we propose the implementation of an efficient optimized block Lanczos solver for large sparse systems on MIMD (Multiple Instruction Multiple Data) architecture shown in "Fig. 2" which is a kind of cluster of multiple nodes, each equipped with either one graphics processing unit or multiple graphics processing unit. The work that we propose, to the best of our knowledge is not yet available that describes an optimized scalable block lanczos solver for large sparse system over GF (2) and scales efficiently over MIMD architecture with hybrid technology.

3 MOTIVATION AND CONTRIBUTION

The proposed work for optimizing Block Lanczos solver for large sparse system over GF (2) is motivated by the following research gaps.

1) Matrix has large size and high sparsity, so we require an efficient storage formats.
2) Characteristics of GF (2) must be exploited to boost time and space efficiency of the algorithm.
3) Parallelization is a must due to Computation Intensive.

To summarize, the main contributions of our research are as follows:

3.1 Matrix Format

a.) An array stores the cumulative number of nonzero entries till a particular row.
b.) A double array stores the column numbers of the non-zero entries for each row.
Example:

$$\text{Matrix} = \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{array}$$

a.) 1 3 6 b.) {0}, {1,2}, {1,2,3}

3.2 Exploiting GF (2)

a.) Block Lanczos Algorithm itself exploits GF (2) by generating N (computer word size) null vectors per iteration.
b.) Bit packing is done to store all the vectors in the code (each element takes one bit)
c.) Bitwise operations are used for computation (XOR for addition, AND for multiplication) [23].
d.) This not only saves space but also makes computation faster.

3.3 Parallelization
Block Lanczos has three major computational steps.
a.) Matrix-Vector Product
b.) Vector-Vector Product
c.) Vector-Vector Addition
Matrix-Vector Product is the most expensive and dominant computation and needs to be parallelized.

3.4 Data distribution
a.) Each processor reads n/p rows from the input file where n represent no. of rows and p represent number of processors).
b) Matrix – Vector product [24] is done in parallel by offloading the massively parallel part to GPUs and result sent back to the root node.
c) Dependencies are broadcasted, and each processor and co-processor modifies part of the matrix it has.

3.5 CUDA-aware MPI
It enables GPU-Direct P2P transfers for communication of data between the GPUs hosted in a node [25][26]. Nvlink technology also enables peer to peer access.

4 PROPOSED METHODOLOGY FOR OPTIMIZATION
Given a system of linear equations over GF (2) and the task is to find out the equations linearly dependent on others and remove them. Consider the system of equations where no of equations equals to no of variables and of order O (10^6) or higher with density ranging from 0.2 to 2%. To solve such problem the possible approach is to select any n equations (where n is the number of variables) and try to solve them. Find the rank of the matrix using LaMacchia and Odlyzko's Algorithm or Diagonally Scaled Wiedemann Algorithm. Then Solve Mx = 0 (i.e. find the null space of M) which is equivalent of finding dependencies in the matrix for e.g. Input system:
1) m equations over n variables
2) Format Ax = b: A (m x n), b (m x 1)
3) Construction of M: Augment b to A and take its transpose to get M (n + 1 x m)
4) Solve Mx = 0 to get a null vector x
5) Each null vector x gives a dependency relation over the columns of matrix M
6) So, we get a dependency relation over equations of the Input System
7) Solving Mx = 0
We use the Block Lanczos Algorithm to solve this equation. The Input is Matrix B (n1 x n2) and to solve: Bx = 0. The original method for Block Lanczos algorithm is roughly split in three steps shown in "Fig. 3".

- Pre-processing
- Lanczos iterations
- Post-processing

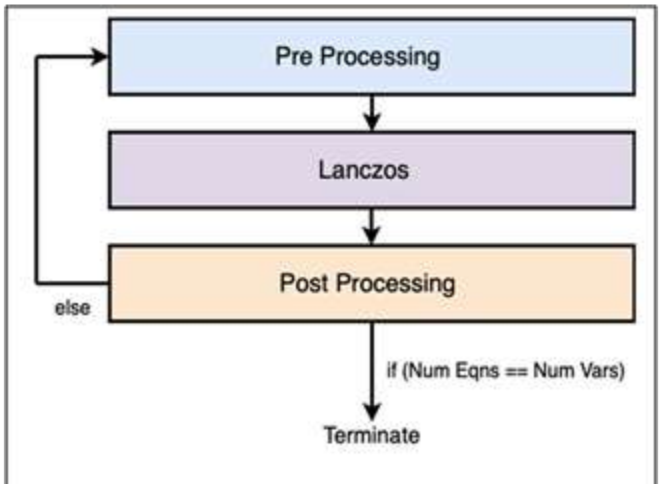


Fig. 3. Steps in Block Lanczos Algorithm

In the pre-processing step, operation such as memory allocation, initialization and loading of the linear system data are done. The Lanczos step involves the iterative part of code that computes the solution. Finally, in post-processing step solution is written to file. The steps for pre-processing are defined as below:
1) Compute $A = B^TB$ (symmetric matrix).
2) Solve Ax = 0 where A (n2 x n2).
3) Generate random matrix Y (n2 x N) where N is computer word size (generally 32 or 64).
4) Solve AX = AY so that column vectors of (X-Y) will belong to the null space of A.
The procedure for lanczos algorithm is explained as below:
1) $V_0 = AY$
2) Algorithm terminates when $V_i^TAV_i = 0$, say for i = m
3) The two distinguish cases are:
  a) If Vm = 0 then AX = AY otherwise
  b) Vm itself belongs to the null space of A
The steps for post processing are described as below:
1) Define Z = [X-Y; Vm]
2) Compute BZ to find which vectors of Z belong to the null space of B.
3) Each null vector gives a dependence relation.
4) Remove as many dependent equations as possible.
The Lanczos step is no longer the most compute intensive step. Especially at lower densities Pre-processing and postprocessing needs to be improved. Both these steps involve considerable amount of file I/O operations. Therefore, avenues for better I/O performance should be explored. At higher densities (>10%), the Block Lanczos is quite costly in terms of performance. Block Lanczos is preferred method for solving sparse linear systems of equations i.e. systems in the form Ax = b, where A is a sparse matrix over GF(2). This paper describes the optimization exercise carried out on an existing GPU enabled code for Block Lanczos algorithm. The optimization exercise started with understanding, performance profiling of the existing Block Lanczos method. For benchmarking the performance of the original as well as optimized method, linear systems with up to 30,000 unknowns and densities (number of non-zero elements) from 0.01% to 16% are considered. The optimization work that has been carried out is further explained:

4.1 Better test data generation
The code requires sparse linear systems as input for benchmarking the performance. A program for generating such systems is also included in the original code. However, the program is quite slow and has hard coded dependency relations. In the optimized code, a new data generating module is added, which is faster and can generate arbitrary relations between columns of the matrix.

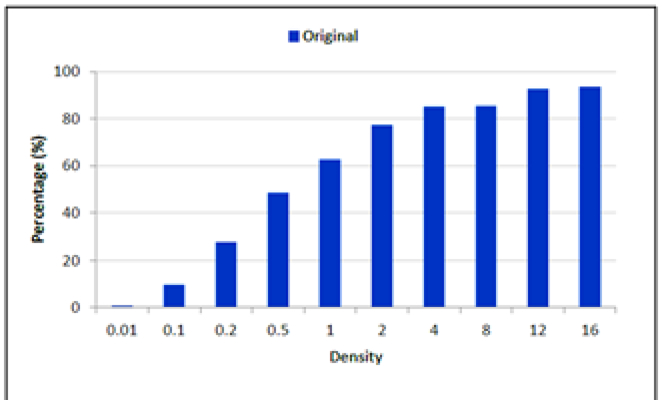


Fig. 4. Execution time share of SpMV and SpMTV in

4.2 Optimization of SpMV and SpMTV operations
The Lanczos step involves repeated calls to two GPU kernels, the Sparse Matrix Vector multiplication (SpMV) and Sparse Matrix Transpose Vector multiplication (SpMTV). “Fig. 4” shows the percentage of total execution time spent on these two kernels combinedly in the original code. The high percentage share of these two kernels make them primary candidate for optimization. Performance of both the kernels is improved with following techniques. The SpMV and the SpMTV are both matrix vector multiplication. The matrix vector multiplication is composed of multiple dot products. Multiple dot products can be executed in parallel. “Fig. 5”, depict dot product of two vectors (in orange colour). In the original code, one single dot product is computed per thread. In the new code, entire warp (vector of 32 threads) is dedicated for computing one dot product. This modification leads to better work distribution among threads of a warp and reduces warp divergence significantly. Warp level approach also results in more coalesced memory access.

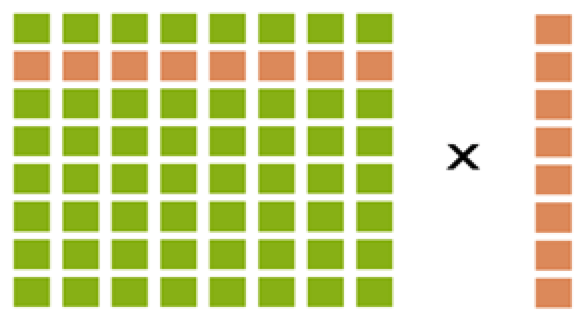


Fig. 5. Dot product performed by warp.

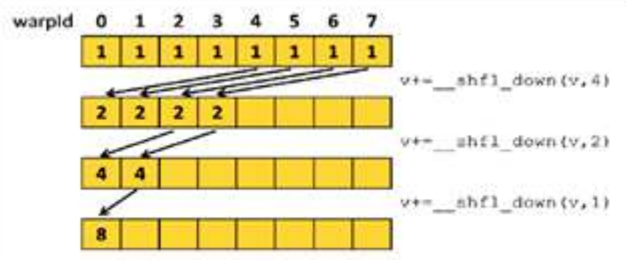


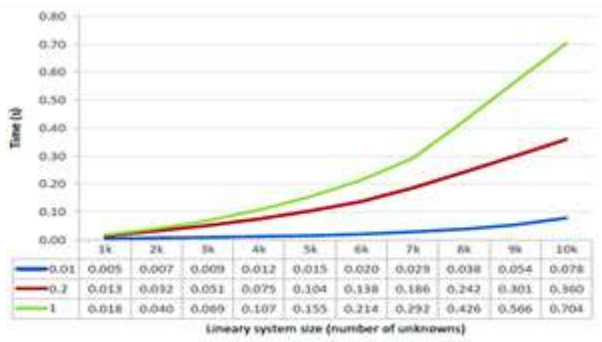
Fig. 6. Warp Shuffle Instruction

In the new optimized method, the launch configurations of kernels are also tweaked for maximum occupancy. Results in terms of higher achieved occupancy can be seen in the profile of new code. Improvement in occupancy leads to better utilization of GPU resources. The dot product operation involves two steps. First point-wise multiplication and second is adding all multiplication results together. The pointwise multiplication can be done in parallel by each thread of the warp. However, for adding the multiplication results together, is reduction operation and thus threads need to cooperate.
The Kepler architecture introduced four shuffle instructions: _shfl(), _shfl_down(), _shfl_up(), and _shfl_xor(). “Fig. 6” shows shuffle down operation on 8 threads. Shuffle instructions allow faster cooperation between threads from same warp. Effectively, threads can read registers of other threads in the same warp. The reduction operation in new version of SpMV is implemented using shuffle instructions. The shuffle-based reduction performs better than even the shared memory atomics-based implementation.

4.3 Miscellaneous changes
a.) A new Makefile is added to source, which can be used to build all the required binaries. It also adds an option for running a test case.
b.) Reduced number of steps needed for running the to three. Manual editing of matrix file is also removed.
c.) Fixed bug in CUDA version checking.

5 EXPERIMENTAL RESULTS
For performance benchmarking of both original and optimized code, linear systems with number of unknowns ranging from 1,000 to 30,000 and density (number of non-zeros) ranging from 0.01% to 16% are generated and then solved. The computed solution is tested against the generated reference solution.

5.1 System Configuration
The configuration used for benchmarking is given below.
- CUDA toolkit 7.5
- CUDA driver version 352.55
- GPU 4 x Tesla K40m
- CPU Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz
- Operating system Cent OS 6.6
- Primary memory (RAM) 128 GB



	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
0.01	0.005	0.007	0.009	0.012	0.015	0.020	0.029	0.038	0.054	0.078
0.2	0.013	0.032	0.051	0.075	0.104	0.138	0.186	0.242	0.301	0.360
1	0.018	0.040	0.069	0.107	0.155	0.214	0.292	0.426	0.566	0.704

Fig. 7. Overall wall-clock time needed for solving the linear system for the original method

"Fig. 7" depicts the wall-clock time taken by program to solve the linear system of equation of varying sizes and densities. The wall-clock time includes time needed for all the three steps (pre-processing, Lanczos, postprocessing). The wall-clock time increases with increase in size of system (number of unknowns). Increase in density also results in increased wall-clock time.
"Fig. 8" shows average amount of time spent on the three steps of the program. The Lanczos step is the most time-consuming part of the code.
"Fig. 9" shows wall-clock time of the optimized code. The improvement in wall-clock time of optimized code is up to 2.5x over original code. Pre-processing and post-processing steps are same in original and optimized code. Therefore, the improvement shown in wall-clock time are from improvements in Lanczos step time.
The resulting change in time share is shown in "Fig. 10". It can be seen that, Lanczos step is no longer the most dominant step.
"Fig. 11" and "Fig. 12" shows exclusive performance gains in the Lanczos step for system with 20,000 and 30,000 unknowns respectively. Figure 4 can be modified to reflect the change in percentage share of SpMV and SpMTV kernels time.

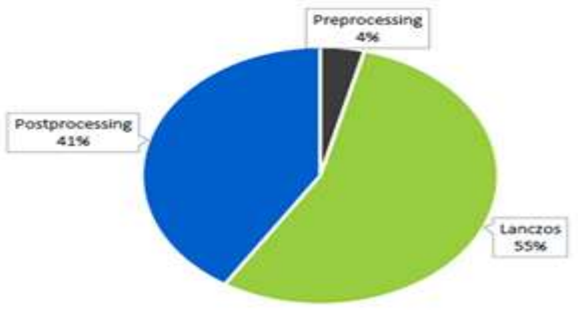
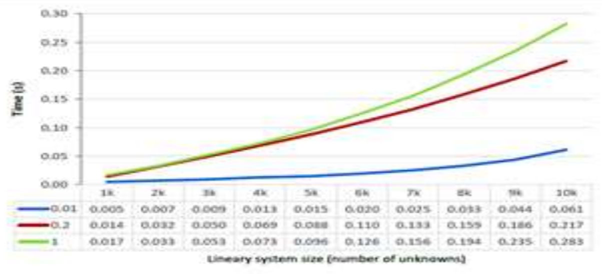


Fig. 8. Time share of different steps (original method)



	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
0.01	0.005	0.007	0.009	0.013	0.015	0.020	0.025	0.033	0.044	0.061
0.2	0.014	0.032	0.050	0.069	0.088	0.110	0.133	0.159	0.186	0.217
1	0.017	0.033	0.053	0.073	0.096	0.126	0.156	0.194	0.235	0.283

Fig. 9. Overall wall-clock time needed for solving the linear system for the optimized method

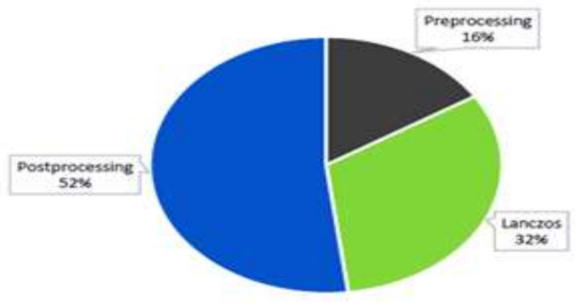
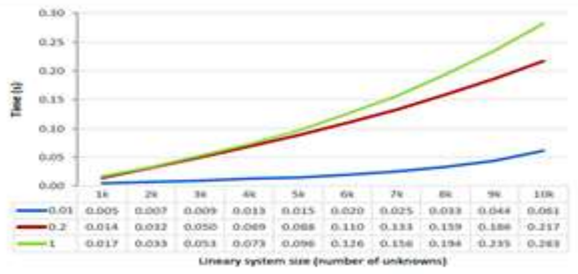


Fig. 10. Time share of different steps (optimized method)



	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
0.01	0.005	0.007	0.009	0.013	0.015	0.020	0.025	0.033	0.044	0.061
0.2	0.014	0.032	0.050	0.069	0.088	0.110	0.133	0.159	0.186	0.217
1	0.017	0.033	0.053	0.073	0.096	0.126	0.156	0.194	0.283

Fig. 11. Comparison of Lanczos step timing (20,000

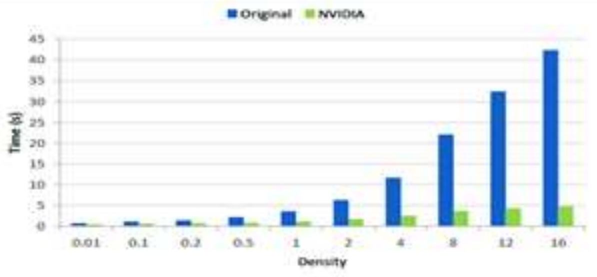


Fig. 12. Comparison of Lanczos step timing (30,000 unknowns)

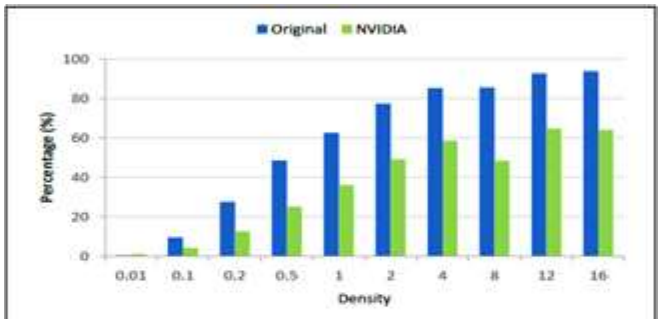


Fig. 13. Execution time share of SpMV and SpMTV in original and optimized method (30,000

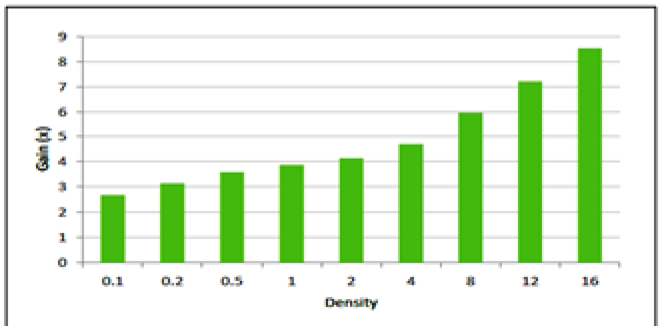


Fig. 14. Performance gain in SpMV and SpMTV

Fig. 13" shows the percentage share of the two kernels. "Fig. 14" shows the gain achieved by of optimized version of code over original code. Note that for this comparison, only execution time for SpMV and SpMTV are considered. The code is also benchmarked on P100 which based on the latest NVIDIA GPU architecture Pascal.

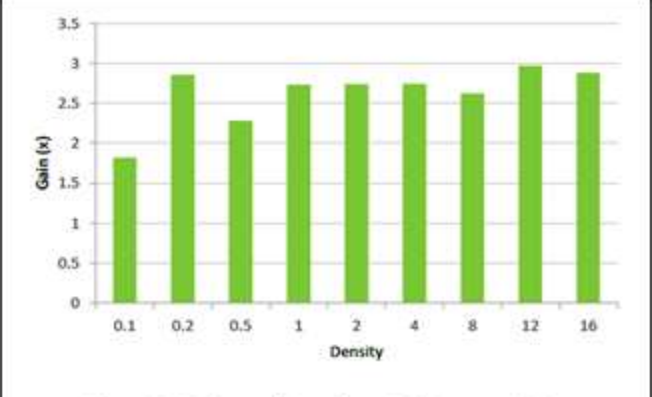


Fig. 15. Gain achieved on P100 over K40

"Fig. 15" illustrates the performance gained by running the same code on P100 vs K40 (Kepler architecture) without any modifications to code. The code on an average executed 2-3x faster on P100 compared to K40.

6 CONCLUSIONS

The Lanczos step is no longer the most compute intensive step in the code. Especially at lower densities Pre-processing and post-processing needs to be improved. Both these steps involve considerable amount of file I/O operations. Therefore, avenues for better I/O performance should be explored. At higher densities (> 10%), the Block Lanczos is quite costly in terms of performance. For such cases, even the dense solver such as Gaussian elimination can be tried. The SpMV and SpMTV are essentially matrix-vector operations. In SpMV the matrix is in normal format while in SpMTV the matrix is in the transposed format. This change leads to huge change in performance of code. The transpose multiply is 3-4x slower than the normal multiply. This penalty can be avoided by precomputing the transpose of matrix and then calling same matrix vector multiply for both operations. The overhead of this approach in terms of execution time is, time needed for transposing the matrix and in terms of memory is doubling the storage space of matrix. The computational overhead should be negligible because transposing is roughly equivalent of two calls to SPMV (one round trip of entire matrix data).

REFERENCES

[1] Qi Wang, Xiubin Fan, Hongyan Zang, Yu Wang, The Space Complexity Analysis in the General Number Field Sieve Integer Factorization, Theoretical Computer Science, Volume 630, 2016, Pages 76-94,ISSN0304-3975, https://doi.org/10.1016/j.tcs.2016.03.028.
[2] B. Sengupta, A. Das, Use of SIMD-based data parallelism to speed up sieving in integer-factoring algorithms, IACR Cryptology ePrint Archive (2015) 44.
[3] P. Giorgi, R. Lebreton, Online order basis algorithm and its impact on the block Wiedemann algorithm, in: Proc. 39th Int. Symp. Symbolic and Algebraic Computation (ISSAC'14), ACM, 2014, pp. 202–209.
[4] A. G. Huang, Parallel Block Wiedemann-based GNFS algorithm for integer factorization, Master thesis, St. Francis Xavier University, Canada (2010).
[5] T. Zhou, J. Jiang, Performance modeling of hyper-scale custom machine for the principal steps in block Wiedemann algorithm, The J. Supercomputing (2016) 1–23.
[6] "Top500 list - November 2017," https://www.top500.org/list/2019/11/.
[7] "Summit: Oak ridge national laboratory's next high-performance supercomputer," https://www.olcf.ornl.gov/olcfresources/computesystems/summit.
[8] I. Flesch, A new parallel approach to the Block Lanczos algorithm for finding null spaces over GF (2), Master thesis, Utrecht University, the Netherlands (2006).
[9] E. Thom´e, A modified block Lanczos algorithm with fewer vectors, arXiv preprint arXiv:1604.02277.
[10] http://en.wikipedia.org/wiki/Lanczos algorithm (2009). Intel Corporation, Technical Report.
[11] Laurence T. Yang, Ying Huang, Jun Feng, Qiwen Pan, Chunsheng Zhu, An improved parallel block Lanczos algorithm over GF (2) for integer factorization, Information Sciences, Volume 379, 2017, Pages 257-273, ISSN 0020-0255, https://doi.org/10.1016/j.ins.2016.09.052.
[12] T. L. Xu Block Lanczos-based parallel GNFS algorithm for integer factorization, Master thesis, St. Francis Xavier University, Canada (2007).
[13] L. T. Yang, L. Xu, S.S. Yeo, S. Hussain, An integrated parallel GNFS algorithm for integer factorization based on

Linbox Montgomery block Lanczos method over GF (2), Computers & Mathematics with Applications 60 (2) (2010) 338–346.
[14] K. Koc and S. N. Arachchige, "A fast algorithm for gaussian elimination over gf (2) and its implementation on the gapp." J. of Parallel and Distributed Computing., vol. 13, no. 1, pp. 118–122, 1991.
[15] D. Parkinson and M. Wunderlich, "A compact algorithm for gaussian elimination over gf (2) implemented on highly parallel computers," Parallel Computing, vol. 1, no. 1, pp. 65–73, Aug. 1984.
[16] A. Bogdanov, M. C. Mertens, C. Paar, J. Pelzl, and A. Rupp, "A parallel hardware architecture for fast gaussian elimination over gf (2)," 14th IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 237–248, 2006.
[17] M. R. Albrecht, G. V. Bard, and C. Pernet, "Efficient dense gaussian elimination over the finite field with two elements," CoRR, vol. abs/1111.6549, 2011.
[18] "M4ri library," https://github.com/malb/m4ri.
[19] W. Bosma, J. Cannon, and C. Playoust, "The magma algebra system i: The user language," Journal of Symbolic Computation, vol. 24, no. 3-4, pp. 235–265, Oct. 1997.
[20] Aydin Buluc¸, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA '09). Association for Computing Machinery, New York, NY, USA, 233–244. DOI: https://doi.org/10.1145/1583991.1584053
[21] N. L. Zamarashkin and D. A. Zheltkov, "Gpu based acceleration of parallel block lancoz solver," Lobachevskii Journal of Mathematics, vol. 39, no. 4, pp. 596–602, May 2018.
[22] "Gpu acceleration of dense matrix and block operations for lanczos method for systems over large prime finite field," in Supercomputing. RuSCDays, ser. Communications in Computer and Information Science, vol. 793. Springer, 2017, pp.14–26.
[23] I. Gupta, P. Verma, V. Deshpande, N. Vydyanathan and B. Sharma, "GPU-Accelerated Scalable Solver for Large Linear Systems over Finite Fields," 2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), Solan Himachal Pradesh, India, 2018, pp. 324-329, doi: 10.1109/PDGC.2018.8745743.
[24] B. Vastenhouw, R. H. Bisseling, A two-dimensional data distribution method for parallel sparse matrix-vector multiplication, SIAM Review 47 (1) (2004) 67–95.
[25] An introduction to cuda-aware mpi". https://devblogs.nvidiacom/parallelforall/introduce-cudaaware-mpi.
[26] "FAQ: Running cuda-aware open MPI" https://www.open-mpi.org/faq/?category=runcuda.
[27] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," Queue, vol. 6, no. 2, pp. 40–53, Mar. 2008.
[28] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.
[29] "Nvidia GPUDirect," https://developer.nvidia.com/gpudirect.
[30] C. Rea˜no and F. Silla, "Performance Evaluation of the NVIDIA Pascal GPU Architecture: Early Experiences," 2016 IEEE 18th International Conference on High Performance Computing and Communications; Sydney, NSW, 2016, pp. 1234-1235, doi: 10.1109/HPCCSmartCity-DSS.2016.0173.

AUTHORS PROFILE

Prashant Verma received the B.Tech. degree in information technology from Harcourt Butler Technological University, Kanpur, India, in 2007. He is a scientist with Defence Research and Development Organization, New Delhi. Presently he is pursuing M. Tech at Department of IT, Delhi Technological University, New Delhi, India. His current research interests include parallel and distributed computing, grid and cloud computing and information security.

Kapil Sharma received the B.E. degree in computer science from Maharshi Dayanand University, Rohtak, India, in 2000, the M. Tech degree in computer science from the Institute of Advanced Studies in Education, Sardarshahar, India, in 2005, and the Ph.D. degree in computer science from Maharshi Dayanand University in 2011. He is a Professor and the Head of the Department of IT, Delhi Technological University, New Delhi, India. He has published over 100 research papers in referred journals. His current research interests include pattern recognition, computer vision, and soft computing.