Block Lanczos to Solve Integer Factorization Problem Using GPU's

Harish Malla*, Vilas SantKaustubh, Rajasekharan Ganesh, and Padmavathy R.

National Institute of Technology
Warangal

Abstract. Public key cryptography is based on some mathematically hard problems, such as Integer Factorization and Discrete Logarithm problems. The RSA is based on Integer factorization problem. Number Field Sieve is one of the popular algorithms to solve these two problems. Block Lanczos algorithm is used in the linear algebra stage of Number Filed Sieve method for Integer Factorization. The algorithm solves the system of equations Bx=0 for finding null spaces in the matrix B. The major problems encountered in implementing Block Lanczos are storing the entire sieve matrix and solving the matrix efficiently in reduced time. Implementations of Block Lanczos algorithm have already been carried out using distributed systems. In the current study, the implementation of Block Lanczos Algorithm has been carried out on GPUs using CUDA C as programming language. The focus of the present work has been to design a model to make use of the high computing power of the GPUs. The input matrices are very large and highly sparse and so stored using coordinate format. The GPU on-chip memories have been used to reduce the computation time. The experimental results were obtained for the following problems; RSA100, RSA110, RSA120. From the results it can be concluded that a distributed model over GPUs can be used to reduce the iteration times for Block Lanczos.

Keywords: Public Key cryptography, RSA, Block Lanczos, GPUs.

1 Introduction

Public key cryptography is based on some mathematically hard problems. The popular RSA is based on integer factorization and the counterparts ElGamal and Diffie-Hellman are based on discrete logarithm problem. In number theory, integer factorization problem is to factor the given composite number into its factors. The problem is found to be hard when the factors are big primes. The best known method to solve the above problem is Number Field Sieve.

The Number Field Sieve consists of two steps, such as sieving and solving. The sieving phase generates a large and sparse matrix called as sieve matrix. The solving phase, first reduces the large size matrix into small and still sparse matrix and later solves the linear system of equations.

*Corresponding author.

N. Meghanathan et al. (Eds.): CCSIT 2012, Part I, LNICST 84, pp. 520–529, 2012.

The solving phase is the main bottle-neck in the overall process. In the literature, many algorithms are reported. The method proposed by Lanczos is widely known and attempted method, since it needs less memory and easily adoptable for large and sparse matrices.

Block Lanczos algorithm which is a modified version of Lanczos algorithm used in the linear algebra stage of Number Field Sieve (NFS) is proposed in [1]. This algorithm is one of the ideal candidates for parallelization. The algorithm uses subspaces instead of vectors for solving the sparse matrix generated in sieving stage, for finding null spaces. The subspaces are represented using matrices. The parallel implementation of Block Lanczos using Mondriaan partitioning for sparse matrices is discussed in [2]. In this he discussed about the global-local indexing mechanism, vector partitioning, sparse matrix partitioning, sparse matrix-vector multiplication, AXPY operations and dense vector inner product computation. Coppersmith et.el., discussed how Block Lanczos is much competitive than Gaussian Elimination for solving linear system of equations [4]. The paper also discusses that the block operations performed in Block Lanczos reduces the 32 matrix-vector operations to one. Nathan Bell et. el., reported the different format of representation for sparse matrix to store and perform matrix operations on them efficiently [6]. The different formats given by the author are DIA, ELL, CSR, COO, hybrid format. The use of COO format shows very little variance in efficiency over different data and applications. They also discussed about how matrix operations can be performed efficiently on different matrix formats that have been discussed in their previous work [7].

In the present study Block Lanczos is implemented on GPUs. The GPUs have larger number of cores on a chip when compared to CPUs. Also the Arithmetic Logical Units (ALUs) in case of GPUs are much more than in CPUs. Many-coreprocessors, especially the GPUs, have high floating-point performance. As discussed in [4], Block Lanczos algorithm is one of the ideal candidates for parallelization. Also from [6] and [7] it can be inferred that the sparse matrix operations of Block Lanczos can be performed efficiently on GPUs using CUDA. These ideas provided the motivation for implementing the Block Lanczos algorithm on GPUs using CUDA C.

1.1 Integer Factorization

There are different methods for Integer Factorization like continued fraction method, quadratic sieve, and number field sieve. Integer factorization algorithms require several nonzero vectors x belonging to Galoise field $(GF(2)^n)$ such that a system of equations $Bx= 0$ is obtained, where B is a given m×n matrix over the field GF(2). This matrix B is called sieve matrix and is usually very large and highly sparse with $m < n$. Suppose, an integer M is to be factored, the quadratic sieve method finds congruence's between a_j^2 and product of p_i raised to some exponents b_{ij}, modulus M. Here p_i are primes or -1 and the b_{ij} are exponents, which are mostly zeroes. The quadratic sieve method then tries to find S $\in$ { 1, 2, - - ,n } such that both sides of the congruence.

$$\prod_{j\in S} a_j^2 = \prod_{j\in S} \prod_{i=1}^{m} p_i^{b_{ij}} \text{ (mod M)} \quad (1)$$

are perfect squares. The left hand side product is automatically a square, but the right hand side product is a square only if all exponents are even, i.e., if$\prod_{j \in S} b_{ij} \equiv 0 \pmod 2$ for $1 \leq i \leq m$. This is equivalent to the system of equations $Bx \equiv 0 \pmod 2$, where $B = (b_{ij})$, and $x = (x_j)$, and where $x_j = 1$ if $j \in S$ and $x_i = 0$ if $j \in S$.

The matrix B that arises in the sieving stage of factoring has a specific structure. This matrix is extremely sparse, with around 60-80 non-zero entries per row. It is divided into dense block and sparse block. The dense parts in columns correspond to smaller primes and very sparse parts in columns correspond to larger primes. The best way to solve the matrix is the combination of Structured Gaussian elimination with Lanczos or Wiedemann. The Structured Gaussian elimination algorithm is applied first to reduce the large matrix to a comparatively smaller matrix which is still sparse. This step is called filtering. After getting the filtered matrix, the Block Lanczos or Wiedemann iterations can be applied efficiently on a smaller matrix.

Wiedemann is found to be slower compared to Lanczos and hence, Block Lanczos algorithm is chosen as the best method for finding the required linear dependencies [5, 8, 9, 13].

1.2 Block Lanczos

The Lanczos method is used for solving linear equations Ax=b for finding eigenvectors. But the algorithm fails in GF(2) due to the self orthogonality property of the binary vectors. To eliminate this problem, a set of vectors(representing subspaces) instead of a single vector were used. Each subspace is represented by a matrix. The matrix-vector products in Lanczos are replaced by matrix-matrix products in $GF(2^n)$. The matrix A can be applied to N (generally 32 or 64) different vectors in $GF(2^n)$ at once using bitwise operators. This modification is called Block Lanczos.

The Block Lanczos method, which is a variation of the Lanczos procedure uses block versions of the three-term recursions. As a general thrust, block algorithms substitute matrix block multiplies and block solvers for matrix-vector products and simple solvers in unblocked algorithms. In other words, higher level block arithmetic operations are used in the inner loop of the block algorithms. This decreases the I/O costs essentially by a factor of the block size. In addition, the block algorithms are generally more robust and efficient for matrices with multiple or closely clustered eigen values.

Suppose A is a symmetric $n \times n$ matrix over the field GF(2). The Block Lanczos algorithm produces a sequence of subspaces $\{W_i\}_{i=0}^{m-1}$ of GF (2^n) which are pair wise A−orthogonal. The properties of vectors w_i in the Lanczos algorithm ensures the finding of a solution vector. These properties were generalized to a A-orthogonal subspaces W_i to ensure a solution in the modified sequence of iterations.

The condition $w_i^T A w_i \neq 0$ in Lanczos is replaced by a requirement that no non-zero vector in W_i be A−orthogonal to all of W_i. The subspace W satisfying this property is said to be A− invertible.(A subspace $W \in K^n$ is said to be A-invertible if it has a basis W of column vectors such that $W^T AW$ is invertible). It will have a basis W of column vectors such that $W^T AW$ is invertible.

The property of being A-invertible is independent of the choice of basis, since any two bases for W are related by an invertible transformation. If W is A-invertible, then any u $\in K^n$ can be uniquely written as v + w where w ∈ W and WAv = (0). The generalization to subspaces can be given as,W_i is A-invertible , $W_j^T AW_i = \{0\}$ for i ≠j , and AW $\underline{C}$ W, where $W=W_0+W_1+...+W_{m-1}$.

Assuming the above statement, given b ∈ W, an x ∈ W can be constructed such that AX = b. Let x = $\sum w_j$, where $w_j \in W_j$ is chosen so that Aw_j - b is orthogonal to all of W_j. If the columns of W_j form a basis for W_j, then x can be given as

$$x = \sum_{j=0}^{m-1} W_j (W_j^T A W_j)^{-1} W_j^T b \tag{2}$$

Now fix N > 0. At certain step i, an n x N matrix Vis generated, which is A-orthogonal to all earlier W_j. The initial V_0 is taken to be arbitrary. W_i is selected using as many columns of V_i as can be possible, subject to the requirement that W_i be A-invertible. The Lanczos iterations are replaced by

$$W_i = V_i S_i, \tag{3}$$

$$V_{i+1} = AW_i S_i + V_i - \sum_{j=0}^{i} W_j C_{i+1,j} \quad (i \geq 0) \tag{4}$$

$$w_i = \langle W_i \rangle \tag{5}$$

Iterations are stopped when $V_i^T AV_i = 0.$ suppose this occurs for i = m. In the above equation S_i is an N x N_i projection matrix which has been chosen so that $W_i^T AW_i$ is invertible while making $N_i \leq N$ as large as possible. The matrix S_i should be zero except for exactly one 1 per column and at most one 1 per row. These ensure that $S_i^T S_i = I_{N_i}$ and that $S_i S_i^T$ is a sub matrix of I_N reflecting the vectors selected from V_i. The equation V_{i+l} tries to generalize while ensuring that $W_j AV_{i+1} = \{0\}$ for j ≤i, if the earlier W_j exhibits the desired property of A-orthogonality. Then the following expression can be used

$$C_{i+1,j} = \left(W_j^T A W_j\right)^{-1} W_j^T A (AW_i S_i^T + V_i) \tag{6}$$

The terms V_i -$W_i C_{i+l,i}$ select all the columns of V_i not used in W_i; those columns are known to be A-orthogonal to W_0 through W_{i-1}, and the choice of $C_{i+l,i}$ adjusts them so they are A-orthogonal to W, as well. Without the V_i term, rank (V_{i+l}) would be bounded by rank($AW_i S_i^T$) ≤ rank(V_i), and would soon drop to zero. After further simplification,

$$V_{i+1} = AV_i S_i S_i^T + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1} \tag{7}$$

2 Review of Basic Operations Involved in Block Lanczos Algorithm

2.1 Block Lanczos Algorithm

The Block Lanczos algorithm is used to solve the linear system of equations Bx=0 for finding the null spaces in matrix B. This is achieved by decomposing $GF(2)^n$ into several subspaces of dimension almost N which are pair wise orthogonal with respect to the symmetric n x n matrix $A = B^T B$**.** In each of the iteration the matrices B and B^T are applied to an n x**N** matrix and then a few supplementary operations are performed [12].

The pseudo-code for the algorithm is given as Algorithm 1.

Algorithm 1: Block Lanczos

Input:Matrices B of size $n_1 \times n_2$ and Y of size $n_2 \times N$

Output:The matrices X and V_m

1: Initialization: X = 0 2: $V_0 = AY = B^T * (BY)$

3: $C_0 = V_0^T A V_0 = V_0^T (B^T B) V_0 = (BV_0)^T * BV_0$ 4: Compute $AV_0 = B^T * (BV_0)$

5: i = 0

6: while $C_i \neq 0$ do

7: compute $V_i^T A^2 V_i = (AV_i)^T * (AV_i)$

8: $[W_i^{inv}, SS_i^T] = \text{Inverse}\ (V_i^T A V_i, SS_{i-1}^T, N)$

9: X = X + Vi * $(W_i^{inv} * (V_i^T * V_0))$ 10: $Z_i = (V_i^T A^2 V_i) * (SS_i^T) + C_i$

11: $D_{i+1} = I_N - W_i^{inv} (Z_i)$ 12: $E_{i+1} = -W_{i-1}^{inv} (C_i * SS_i^T)$

13: $F_{i+1} = -W_{i-2}^{inv} (I_N - C_{i-1} + W_{i-1}^{inv})(Z_{i-1})\ SS_i^T$

14: $V_{i+1} = AV_i S_i S_i^T + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}$

15: compute BV_{i+1}

16: $C_{i+1} = V_{i+1}^T A V_{i+1} = (BV_{i+1})^T * (BV_{i+1})$ and $AV_{i+1} = B^T * BV_{i+1}$

17: i = i + 1

18: end while

19: Return X and V_m

The experimental results are carried out on a machine with following characteristics**:-** A system with following configurations

- Intel i7 740QM processor
- 4GB DDR3 RAM
- NVIDIA Ge Force 330M GPU (1 GB)
- Another system for debugging using SSH portal.

Software Requirements

- Operating System: Ubuntu 10.10
- NVIDIA Developer Driver for Linux (260.19.26)
- CUDA 3.2 Toolkit
- SSH server on system with GPU and SSH client on another system

3 Implementation

3.1 Distribution of Data

The typical matrices for which the Block Lanczos algorithm is applied are very large and mostly sparse. Taking advantage of the latter, the matrix can be stored in a way that is much more clever than just explicitly storing every entry in the matrix. Storing each entry is already infeasible for a matrix with n of size 500,000, since that would need about 32 GB of RAM to store it. Note that this requirement is much too large to be fulfilled by the RAM of today's typical machine. Also the typical n may be two to twenty times larger than this, increasing the RAM requirement substantially.

The matrix corresponding to the system of equations that is generated from the number field sieve follows a very predictable pattern. The matrix that is obtained from sieving stage is stored by collections of columns, each collection may form a dense block or a sparse block. The number field sieve (much like the quadratic sieve) uses three factor bases (rational, algebraic, and quadratic characters) in sieving as part of the process of factoring a large number. Dense rows of the matrix correspond to the smaller primes, and sparse rows correspond to larger primes. These first few rows are called dense since they have relatively higher non-zero entries. Once sparsity of the matrix increases, it is more worthwhile to store the locations of these entries rather than storing all the particular entries.

The sizes of the sieve matrices are too huge. Hence sometimes it may not be possible to store the entire matrix on a single device. Hence the need to keep the matrix on several devices is arising. So storing the matrix on many number of devices and dealing with them efficiently is necessary. Distributing the matrix uniformly over the devices is necessary so as to distribute the computation load uniformly.

Therefore the matrix B is stored in co-ordinate form with each entry giving the indices of row and column to which then on-zero element belong. The use of coordinate format greatly reduces the memory requirements, which is directly proportionate to

number of non-zero entries in the matrix. Also an advantage of storing the matrix in coordinate format was that no extra memory was used to store the transpose of memory. Only a slight change in the logic for multiplication of transpose of matrix was required. The matrix is being stored in global memory on GPU because this matrix is used in all the iterations twice. The matrix is also divided into strips of size which depends on shared memory restrictions of GPU. The strip size of the matrix depends on the shared memory because the output of the multiplication of the matrix with column vector is being stored in shared memory which greatly reduces the write access time costs. The offsets of these strips have also been stored in global memory on GPU. These offsets are used in relation to transpose of matrix B. The dense part of matrix is stored as an array of bit strings. The data distribution can be found in [3]. Programming massively parallel processors is reported in [10] and the GPU programming is illustrated in [11]

3.2 Basic Functions Implemented Using CUDA C on GPUs

The following are some of the main operations to be computed to carry out the block lanczos algorithm.

Random Vector Y, Computation of V_0, Computation of C_i, Computation of V_{i}, $^{T}A^{2}V_i$ and Computation of V_{i+1}

3.2.1 The List of Functions Written in CUDA C

The algorithm is implemented in CUDA C. The implementation consists of different functions on device based on operations that are to be performed in the algorithm. The word-size N is 32 bits. A brief description of functions is given below:

rand_gen( ) This function randomly generates vector Y of size of n x N which is represented as n words. To increase the randomness of the data used Y is divided into 3 blocks and each block uses a different seed.

strip_mul ( ) This function performs the operation in1 for a block of matrix B of size m x n with column vector Y of size n words on the device. The output of this function is a partial matrix product that is passed to the reduce() function to get final product.

reduce( ) This function takes the partial products and performs the XOR operation on them to get the final result.

productBY( ) This function performs the multiplication of matrix B (size m x n) with vector Y (size n x N) by calling the function strip_mul( )for each strip in B. It also makes use of pthreads and parallely reduces the outputs of strip_mul( ) for the previous strip. The same function is used for calculating product BV (size m x N) during each iteration of the algorithm.

dense_mul( ) This function computes the product of the dense block of matrix B and vector Y which is later on combined with product of sparse block of B with Y.

strip_trans_mul( ) This function performs the operation in2 for block of matrix B of size m x n with vector V of size n words. The output of this function is a partial matrix product that is passed to the reduce( ) function to get final product.

productV( ) This function performs the multiplication of matrix B^T with vector V by calling the function strip_trans_mul( ) for each strip. The same functionis used for calculating the product AV during each iteration of the algorithm.

dev_trans_dense_mul( ) This function computes the product of the dense block of matrix B^T(size m × n) and vector Y (sizen × N) which is later on combined with product of sparse block of B^T with Y.

device_mul_NnnN( ) This function performs operation op2 for matrix AV and its transpose.

dev_mul_nNNN( ) This function performs operation op1.

mul_NNNN( ) This function performs the operation op3.

inverse( ) This function implements the algorithm given in algorithm 4 to compute W_i^{inv}, SS_i^T

The inputs were generated using msieve 1.48 which is developed for factoring RSA numbers. Msieve is implemented on pthreads and MPI that make use of multicore architecture of CPUs. These matrices that were obtained from Msieve were in column-major order format for the sparse part and in bit string representation for the dense part of the sparse matrix. These matrices were later on reordered to suit the implementation model that was followed in the current work. Block Lanczos was carried out for three different matrices that are obtained in three different RSA numbers. Details of which are tabulated in following table.

Table 1. Input matrices dimensions.

Input Matrix	Number of rows	Number of columns
RSA100	186821	186999
RSA110	346763	346940
RSA120	736255	736431

The operations were carried out by varying the number of blocks in the grid and also number of threads in each block. The results obtained were as follows:

Table 2. Experimental results obtained by varying the number of blocks keeping the number of threads constant

Input Matrix	Number of Blocks (Number of threads=512)				
	8	16	32	64	128
RSA100	1922	1924	1866	1855	1941
RSA110	3303	3213	3129	3102	3271
RSA120	6053	5994	5865	5847	6175

These results are single iteration times in milliseconds.

Table 3. Results obtained on varying the number of threads keeping the number of blocks constant

Input Matrix	Number of threads (Number of blocks=64)			
	64	128	256	512
RSA 100	2077	1996	2007	1855

These results are also single iteration times in milliseconds.

It can be seen that by varying the number of blocks and threads in all the three cases best results are obtained by keeping number of blocks as 64. Also, the number of threads to get the least time possible for this implementation is 512.

From the results above, it can be concluded that the single iteration times increase by a factor of less than 2 for each increase of 10 digits in the RSA number factored. Also it was found that the optimum time was reached when the program was executed using 64 blocks each of 512 threads. In this way it was found that by focusing on the architecture of device an efficient implementation of Block Lanczos algorithm on GPUs can be carried out.

References

[1] Montgomery, P.L.: A Block Lanczos Algorithm for Finding Dependencies over GF(2). In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 106–120. Springer, Heidelberg (1995)

[2] Bisseling, R.H., Flesch, I.: Mondriaan Sparse Matrix Partitioning for Attacking Cryptosystems by a Parallel Block Lanczos Algorithm -a case study. In: Proceeding of International Conference on Parallel Computing: Current and Future Issues of High-End Computing, ParCo 2005. NIC Series, vol. 33, pp. 819–826 (2006) ISBN 3-00-017352-8

[3] Vastenhouw, B., Bisseling, R.H.: A Two Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication, Preprint 1238, Department of Mathematics, Utrecht University, Utrecht, Netherlands (May 2002)

[4] Coppersmith, D.: Solving Linear Equations over GF(2): Block Lanczos Algorithm. Linear Algebra Application 192, 33–60 (1993)

[5] Cullum, J.K., Wiloughby, R.A.: Lanczos Algorithm for Large Symmetric Eigenvalues Computation. Theory, vol. 1. Birkhauser, Boston (1985)

[6] Bell, N., Garland, M.: Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In: Proceedings Supercomputing 2009 (November 2009)

[7] Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report, NVR-2008-004 (December 2008)

[8] Coppersmith, D.: Solving Homogenous Linear Equations over GF(2) via Block Wiedemann Algorithm. Mathematics of Computation 62, 333–350 (1994)

[9] Montgomery, P.L.: Square roots of product of algebraic. In: Gautschi, W. (ed.) Proceedings of Symposia in Applied Mathematics. Mathematics of Computation 1943-1993: a Half Century of Computational Mathematics, pp. 567–571. American Mathematical Society (1994)

[10] Sanders, J., Kandrot, E.: CUDA by Example-An Introduction to General Purpose GPU programming (2011)
[11] Kirk, D., Hwu, W.-M.W.: Programming Massively Parallel Processors- A Hands on Approach (2010)
[12] Ramanjulu, M.: Parallel Computations for the Matrix stage of Integer factorization, M.Tech thesis (2008)
[13] LaMacchia, B.A., Odlyzko, A.M.: Solving Large Sparse Linear Systems Over Finite Fields. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 109–133. Springer, Heidelberg (1991)