Defence Science Journal, Vol. 72, No. 6, November 2022, pp. 836-845, DOI : 10.14429/dsj.72.17656

Solution of Large Sparse System of Linear Equations over GF(2) on a Multi-Node Multi-GPU Platform

Shruti Rawal* and Indivar Gupta

DRDO - Scientific Analysis Group (SAG), Metcalfe House, Delhi – 110 054, India
**E-mail: shrutirawalhans@gmail.com*

ABSTRACT

We provide an efficient multi-node, multi-GPU implementation of the Block Wiedemann Algorithm (BWA) to find the solution of a large sparse system of linear equations over GF(2). One of the important applications of solving such systems arises in most integer factorization algorithms like Number Field Sieve. In this paper, we describe how hybrid parallelization can be adapted to speed up the most time-consuming sequence generation stage of BWA. This stage involves generating a sequence of matrix-matrix products and matrix transpose-matrix products where the matrices are very large, highly sparse, and have entries over GF(2). We describe a GPU-accelerated parallel method for the computation of these matrix-matrix products using techniques like row-wise parallel distribution of the first matrix over multi-node multi-GPU platform using MPI and CUDA and word-wise XORing of rows of the second matrix. We also describe the hybrid parallelization of matrix transpose-matrix product computation, where we divide both the matrices row-wise into equal-sized blocks using MPI. Then after a GPU-accelerated matrix transpose-matrix product generation, we combine all those blocks using MPI_BXOR operation in MPI_Reduce to obtain the result. The performance of hybrid parallelization of the sequence generation step on a hybrid cluster using multiple GPUs has been compared with parallelization on only multiple MPI processors. We have used this hybrid parallel sequence generation tool for the benchmarking of an HPC cluster. Detailed timings of the complete solution of number field sieve matrices of RSA-130, RSA-140, and RSA-170 are also compared in this paper using up to 4 NVidia V100 GPUs of a DGX station. We got a speedup of 2.8 after parallelization on 4 V100 GPUs compared to that over 1 GPU.

Keywords: GF(2); GPGPU computing; MPI; CUDA; Block Wiedemann Algorithm; NVidia V100 GPU; NVidia DGX station; HPC cluster

1. INTRODUCTION

We propose here to solve a large sparse system of linear equations over GF(2) which arises in most of the integer factorization algorithms like a quadratic sieve or General Number Field Sieve (GNFS)[1] where we need to select those rows of the matrix whose linear combinations will result in a zero row modulo 2. These integer factorization algorithms have great importance in cryptanalysis because the security of one of the widely used cryptosystems, namely RSA,[2] relies on the hardness of factoring large integers. There are numerous other scientific and engineering problems that involve solution of large sparse linear systems, such as analysis of electric power systems and study of many real-world multi-physics problems.[3]

We cannot use general matrix-solving methods like Structured Gaussian Elimination[4] to solve such matrices because the sparsity of the matrix will get lost in the intermediate stages of such methods, and we will need larger memory storage to store these large size denser matrices. Thus, using Iterative Solvers like Wiedemann[5] or Lanczos[6] Methods is better. However, for a matrix with elements over GF(2), two more optimised algorithms with lesser iterations exist, namely, Block Wiedemann and Block Lanczos.[7] We chose the former method to solve our problem because it is highly parallelizable.

Block Wiedemann Algorithm (BWA) is a well-researched topic. However, parallel BWA is relatively less studied. Coppersmith[8] proposes in the Block form of the Wiedemann algorithm to use the capability of computers to do multiple bit-level operations in one clock cycle. Unlike the Wiedemann algorithm, which considers single vectors, he considers blocks of vectors or matrices to do simultaneous computations. Thus, the number of iterations required to compute the solution decreases with the column size of these block vectors.

Gilles Villard[9] analyses the probability of success of the block algorithm proposed by Coppersmith for solving large sparse systems $Aw = 0$ of linear equations over a field K. He proves that the input parameters of the algorithm may be tuned such that, for any input system, a solution is computed with high probability for any field. He[10] also studies the algorithm using matrix polynomials. Emmanuel Thomé[11] describes how the half-gcd algorithm can be adapted to speed up the sequential linear generator computation stage of Coppersmith's BWA for solving large sparse linear systems over any finite field. He later also describes a new algorithm,[12] where he provides

Received : 12 November 2021, Revised : 24August 2022
Accepted : 3 November 2022, Online published : 6 December 2022

a subquadratic variant of Coppersmith's "matrix Berlekamp-Massey." Pascal Giorgi and Romain Lebreton[13] propose an online algorithm for order basis, which allows for both early termination and minimal input requirement while keeping quasi-optimal complexity in the order.

Erich Kaltofen[14] analyzes Coppersmith's BWA for the parallel solution of sparse linear systems. The author proves that by using certain randomizations on the input system, the parallel speedup is roughly by the number of vectors in the blocks when using as many processors. Erich Kaltofen and Austin Lobo[15] describe a coarse-grain parallel approach for the homogeneous solution of linear systems. They solve a $252,222 \times 252,222$ system with about 11.04 million nonzero entries over the Galois field with two elements using four processors of an SP-2 multiprocessor in about 26.5 hours of CPU time. Bastien Vialla[16] provides an efficient implementation of BWA on NUMA multicore architecture using the tbb/MPI.

Allan K. Steel[17] shows how some very large multivariate polynomial systems over finite fields can be solved by Gröbner basis techniques coupled with the BWA. They have implemented a dense variant of the Faugère F4 Gröbner basis algorithm and the BWA within the Magma Computer Algebra System.

Our contribution: In this paper, we use the Block Wiedemann Algorithm to solve very large sparse square matrices with entries over the Galois field with two elements whose size may be as large as hundreds of millions of rows. To solve such large systems in a considerable amount of time, it becomes necessary to exploit larger degrees of parallelism. Therefore, this paper proposes a multi-node multi-GPU scalable implementation of a sparse linear solver over GF(2) based on the Block Wiedemann Algorithm. The solver is implemented using Nvidia CUDA (Compute Unified Device Architecture),[18-19] which is a parallel computing architecture and application programming interface(API) developed by Nvidia for general computing on its graphical processing units (GPUs). We use the Message Passing Interface (MPI)[20] to distribute the implementation of the sparse solver across multiple multi-GPU nodes. To the best of our knowledge, this is the first work that describes the scalable implementation of the Block Wiedemann algorithm to solve a large sparse system of linear equations over GF(2) that scales efficiently over a hybrid cluster where each node is equipped with graphics processing units (GPUs). We could not compare any earlier results with our findings because we could not find any other latest work on multi-node multi-GPU parallel implementation of the Block Wiedemann Algorithm for the solution of large sparse linear systems, especially over GF(2). The last parallel approach for the homogeneous solution of linear systems over GF(2) was done in 1999.[15] However, it will not be justified to compare our implementation with theirs because of significant advancements in hardware in these many years.

We use the sparse linear solver's parallel module to benchmark an HPC cluster. The performance and scalability of this parallel module are compared in this paper by performing its parallelization in two ways. In the first method, we parallelize the solver over multiple CPU ranks of multiple cluster nodes using MPI. In the second method, we parallelize it over multiple GPUs spread over multiple nodes. Our experimental results show that the performance of multi-node multi-GPU parallelization is much better than that of parallelization over just multiple MPI processors of multiple CPU nodes. We observe that only 4, 8, or 12 GPUs spread over multiple nodes solve a given large sparse system in approximately less than half time as compared to the least time taken to solve the same system by 512, 1024, or 2048 number of CPU ranks spread over multiple nodes. The GPU architecture is Volta, on which the solver shows a parallel efficiency of 94.4 % on up to 8 V100 GPUs (spread over two nodes), which drops to 72.7 % and 49.4 % on 12 (spread over three nodes) and 16 (spread over four nodes) V100 GPUs, respectively. Thus, we can say that the solver can effectively scale across multiple GPUs on multiple nodes, and by scaling the problem size, we can further saturate a larger number of GPUs.

The rest of this paper is organized as follows. The next section overviews the Block Wiedemann Algorithm (BWA) and explains the sequential approach to compute a minimal polynomial matrix. The hybrid parallel implementation of matrix-matrix products and matrix transpose-matrix products computation for the sequence generation and solution evaluation parts of BWA is described in section III. Detailed timings of the complete solution of some number field sieve matrices are compared in section IV using up to 4 NVidia V100 GPUs of a DGX station. Section IV further compares the scalability and performance of the parallel implementation of the sequence generation part of the sparse linear solver across multiple GPUs of multiple nodes with that across only multiple CPU ranks of multiple nodes. Section V concludes the paper.

2. BLOCK WIEDEMANN ALGORITHM

Coppersmith's Block Wiedemann Algorithm (BWA)[8] is useful when we need to find a homogeneous solution for a singular, square matrix over GF(2). It uses the ability to simultaneously perform multiple operations in GF(2) on blocks of vectors using complete word size. Let us first briefly go through the Wiedemann algorithm to understand this algorithm.

2.1 Wiedemann Algorithm

Wiedemann Algorithm,[5,8] when applied to a singular square matrix A of size $N \times N$ gives a nonzero solution vector w, such that

$$Aw = 0 \tag{1}$$

Let there be two random vectors, $u, v \in F_2^N$. These vectors u, v are then used to compute

$$a^{(i)} = u^t A^i v, 1 \le i \le 2N+1 \tag{2}$$

and let

$$a(\lambda) = \sum_i a^{(i)} \lambda^i \tag{3}$$

Suppose $f(\lambda)$ is a least degree monic polynomial such that $f(A) = 0$, then $f(\lambda)$ is called the minimal polynomial[15] of

A. If $f(\lambda) = \lambda^l + c_{l-1}\lambda^{l-1} + ... + c_0$, then we know that

$$f(A)v = 0 \tag{4}$$

Then

$$A^{\delta}(A^{l-\delta}v + c_{l-1}A^{l-\delta-1}v + ... + c_{\delta}v) = 0 \tag{5}$$

where $\delta \geq 0$ is the minimum value for which c_{δ} remains nonzero.[15] If A is nonsingular, then δ becomes 0. This minimal polynomial is found using Berlekamp-Massey Algorithm.[21] Assume that the vector in parenthesis in (5) is $\tilde{w}$ and suppose $\tilde{w}$ is nonzero. Thus it implies that $A^{\delta}\tilde{w} = 0$ and, therefore,

$$w = A^{\delta-1}\tilde{w} \neq 0 \tag{6}$$

Thus $Aw = 0$.

2.2 Block Wiedemann Algorithm

A block version of the Wiedemann algorithm is proposed by Coppersmith[8] which works with matrices $U, V \in F_2^{N\times m}$ instead of vectors u and v. Here, m is a multiple of word size. Thus, we can solve for $AW = 0$ using this algorithm, where A is a singular, square matrix such that $A \in F_2^{N\times N}$ and W is a matrix such that $W \in F_2^{N\times m}$. This algorithm can be divided into four steps described here in the subsections.

2.2.1 Random Block Vector Generation

In this part of the algorithm, we get as input the singular, square matrix $A \in F_2^{N\times N}$ and we generate two matrices $U, V \in F_2^{N\times m}$ randomly in such a manner that U^TAV becomes a nonsingular matrix with entries over GF(2) of size m x m. Here, m is chosen to be a multiple of word size, i.e., 64 or 128. If we cannot satisfy the non-singularity condition, we keep finding another set of random matrices U and V until the product U^TAV becomes nonsingular.

2.2.2 Sequence Generation

Using the block vectors or matrices generated in the previous step, a sequence of matrix-matrix and matrix transpose-matrix products η^i are computed such that:

$$\eta^i = U^TA^iV, 1 \leq i \leq \frac{2N}{m} + \varepsilon \tag{7}$$

In our implementation, we consider $\varepsilon = 6$. The result is a sequence of matrices with entries in GF(2) of size $m \times m$, and we consider them as coefficients of a polynomial matrix, $\eta(\lambda)$ such that

$$\eta(\lambda) = \sum_i \eta^{(i)}\lambda^i \tag{8}$$

2.2.3 Minimal Polynomial Matrix Computation

In this step of the algorithm, $\eta(\lambda)$, the sequence of $m \times m$ matrices, is the input, and $\psi(\lambda)$, the linear generator or the minimal polynomial matrix of sequence $\eta(\lambda)$ is the output. Assume that the length of the linear generator $\psi(\lambda)$ is $K+1$ then

$$\psi(\lambda) = \lambda^K + c_{K-1}\lambda^{K-1} + ... + c_1\lambda + ... + c_0 \tag{9}$$

where, $c_i \in F_2^{m\times m}$ for $0 \leq i \leq K-1$.[15] $\psi(\lambda)$ is generated using a generalized block version of the Berlekamp-Massey algorithm,[8,22] which is explained further in a subsequent section. This computation is highly sequential.

2.2.4 Solution Evaluation:

We know that $\psi(\lambda)$, as defined in Eqn. (9), is the minimal polynomial matrix of A. Thus with reference to Eqn. (4), it implies that $\psi(A)V = 0$. The only difference here is that the coefficients of this polynomial are matrices. With reference to (6), we get

$$W = A^{\delta-1}\tilde{w} \neq 0 \tag{10}$$

where, $\tilde{w} = A^{K-\delta}V + C_{K-1}A^{K-\delta-1}V + ... + C_{\delta}V$ and δ is the minimum positive value for which C_{δ} remains nonzero. It is to be noted that $W, \tilde{w} \in F_2^{N\times m}$ in Eqn. (10) are matrices unlike w and $\tilde{w}$ in Eqn. (6), which are vectors of size N. Thus, W gives m number of candidate solution vectors of size N in a single invocation simultaneously. All these solution vectors may or may not be different or nonzero. The nonzero, unique solutions can also be verified after multiplication with matrix A.

2.3 Sequential Approach for Minimal Polynomial Matrix Computation

The minimal polynomial matrix, $\psi(\lambda)$ as in Eqn. (9) of sequence, $\eta(\lambda)$ as in Eqn. (8), is generated using a generalized block version of the Berlekamp-Massey algorithm.[8-22] This algorithm is purely sequential and iteratively calls three functions $2N/m + \varepsilon$ times. Before understanding these functions, let us review the matrices, polynomial matrices, and other data elements involved.

Algorithm 1. Minimal polynomial matrix computation.

- **Input**

  $\eta(\lambda)$: sequence of $m \times m$ matrices of length L,

  where $L = \frac{2N}{m} + \varepsilon$

- **Output**

  $\psi(\lambda)$: linear generator of $\eta(\lambda)$,

  d: vector of size $2m$ which stores the degrees of each row of $\psi(\lambda)$ and $\beta(\lambda)$, where $\psi(\lambda), \beta(\lambda)$ are $m \times m$ polynomial matrices. Here, $\psi(\lambda)$ is the polynomial matrix which stores the required Minimal Polynomial Matrix, and $\beta(\lambda)$ stores that previous value of $\psi(\lambda)$ for

which the length of the polynomial matrix $\psi(\lambda)$ was last updated.

- **Discrepancy Matrix**
  Δ: $2m \times m$ discrepancy matrix
  where, Δ is the discrepancy matrix used to calculate discrepancy in this algorithm. Δ is the combination of the highest degree coefficients in $\psi(\lambda)$ and $\beta(\lambda)$ such that the first m rows of Δ represent the highest degree coefficient in $\psi(\lambda)$ and the next m rows represent the highest degree coefficient in $\beta(\lambda)$. It is to be noted that these coefficients are matrices of size $m \times m$.
- **Linear Transformation matrix**
  τ: $2m \times 2m$ linear transformation matrix where τ is a matrix that is set to be an identity matrix in each iteration and then updated based on values of Δ and d.
- **Initialization**

$\psi(\lambda) = I, \beta(\lambda) = Ix, d[1,..,2m] = 1, \tau = 0, i = 1, L = 2N/m + 6$

$i \leftarrow 0$

while $i < L$ do

DiscrepancyFinder$(\Delta, i, \eta(\lambda), \psi(\lambda), \beta(\lambda))$

/* find coefficient of λ^i */

Triangulation(τ, Δ, d, m)

/* compute τ using Δ */

UpdatePolyMatrix$(\psi(\lambda), \beta(\lambda), \tau, m)$

/* update $\psi(\lambda), \beta(\lambda)$ using τ */

- **End**

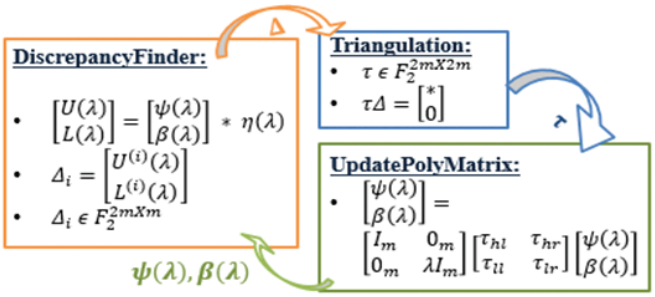


Figure 1. Data flow between functions of a block version of the Berlekamp-Massey algorithm.

The three functions invoked L times in algorithm 1 are described as follows. Figure 1 shows the data flow between these three algorithm functions.

2.3.1 Discrepancy Finder

This function takes the polynomial matrices $\eta(\lambda), \psi(\lambda)$, and $\beta(\lambda)$ as inputs. It first finds the coefficient of λ^i from the product of $\psi(\lambda)$ and $\eta(\lambda)$. The resulting coefficient is a matrix of size $m \times m$. This coefficient forms the first m rows of the discrepancy matrix Δ. Similarly, the coefficient of λ^i from the product of $\beta(\lambda)$ and $\eta(\lambda)$ is stored in the following m rows of Δ. Hence, Δ is the output of this function.

2.3.2 Triangulation

This function takes the discrepancy matrix Δ and the degree vector d as inputs. In every iteration, it first initializes the transformation matrix τ to be an identity matrix. Then it updates τ with respect to Δ such that product of τ and Δ gives a matrix that has nonzero values in only the first m rows, and all the values in the following m rows of the resultant product are zero, i.e., $\tau\Delta = \begin{bmatrix} * \\ \hline 0 \end{bmatrix}$.

2.3.3 UpdatePolyMatrix

In this function, polynomial matrices $\psi(\lambda)$, $\beta(\lambda)$, and the transformation matrix τ are the inputs. Here, each coefficient of $\psi(\lambda)$ and $\beta(\lambda)$ is updated using values in τ as shown in Fig. 2. For ease of computation, τ is divided into four parts.

$$\begin{bmatrix} \psi(\lambda) \\ \beta(\lambda) \end{bmatrix} = \begin{bmatrix} I_m & 0_m \\ 0_m & \lambda I_m \end{bmatrix} \begin{bmatrix} \tau_{hl} & \tau_{hr} \\ \tau_{ll} & \tau_{lr} \end{bmatrix} \begin{bmatrix} \psi(\lambda) \\ \beta(\lambda) \end{bmatrix}$$

Figure 2. Updation of polynomial matrices $\psi(\lambda)$, $\beta(\lambda)$ using τ

3. HYBRID PARALLEL SEQUENCE GENERATION AND SOLUTION EVALUATION

The sequence generation part of the algorithm is highly compute-intensive as it involves multiple matrix-matrix and matrix transpose-matrix multiplications to generate the sequence

$$\eta^i = U^T A^i V, 1 \le i \le \frac{2N}{m} + \varepsilon \qquad (11)$$

where, $U, V \in F_2^{N \times m}$ and $A \in F_2^{N \times N}$. Here, the sheer size of matrix A raises the primary difficulty. With reference to the data from the factorization of RSA-768, it is known that the matrix generated after the Sieving step had 48 billion rows and 35 billion columns which, after filtering, got reduced to 193 million rows and columns with only 144 nonzero entries per row in average. Also, from the RSA-130 challenge, we know that matrix had 3.5 million rows and columns with an average of 39.4 nonzero entries per row. It is a waste of memory if we store such large matrices in their standard form, so we store them in a compressed format, as described in the following subsection.

3.1 Sparse Matrix Representation

We chose a modified Compressed Sparse Row (CSR) format to store our large sparse matrix A. In Fig. 3, we have taken the example of a small, sparse square matrix with elements over GF(2) to understand this format. In this modified CSR format, the first element in each row represents the number of nonzero values (1s in our case) in the respective sparse row. After the first element, the column indices of those nonzero values are stored. We store all these values in hexadecimal. Unlike the normal CSR format, we do not need to store any values because the only nonzero value in GF(2) is 1. Thus, the amount of memory required to store large sparse matrices in these formats directly depends on the number of nonzero values in these matrices. The larger the sparsity of a matrix, the lesser the memory required to save such a matrix.

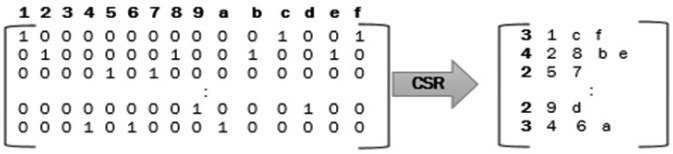


Figure 3. Modified CSR format to store sparse matrices.

3.2 Sparse Matrix-Matrix Multiplication

The matrices U and V are stored in chunks or blocks of 64-bits in hexadecimal format, as shown in Fig. 4. Thus, to find the Sparse Matrix-Matrix (SpMM) product AV, which will be a matrix of size $N \times m$ (where m is a multiple of word size, i.e., 64 or 128), we first read the elements of matrix A in each row. Each of these elements signifies the column indices of nonzero values. Then only the corresponding rows of each word-size block of matrix V are XORed to generate corresponding word-size blocks of matrix AV. Figure 5 shows an example of an SpMM multiplication where $m = 128$ and x_i, y_i are 64-bit words in GF(2) where $i \in [1, N]$. This implies that XORing one 64-bit size block of V generates the results for 64 columns of each row of matrix AV in just one operation cycle.

```
850794ceb3432972   f1287352efef2af4
2aaa2ad73d3d2e73   7f737fc575885825
321559f8cbfd53c7   c42362c0f72c4e48
        :                  :
40298978c7e52bc2   0048a9f1e152c283
eab9dfded0355983   8f603a84363f02dd
```

Figure 4. An example of matrix V of size $N \times 128$

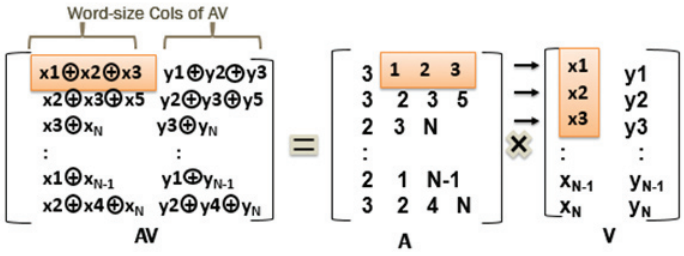


Figure 5. Word-wise XORing of rows of V based on column indices of A

3.3 Hybrid Parallel Sparse Matrix-Matrix Multiplication

We have parallelized this SpMM multiplication on a multinode multi-GPU platform. The number of MPI processors launched on each node is chosen to equal the number of GPUs available on each node. By doing this, we can equally distribute the load among all the GPUs available on a node. Thus, with each GPU, we associate one MPI processor. The MPI node local rank of each process on each node is used to bind that process to the appropriate GPU of that node. We pass the node's local rank to the cudaSetDevice() function to set the device (GPU) on which the active host thread should execute the device code.

To parallelize any operations, the most important requirement is that those computational operations should be independent of each other. In SpMM multiplication explained above, we can easily identify that the computation of all rows of AV are independent operations; therefore, they can be computed in parallel. Thus, we first scatter the rows of matrix A among the chosen number of MPI processes of multiple cluster nodes using MPI_Scatter. Matrix V is broadcasted to all the MPI processes using MPI_Bcast, and further, its memory allocation is done at each of the GPUs using CUDA_Malloc and CUDA_Memcpy. Then using a CUDA kernel, we further distribute the rows of matrix A available with each MPI process among the GPU cores of the GPU associated with it to compute respective rows of AV. Figure 6 shows the parallel distribution of rows of matrix A for parallel computation of AV among three cluster nodes, each of which has 4 GPUs; thus, 12 MPI processors are launched.

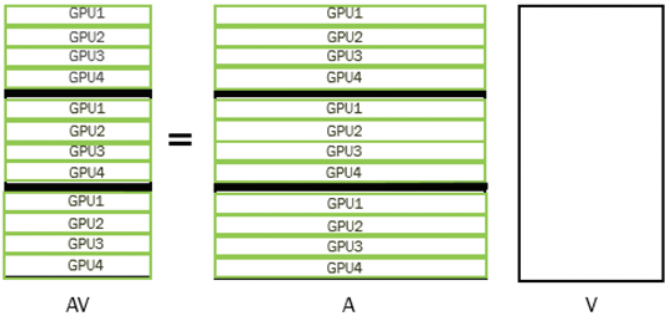


Figure 6. Equal Distribution of rows of A among 3 Nodes, each having 4 GPUs: A total of 12 MPI processors are launched.

3.4 Matrix Transpose-Matrix Multiplication

To obtain U^TAV, we word-wise XOR all those rows of matrix AV, for which the corresponding values in the transposed matrix U^T are 1. The challenging part in this implementation is to read the transpose of matrix U of size Nxm. To form the first row of U^T, we read the leftmost bit of each row of matrix U by masking. Subsequently, we read the second leftmost bit of each row and so on. Thus, we read a non-transposed matrix so that it will be equivalent to its transposed matrix, as shown in Fig. 7.

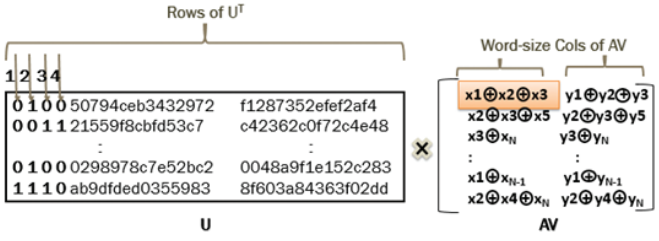


Figure 7. Matrix Transpose-Matrix Multiplication

3.5 Parallel Matrix Transpose-Matrix Multiplication

Matrix U is stored in memory, just like in Fig. 4. To parallelize this matrix transpose-matrix multiplication, we equally distribute matrices $\begin{bmatrix} \psi(\lambda) \\ \beta(\lambda) \end{bmatrix} = \begin{bmatrix} I_m & 0_m \\ 0_m & \lambda I_m \end{bmatrix} \begin{bmatrix} \tau_{hl} & \tau_{hr} \\ \tau_{ll} & \tau_{lr} \end{bmatrix} \begin{bmatrix} \psi(\lambda) \\ \beta(\lambda) \end{bmatrix}$ and AV row-wise and scatter it across all MPI processors using MPI_Scatter. We use these U_is and AV_is (where i is

equal to the MPI process rank) to generate $U_i^T AV_i$s at each MPI processor, as shown in Fig. 8. Each MPI processor invokes a CUDA kernel, which further distributes U and AV by assigning the indices of rows of U to the GPU cores using their threadId and blockId. Each $U_i^T AV_i$s are XORed using MPI_BXOR operation in MPI_Reduce to give the final required product $U^T AV$.

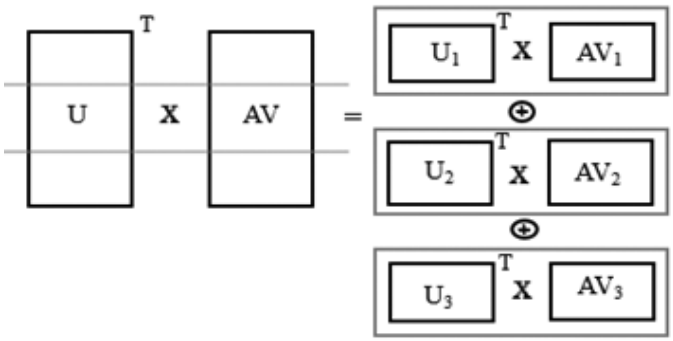


Figure 8. Equal Distribution of rows of U and AV among 3 MPI processors. Each $U_i^T AV_i$s computed in corresponding GPUs where i = MPI process rank. The 3 $U_i^T AV_i$s are then XORed to compute final $U^T AV$.

3.6 Iterative Sequence Generation

We have described how two types of multiplications involved in generating sequence (11) can be parallelized. However, we need to iteratively call these two multiplication CUDA kernels L times to generate the complete sequence. The only difference in each iteration is that the matrix AV computed in the earlier iteration becomes matrix V for the next iteration. Thus after every iteration, we gather all AV_is (where i is equal to the MPI process rank) generated with each MPI process to a single MPI process using MPI_Gather and then broadcast it as matrix V for the next iteration using MPI_Bcast.

3.7 Hybrid Parallel Solution Evaluation

The solution evaluation step in BWA also involves repetitive sparse matrix-matrix multiplications. Thus, we compute all these products in a hybrid parallel manner using MPI and CUDA on a multi-node multi-GPU platform as described in the previous subsection 3.3.

4. EXPERIMENTAL ANALYSIS

As described previously, the sparse linear solver comprises three main steps: (a) Sequence Generation, (b) Minimal Polynomial Matrix (MPM) Computation, and (c) Solution Evaluation. All the matrix-matrix and matrix transpose-matrix multiplications involved in the first and third steps are parallelized. As the first step involves the most number of such product computations, therefore it is highly scalable and gives the best results after parallelization. Detailed timings of the complete solution of some NFS matrices are compared in subsection 4.2 using up to 4 NVidia V100 GPUs of a DGX station. In subsection IV-B, we further discuss the scalability and performance of the parallel sequence generation part of the sparse linear solver across multiple multi-GPU nodes. We have used this tool for benchmarking an HPC cluster, which is a hybrid CPU-GPU cluster where each node is embedded with 4 Tesla V100 GPUs.

4.1 Complete Solver Performance on DGX Station

In our experiment, the input matrices are the filtered matrices obtained after the Sieving step of the Number Field Sieve for cryptanalysis of RSA-130, RSA-140, and RSA-170. Table 1 shows the detailed timings of all three steps of our sparse solver for the three input matrices over an NVidia DGX station. The DGX station in this experiment has 4 NVidia Tesla V100 GPUs and 256 GB system memory with 500 TFlops of supercomputing performance. Its clock speed is 2.20 GHz. Each V100 GPU has 5120 CUDA cores with 16 GB device memory and a peak memory bandwidth of 900 GB/s. The versions of the software used are OpenMPI 2.1.1 and CUDA 10.1.

We distributed the implementation of the sequence generation step of BWA over 1, 2, and 4 V100 GPUs using both MPI and CUDA. It can be noted from Table 1 that the time for the parallel solution evaluation step is approximately half of the time for the parallel sequence generation step. So we executed the parallel solution evaluation step on only 4 V100 GPUs of the DGX station. Both these steps involve multiple matrix-matrix and matrix transpose-matrix multiplications with some additional tasks. The cost of these matrix-matrix and matrix transpose-matrix multiplications dominates over all other tasks. Thus, the intense parallelization of these tasks gives us even better results with the increase in the matrix size.

Our experimental results show good speedup and parallel efficiency of the highly parallelizable sequence generation step after parallelization over up to 4 V100 GPUs. Table 1 shows the parallel efficiency and speedup on GPUs for three input matrices such that

$$PE(x) = \frac{T_1}{x \times T_x} \tag{12}$$

$$Speedup(x) = \frac{T_1}{T_x} \tag{13}$$

where, $PE(x)$ is the parallel efficiency on x GPUs for the given matrix and T_1, T_x are the execution times on 1 and x number of GPUs, respectively. As seen in table 1, the speedup compared to parallelization over 1 GPU goes from 1.6 to 2.8 when we parallelize the sequence generation step of BWA on 2 and 4 V100 GPUs, respectively. The parallel efficiency drops from 82% to 71% when using 2 and 4 V100 GPUs, respectively, for the cryptanalysis of RSA-170.

The minimal polynomial matrix computation in BWA requires large system memory, which is available on the DGX station. This step is executed sequentially, and the time required to find the minimal polynomial matrix grows quadratically with the dimensions of the matrix. The time complexity of the complete algorithm is $O(N^{2+\varepsilon})$ where $0 < \varepsilon \leq 1$.[15]

Table 1. Detailed timings of sparse solver over an nvidia dgx v100 station. The results are with $m = 128$ such that $U, V \in F_2^{N \times m}$ as in Eqn. (7)

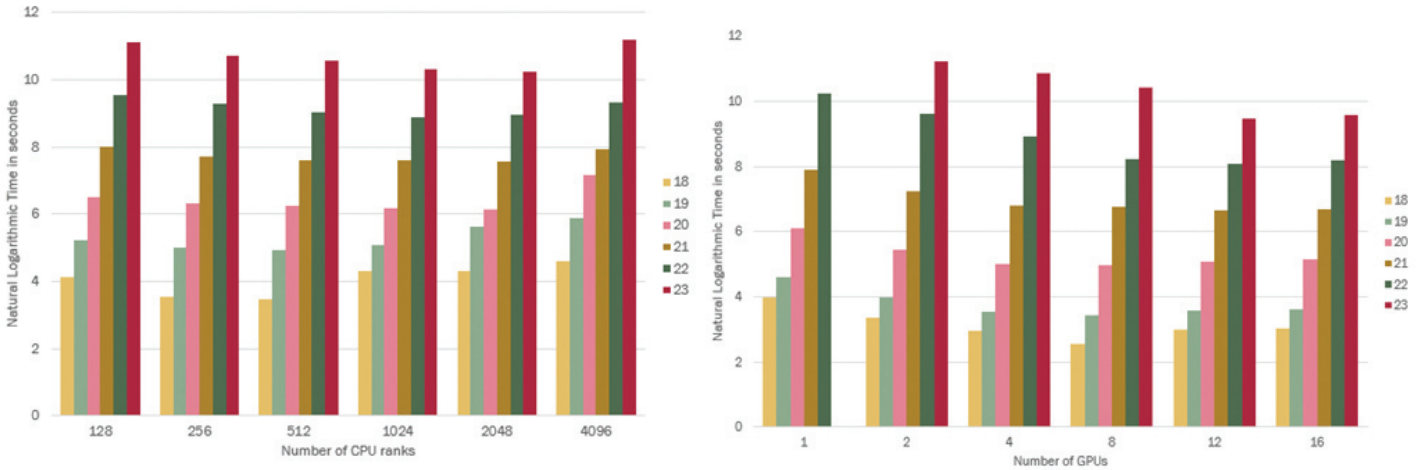
	Square Matrix Size (N as in (Eqn. 7))	Average nonzero entries per row	Parallel Sequence Generation (s)			MPM Computation (s)	Parallel Solution Evaluation (s)
			1 V100 GPUSs	2 V100 GPUs	4 V100 GPUs	CPU	4 V100 GPUs
RSA-130	2,097,152	92	10, 625	6,627 Speedup=1.6 PE = 80 %	3,968 Speedup=2.7 PE= 67 %	29,673 = 8h15m	1,926
RSA-140	4,194,304	82	38, 528	24,168 Speedup=1.6 PE = 80 %	14,336 Speedup=2.7 PE= 67 %	111,933 = 31h5m	7,056
RSA-170	8,388,608	117	206,336	125,888 Speedup=1.6 PE = 82 %	72,832 Speedup=2.8 PE= 71 %	398, 998 = 110h50m	36,216

4.2 Performance of Hybrid Parallel Sequence Generation on HPC Cluster

We have parallelized the sparse linear solver over GF(2) in two ways to compare its performance and scalability. In the first method, we parallelize the system over multiple CPU ranks on multiple cluster nodes using only MPI. In the second method, we parallelize it over multiple GPUs available on multiple cluster nodes using CUDA and MPI. Each cluster node is embedded with 4 Tesla V100 GPUs, 32 CPU cores, and 192GB DDR4 RAM. Each of these GPUs has 5120 CUDA cores, 900 GB/sec memory bandwidth, and 16 GB VRAM. The clock speed of the cluster is 2.00GHz, and the network used is 100Gbps EDR Infiniband Switch Fabric. The versions of the software used are CUDA 10.1 and OpenMPI 4.0.1.

Figure 9 shows the performance and scalability comparison between the parallelization methods of the parallel sequence generation step for six sparse linear systems. The sizes of chosen square matrices go from 2^{18} to 2^{23} rows. These sparse matrices with entries over GF(2) are generated randomly, where we distribute some dependent rows across the matrix to ensure that it is always solvable. The average number of nonzero entries per row is chosen to be 40. The difference in the range of timings of all six systems is so huge that we could not plot the observations in the same graph. Hence, natural logarithmic timings in seconds are used to plot the bar charts. Therefore, it can be understood that even a small difference in Y-axis in figure 9 signifies significant timing differences for the various number of CPU ranks or GPUs as plotted on X-axis. It can also be noticed from figure 9 that there is a point for each system when adding more processors (GPU or CPU) does not improve the overall performance. This means that the solver for every system is scalable for only that many GPUs or CPU ranks for parallelism.

Let us compare both the bar charts shown in Fig. 9. We can easily see that the least time taken to generate the sequence η^i as in Eqn. (11) for any given system on any number of CPU ranks (as shown on the X-axis of (a)) is approximately more than twice that of the least time taken by GPUs as shown in (b). For instance, for matrix size $2^{23} \times 2^{23}$, the least time to generate the parallel sequence using only MPI-level parallelization on 2048 CPU ranks spread over 64 nodes (32 CPU cores per node) is 27,421.88 seconds. In contrast, the time taken for the same matrix using hybrid parallelism over 12 GPUs spread across three nodes



(a) (b)

Figure 9. Solver Performance with: (a) CPU-level Parallelization (b) Multi-node Multi-GPU Parallelization (Natural Logarithmic Timings).

(each having 4 V100 GPUs) is only 13,075.2 seconds. Figure 10 and Fig. 11 show the exact timings for the parallel sequence generation step of the solver for the six square sparse systems of linear equations using both parallelization methods on the HPC cluster.

Figure 12 plots the parallel efficiency on GPUs for square matrices with 2^{18} to 2^{22} rows as in Eqn. (12). As discussed earlier, each node of the GPU cluster has 4 V100 GPUs; therefore, we use a single node to work on up to 4 GPUs, while two nodes for up to 8 GPUs, and for up to 12 GPUs, we use three nodes and for 16 GPUs we use four nodes of the cluster. It can be seen from figure 12 that for a matrix with 2^{18} rows, we obtain a parallel efficiency of 94.07 % on 2 V100 GPUs, which drops to 72.29 % and to 52.28 % on using 4 and 8 V100 GPUs, respectively.

Also, from Fig.12, it can be seen that for a matrix with 2^{22} rows, we obtain a parallel efficiency of 94.4 % on 2 V100 GPUs, which is maintained to be 94 % and 94.4 % at 4 and 8 V100 GPUs, respectively. This parallel efficiency drops to 72.74% at 12 V100s and further drops to just 49.34% at 16 GPUs. This proves that if we increase the problem size, we can saturate more GPUs, i.e., 12 or 16 V100s. We could not calculate the parallel efficiency for 2^{23} problem size because it took considerable time to solve on a single GPU.

The hybrid parallel sequence generation using both parallel implementation methods can be later verified by using those output sequences as inputs for the subsequent modules of the Block Wiedemann Algorithm. If the solution of the input matrix is obtained, then we can say that the parallel implementation is verified. Hence, we could use this parallel sequence generation tool for the benchmarking of the given hybrid cluster.

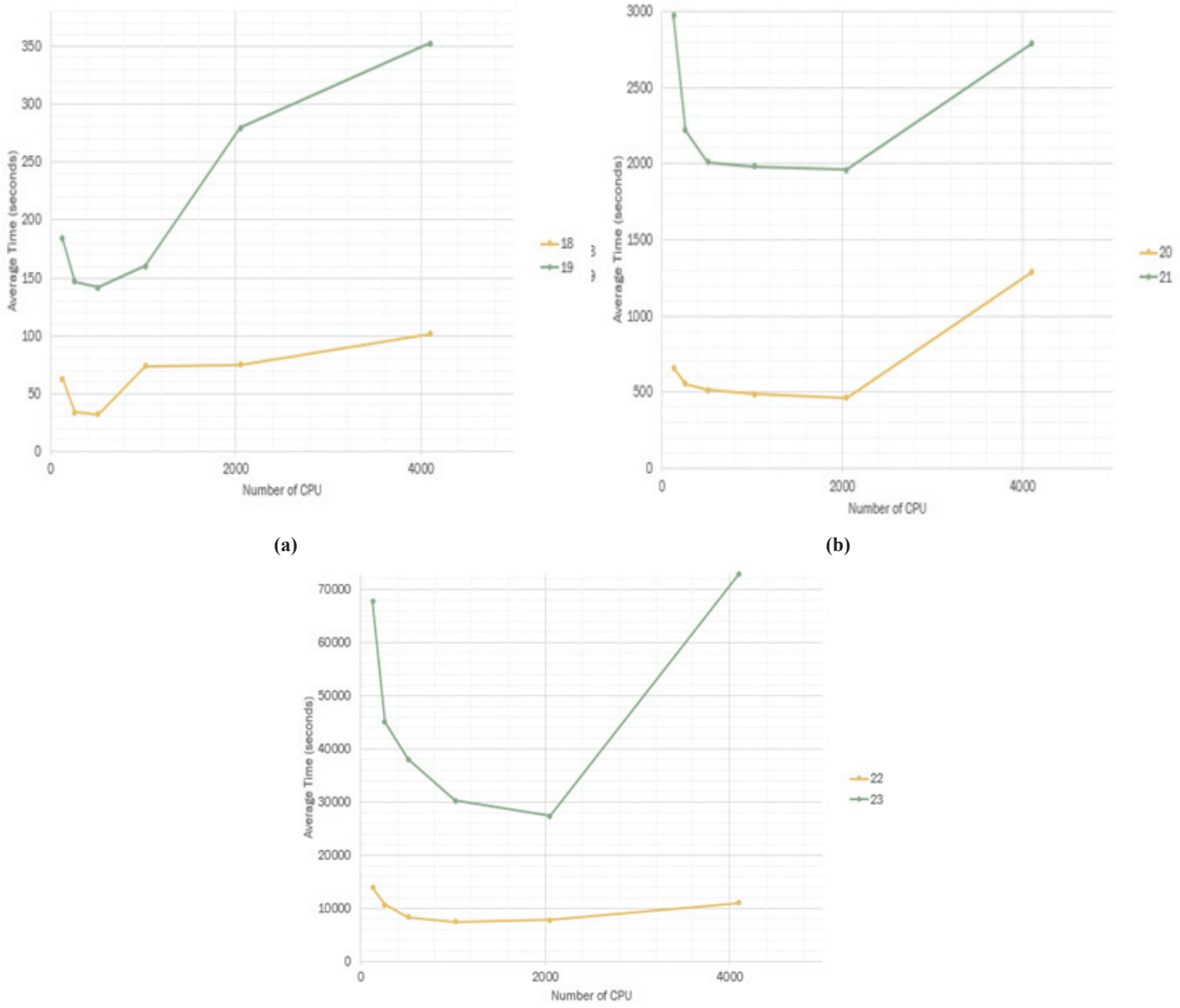


Figure 10. Solver Performance with CPU-level Parallelization for square matrices of size: (a) 2^{18} and 2^{19} (b) 2^{20} and 2^{21} and (c)2^{22} and 2^{23}.

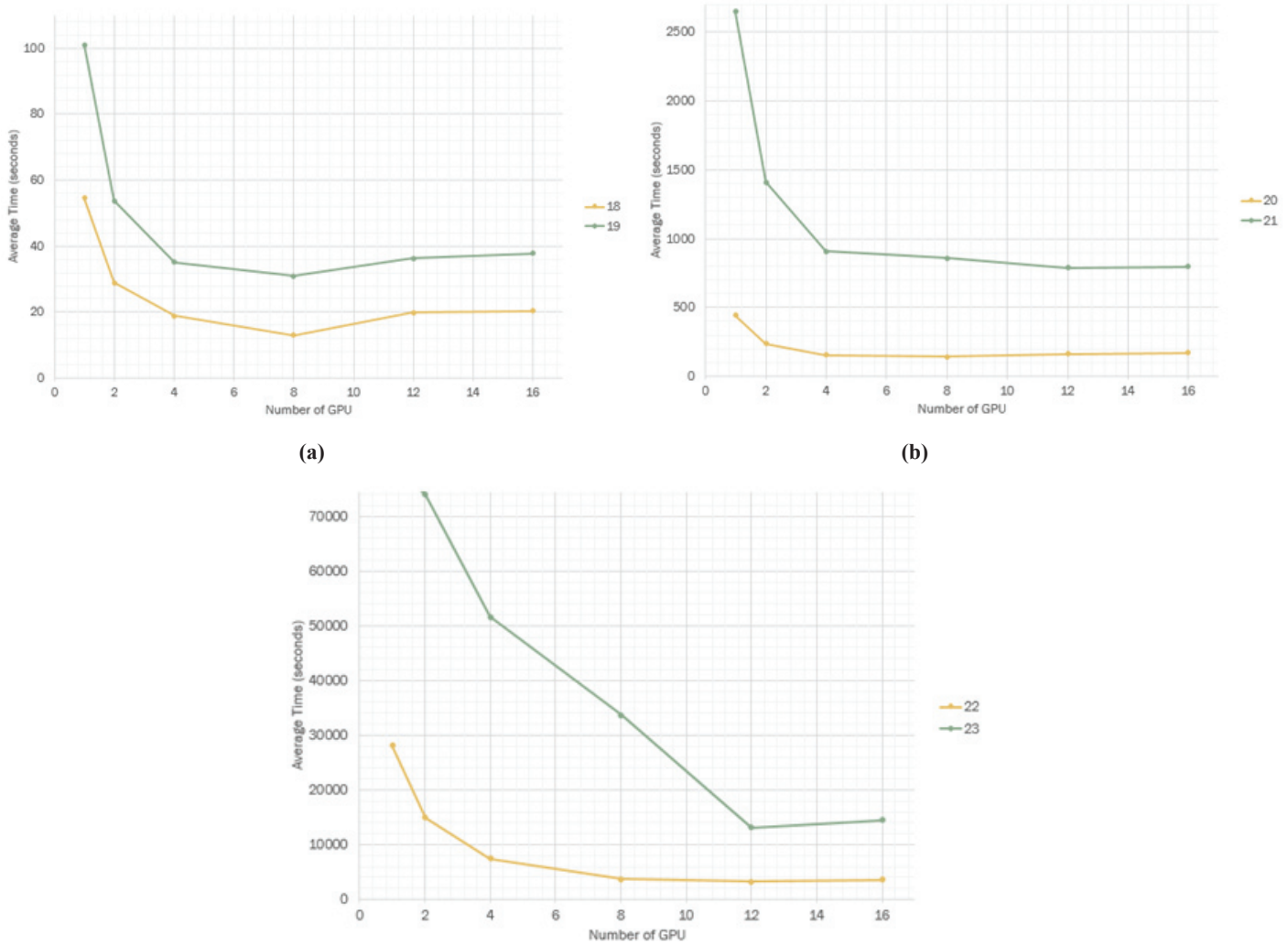


Figure 11. Solver Performance with multi-node multi-GPU (V100) parallelization for square matrices of size: (a) 2^{18} and 2^{19} (b) 2^{20} and 2^{21}, and (c) 2^{22} and 2^{23}.

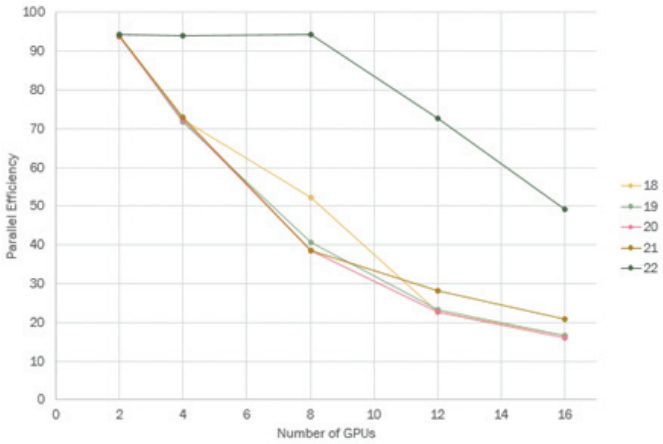


Figure 12. Parallel efficiency on GPUs for square matrices of size 2^{18} to 2^{22}.

5. CONCLUSIONS

This paper demonstrates the step-wise hybrid parallel implementation of the Block-Wiedemann Algorithm(BWA) to find the solution of a large sparse system of linear equations over GF(2). This linear solver can be used in cryptanalysis applications like cryptanalysis of RSA using Number Field Sieve. The solver exploits multiple levels of parallelism on a multi-node, multi-GPU hybrid cluster. CUDA and MPI are used for the parallelization of the solver. The experimental results compare the parallelization of the solver over multiple CPU ranks spread over multiple nodes with its parallelization over multiple GPUs (V100s) spread over multiple cluster nodes. The timings and performance obtained by GPU-accelerated parallelization of the solver are much better than the other parallelization method. It is observed that the least time taken by GPUs to parallelize a task is less than half of the time taken by any number of CPU ranks. The proposed solver shows a parallel efficiency of around 94 % on 2, 4, and 8 Volta V100 GPUs. This parallel efficiency drops to 72.74% and 49.34% at 12 V100s and 16 V100 GPUs, respectively. Thus, we can say that the solver is highly scalable over multiple GPUs spread across multiple cluster nodes and effectively utilizes the device's memory bandwidth.

We also show the solution of three sparse systems for cryptanalysis of RSA-130, RSA-140, and RSA-170, where the highly parallelizable modules of Block Wiedemann algorithm give a speedup of 2.8 after parallelization on 4 V100 GPUs of NVidia DGX station as compared to that over 1 GPU. Our future

work includes investigating methods to offload those steps of the Block Wiedemann Algorithm (Minimal Polynomial Matrix Computation) to the GPU, which have not been parallelized before.

REFERENCES

1. Briggs, M. An introduction to general number field sieve. M.S. Thesis, Virginia Polytechnic Institute, Blacksburg, Virginia, 1998.
2. Rivest, R.L.; Shamir, A. & Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. of ACM*, 1978, **21**(2),120-126.
3. Fevgas, A.; Daloukas, K.; Tsompanopoulou, P. & Bozanis, P. Efficient solution of large sparse linear systems in modern hardware. 6th International Conference on Information, Intelligence, Systems, and Applications (IISA), 2015.
4. Pomerance, C. & Smith, J.W. Reduction of huge, sparse matrices of finite fields via created catastrophes. *Experiment. Math.*, 1992, **1**(2), 89-94.
5. Wiedemann, D. Solving sparse linear equations over finite fields. *IEEE Trans. Inform. Theory,* 32, 1986, 54-62.
6. LaMacchia, B. A., & Odlyzko, A. M. Solving large sparse linear systems over finite fields. *Adv. in Cryptology —* CRYPTO '90, 1990, vol. 537 of Lecture Notes in Comput. Sci., Springer-Verlag, 109–133.
7. Coppersmith, D. Solving Linear Equations over GF(2): Block Lanczos Algorithm. *Linear algebra and its Appl.*, 1993, **192**, 33-60.
8. Coppersmith, D. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.*, 1994, 62/205, 333-350.
9. Villard, Gilles. Further analysis of coppersmith's block wiedemann algorithm for the solution of sparse linear systems. ISSAC 1997.
10. Villard, Gilles. A study of coppersmith's block wiedemann algorithm using matrix polynomials. Technical report, LMC-IMAG, 1997, REPORT 975 IM.
11. Thomé, Emmanuel. Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. Proceedings of the 2001 international symposium on Symbolic and algebraic computation, London, Ontario, Canada, July 2001, p.323-333.
12. Thome´, Emmanuel. Subquadratic computation of vector generating polynomials and improvement of the block wiedemann algorithm. *J. of Symbolic Comput.,* 2002, **33**(5), 757-775.
13. Giorgi, Pascal & Lebreton, Romain. Online order basis algorithm and its impact on the block Wiedemann algorithm. Proceedings of the 39th International symposium on symbolic and algebraic computation, Kobe, Japan, July 23-25, 2014.
14. Kaltofen, Erich. Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems. Mathematics of Computation, April 1995, vol. 64, nr. 210, pp. 777-806.
15. Kaltofen, E., & Lobo, A. Distributed matrix-free solution of large sparse linear systems over finite fields. algorithmica, 1999, **24**, pp. 331-348.
16. Vialla, Bastein. Block Wiedemann algorithm on multicore architectures. ACM communications in computer algebra, Association for Computing Machinery (ACM), 2014, **47** (3/4), pp.102 –103.
17. Steel, Allan K. Direct solution of the (11,9,8)-MinRank problem by the block Wiedemann algorithm in Magma with a Tesla GPU. In PASCO'15, ACM. Pp. 2-6, 2015.
18. Nickolls, J., Buck, I., Garland, M. & Skadron, M. Scalable parallel programming with Cuda. Queue, Mar. 2008, **6**(2), pp. 40–53.
19. Nvidia gpudirect. https://developer.nvidia.com/gpudirect. Accessed on 25 October 2021
20. Forum, M.P. Mpi: A message-passing interface standard. Knoxville, TN, USA, 1994, Tech. Rep.
21. Massey, J.L. Shift-register synthesis and BCH decoding. IEEE Trans. Inf. Theory IT-15, 1969,122-127.
22. Kaltofen, E. & Yuhasz, G. On the matrix Berlekamp-Massey algorithm. ACM Trans, September 2013, Algor. 9, 4, Article 33, 24 pages.

CONTRIBUTORS

Ms Shruti Rawal joined DRDO as a Scientist at DEAL, Dehradun in 2010. She is currently working in Scientific Analysis Group, New Delhi since 2012. She has received her BE in Computer Science and Engineering from Delhi College of Engineering, India in the year 2010. At present, her areas of research include cryptology, linear algebra, algorithms, high performance computing and AI, ML, DL in system security.

In the current study, she carried out the development of initial concept, implementation and generation of results. She also carried out the manuscript preparation and review.

Dr Indivar Gupta completed his PhD from IIT Delhi, India. He has been working as a scientist in Scientific Analysis Group, DRDO since 2000, and has research contributions in various areas related to cryptography and information security. Presently, his areas of research include computational algebra, number theory, cryptography, information security, and high-performance computing.

His contributions in the current study involve the initial concept and manuscript review.