Implementation of Block Lanczos Algorithm on GPU

A Project submitted in partial fulfillment of requirements for the degree of

Bachelor of Technology

in

Computer Science and Engineering

Submitted by

Malla Harish (CS7745)

Ganesh Rajasekharan (CS7726)

Sant Kaustubh Vilas (CS7761)

Contents

[ABSTRACT 3](#_Toc375409781)

[Chapter 1 Introduction 4](#_Toc375409782)

[1.1 Introduction 4](#_Toc375409783)

[1.2 Notations 6](#_Toc375409784)

[Chapter 2 Lanczos 7](#_Toc375409785)

[2.1 The Lanczos algorithm 7](#_Toc375409786)

[2.2 Failure of Lanczos in GF (2) 9](#_Toc375409787)

[2.3 Block Lanczos 10](#_Toc375409788)

[2.4 Simplifying Block Lanczos 12](#_Toc375409789)

[2.5 Finding the Null Space of the Sieve Matrix 14](#_Toc375409790)

[2.6 Basic operations with bitwise operators 16](#_Toc375409791)

[Chapter 3 GPUs 18](#_Toc375409792)

[3.1 History 18](#_Toc375409793)

[3.2 Design 19](#_Toc375409794)

[Chapter 4 CUDA 21](#_Toc375409795)

[4.1 Threads and Kernels 21](#_Toc375409796)

[4.1.1 Thread Cooperation 21](#_Toc375409797)

[4.2 Memory Types 22](#_Toc375409798)

[4.3 Atomics 23](#_Toc375409799)

[4.4 Page-Locked Host Memory 23](#_Toc375409800)

[4.4.1 Zero Copy Host Memory 24](#_Toc375409801)

[Chapter 5 Implementation Of Block Lanczos On GPU 25](#_Toc375409802)

[5.1 Data Distribution 25](#_Toc375409803)

[5.2 Operations 26](#_Toc375409804)

[5.2.1 Computation of Outer Product(op1) 26](#_Toc375409805)

[5.2.2 Computation of Outer Product(op2) 27](#_Toc375409806)

[5.3 Implementation 29](#_Toc375409807)

[5.3.1 Random Vector Y 29](#_Toc375409808)

[5.3.2 Computation of *V0* 29](#_Toc375409809)

[5.3.3 Computation of *Ci* 29](#_Toc375409810)

[5.3.4 Computation of *ViTA2Vi* 29](#_Toc375409811)

[5.3.5 Computation of V*i+1* 29](#_Toc375409812)

[5.4 Experimental results 29](#_Toc375409813)

[References 30](#_Toc375409814)

#### Table of Figures

[Figure1.1. Matrix B corresponding to a sparse linear system Bx = 0 (mod 2) obtained by the Multi-Polynomial Quadratic Sieve (MPQS) method for a 30-decimal integer. The 179 X 210 matrix B has 1916 nonzero elements. 5](#_Toc291976584)

[Figure 3.1: Enlarging performance gap between GPUs and CPUs. 19](#_Toc291976585)

[Figure 3.2: CPUs and GPUs have fundamentally different design philosophies. 19](#_Toc291976586)

[Figure 3.3: Comparison of number of cores available on CPU and GPU. 20](#_Toc291976587)

# 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.

# Chapter 1 Introduction

## 1.1 Introduction

Some integer factorization algorithms require several nonzero vectors x ∈ GF(2)n such that **BX = 0**, where B is a given m x n matrix over the field GF(2)**,** usually very sparse and with m < n. These include the (obsolete) continued fraction method, quadratic sieve (QS), arid number field sieve. For example, when factoring an integer M, the QS method finds congruence’s

(mod M)(1 ≤ j ≤ n)

(1.1)

Here the *pi*are primes (or -1) and the *bij*are exponents, mostly zero. QS then tries to find *S C*{ 1, 2, - - ,n } such that both sides of

(mod M)

(1.2)

are perfect squares. The left product is automatically a square, but the right product is a square only if all exponents are even, i.e., if ∏ j∈S  bij ≡ 0 (mod 2) for 1 ≤ i ≤ m. This is equivalent to Bx≡ 0 (mod 2), where B = (bij)**,**x= **(**xj**),** and where ***x****j*= 1 if j **∈** *S* and ***xi*** = 0 if jS.

The Block Lanczos achieves this objective by decomposing GF(2)ninto several subspaces of dimension almost N which are pairwise orthogonal with respect to the symmetric n x nmatrix A = BT B**.** The algorithm takes about n/(N - 0.76) iterations. Each iteration applies the matrices Band BT to an n x **N**matrix and does a few supplementary operations (i.e., inner products of two n x **N**matrices, multiplication of an nx **N**matrix by an **N** x **N**matrix, multiplication and inversion of **N** x **N**matrices).

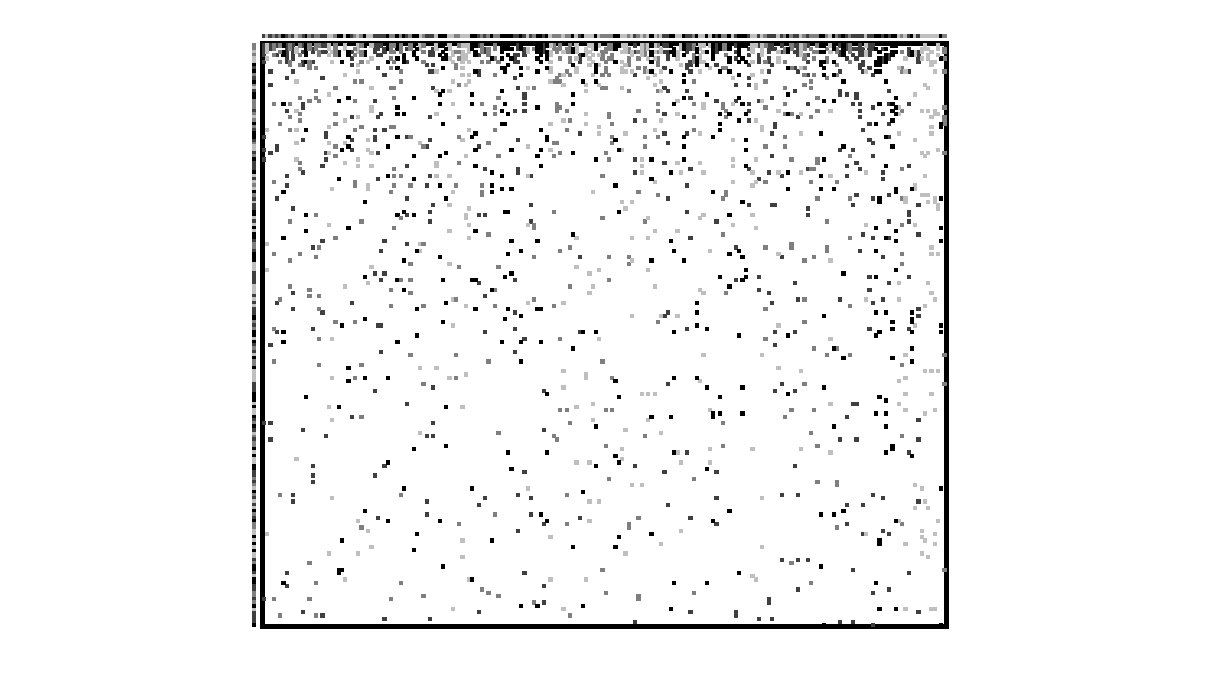
The matrix that arises in the sieving stage of factoring has a specific structure. It is extremely sparse, with around than 60-80 nonzero entries per row. It is relatively dense in one part (columns corresponding to small prime) and very sparse in another part (columns corresponding to large primes). The Gaussian elimination algorithm can be used efficiently for such matrices of order up to 104. The main drawback of the method is, as the algorithm proceeds, the matrix becomes more and more dense due to fill-in. A second problem is its cubic runtime. The algorithm is not a good choice for matrices of order over 107.

Certain other methods have been suggested to take advantage of sparsity. They are:

1. Structured Gaussian elimination

2. Lanczos iterations

3. Wiedemann iteration



#### Figure1.1. Matrix B corresponding to a sparse linear system Bx = 0 (mod 2) obtained by the Multi-Polynomial Quadratic Sieve (MPQS) method for a 30-decimal integer. The 179 X 210 matrix B has 1916 nonzero elements.

After extensive use of these in large integer factoring exercises, the accepted methodology is to use a combination of the principles of structured Gaussian elimination and Lanczos or Wiedemann iterations. 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, We can use Lanczos or Wiedemann iterations efficiently on a smaller matrix.

Theoretically, both Lanczos and Wiedemann algorithms have the same worstcase quadratic time estimate. Their runtimes are expected to be not much bigger than n2 for matrices with total number of entries of order n(i.e, for small d). But in practice, Wiedemann is likely to be slower than Lanczos by a factor of 3/2, and also requires additional storage. Wiedemann is quite a bit more complicated to program, requires randomization procedures and deals with non-square and highly singular cases. Taking all these factors into consideration, we chose the Lanczos algorithm as the best method for finding the required linear dependencies.

We have carried out an efficient implementation of Block Lanczos algorithm exploiting

the nature of GF(2)n vectors and sparsity of the sieve matrix B. We discuss our implementation and performance results in the subsequent chapters.

## 1.2 Notations

Throughout this paper, A denotes a symmetric n x n matrix over a field K. Two vectors v, w ∈ Kn are said to be A-orthogonal if vTAw = 0. If V and W are subspaces (or subsets) of Kn, then we define the block operations

V + W = { v + w : v ∈V and w ∈ W} ,

VTW = {vTw: v ∈ V and w ∈ W},

AV = {Av: v ∈V }.

Two subspaces V and W of Kn are said to be A-orthogonal if vTAw = 0 for all v ∈ V and w ∈ W; this is equivalent to VTAW = (0).

If V is an n1 x n2 matrix, then (V) denotes the subspace of generated by the column vectors of V.

If W is a subspace of Kn , then O(W) represents a vector in W or a matrix with column vectors in W. It satisfies O(V) + O(W) = O(V + W). If M is a matrix of suitable size, then we can replace O(W)M by O(W), but cannot similarly simplify MO(W).

We denote the number of bits per computer word by N.

The k x k identity matrix is denoted by Ik.

# Chapter 2 Lanczos

In this chapter we discuss the Lanczos method for solving linear equations over GF (2) . But the Lanczos algorithm fails in GF(2) due to self orthogonality property of binary vectors. To eliminate this problem, we work in an extension field GF(2) in place of GF(2).This modification is called Block Lanczos. We give the complete description of the Block Lanczos algorithm.

## 2.1 The Lanczos algorithm

Suppose A is a symmetric positive definite n × n matrix over a field K. If b ∈ Kn , we need to find a nonzero x such that Ax = b. The Lanczos iterations are as follows,

*w0 = b*,

(i > 0) where *cij =* (2.1)

Notice that the generated vectors *wi* are A-orthogonal, i.e. = 0, i ≠j

One can verify this by induction on max(i, j). For max(i, j) = 0, the statement is trivial. When max (i, j) = 1, then we have

*w0* = (*Aw0 − c1,0w0*)*Tw0*

= *A2 w0* – () *w0*

= 0 (2.2)

Suppose the statement holds for max (i, j) = y−1. Now we have to show the statement is true for max (i, j) = y also. Without loss of generality, let us suppose i = y and j < i. Then from the equation (2.1) we have

*wj* =(*wi-1*−)*Twj*

= *A*2*wj* −*wj*

= *A2wj − ci−1,jwj* (using induction)

= 0 (2.3)

If i > n, then the vectors w0,w1,w2, . . . ,wn are linearly dependent. Suppose we have ∑*ajwj* = 0 where aj ≠0 , then

(∑*ajwj* = 0 (premultiply by)

*aiwi*= 0 (ai ≠0)

By positive definiteness, wi = 0. So the Lanczos iterations will terminate at some point. Let m denote the first value of i such that wi = 0.

Define

(2.4)

Now we show that x is indeed a solution. From (2.1) we have

*w0 = b*

*w1 = Aw0 − c10w0*

*w2 = Aw1 − c21w1 − c20w0*

...

*wm−1 = Awm−2 − cm−1,m−2wm−2 − . . .cm−1,1w1 − cm−1,0w0*

*wm = Awm−1 − cm,m−1wm−1 − . . .cm,1w1 − cm,0w0*.

From the fact wm = 0, b (= w0) can be written as a linear combination of wi and so can the vectors Awi . So the vector space spanned by Awi and b is contained in the vector space spanned by wi. The vectors wi generate the Krylov sequence and thus we have

span(*Aw0,Aw1, . . .Awm−1, b*) *C* span(*w0,w1, . . . ,wm−2,wm−1*).

From (2.4)

Ax − b ∈ span(*Aw0,Aw1, . . . ,Awm−1, b*)

This implies

Ax − b ∈ span(*w0,w1, . . . ,wm−1*)

(2.5)

By construction, we have

*A = A*

*=Awj* ( from (2.2))

= (0 ≤ j ≤ m − 1)

(2.6)

From (2.5) and (2.6), we have (Ax − b)*T* (Ax − b) = 0, and thus we get

Ax − b = 0

and hence the claim x is a solution. In formula (2.1) , it appears to require addition of suitable multiples of earlier wj when computing wi. However, by A-orthogonality of wi, the coefficients cij become zero when j < i − 2, since

*A2wi−1 = (Awj)TAwi−1*

*=(wj+1 +)T Awi−1*

= *A wi−1* + *A wi−1*

= 0 (j < i − 2).

(2.7)

Hence (2.1) simplifies to a two term addition,

*wi = Awi−1 − ci,i−1wi−1 − ci,i−2wi−2* (i ≥ 2)

(2.8)

Suppose wi−1,wi−2 are known in advance. In each iteration when finding wi, one needs to compute

and

This can be done by three inner products if are known. Another inner product is required for computing while updating the partial sum of x. If the matrix A has an average of d entries per row or column, the cost per iteration is O(dn) to multiply by A and O(n2) for other vector operations. Hence the total time is O(dn2) + O(n2).

Storage requirement is quite small: storing a few temporary binary vectors, and the

0−1 matrix A.

## 2.2 Failure of Lanczos in GF (2)

The above method cannot be directly used for the vectors over GF (2), because of the fact that a large number of 0−1 vectors wi on an average will be self-orthogonal with respect to the matrix A. We discuss below how this obstacle arises and how it is taken care of.

The vectors arising in satisfy

(0 ≤ i ≤ m − 1),

= 0 (j ≠i)

*AWC W* where *W* = <*w0,w1, . . . ,wm−1*>

(2.9)

These 3 conditions ensure finding a solution vector x of the equation Ax = b for b ∈ W. Over GF (2), about half of all vectors are A−orthogonal to themselves. This means we can find many vectors wi ≠0 having property = 0. The self A-orthogonal vectors will present the following problem.

Consider the set of vectors w0,w1, . . . ,wj , If j > n they are linearly dependent.

(ak ≠0)

Premultiplying by , we have

This implies

*ai* = 0

From this we conclude wi = 0. If wi is A−orthogonal to itself, the algorithm will fail to terminate.

## 2.3 Block Lanczos

To eliminate this problem, we use a set of vectors (representing subspaces) instead of a single vector. Now each subspace is represented by a matrix. The matrix-vector products are replaced by matrix-matrix products in GF (2n). One can apply the matrix A to N (typically 32 or 64) different vectors in GF (2n) at once using bitwise operators. This modification is called Block Lanczos.

Let A be a symmetric n × n matrix over the field GF (2). The Block Lanczos algorithm produces a sequence of subspaces of GF (2n) which are pairwise A−orthogonal. The properties of vectors wi in equation (2.9) ensure the finding of a solution vector. These properties are now generalized to a A-orthogonal subspaces Wi to ensure a solution in the modified sequence of iterations.

The condition in (2.9) is replaced by a requirement that no nonzero vector in Wi be A−orthogonal to all of Wi. The subspace W satisfying this property is said to be A− invertible. It will have a basis W of column vectors such that WTAW 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 ∈ Kn can be uniqucly written as v + w where w ∈ W and W Av = (0). Indeed, if the columns of W are a basis for W, then

w = W (WTAW)-l WTAu.

The generalization to subspaces is

Wi is A-invertible ,

(i ≠j) ,

(2.10)

*AW C W*, where *W=W0+W1+...+Wm-1*.

Assume (2.10). Given b ∈ W, we can construct an x ∈ W such that AX = b. Let x = ∑wj , where wj ∈Wj is chosen so that Awj - b is orthogonal to all of Wj. If the columns of Wj form a basis for Wj, then

(2.11)

Fix N > 0. At step i, we will have an n x N matrix V, which is A-orthogonal to all earlier Wj. The initial V0 is arbitrary. We select Wi using as many columns of V, as we can, subject to the requirement that W; be A-invertible. More precisely, we try to replace the Lanczos iterations by

*Wi=ViSi*,

(i ≥ 0) (2.12)

wi = <Wi> .

Stop iterating if say for i = m.

Here Si is an N x Ni projection matrix chosen so that is invertible while making Ni ≤N as large as possible. The matrix Si should be zero except for exactly one 1 per column and at most one 1 per row. These ensure that and that is a submatrix of IN reflecting the vectors selected from Vi. For example, if N = 3, then

T

selects the second and third columns of Vi for inclusion in Wi.

Formula (2.12) for Vi+l tries to generalize while ensuring *Wj AVi+l*= {0} for j ≤ i if the earlier Wj exhibit the desired A-orthogonality. We use

(2.13)

The terms Vi - WiCi+l, i in (9) select any columns of Vi not used in Wi; those columns are known to be A-orthogonal to W0 through Wi-1, and the choice of Ci+l,i adjusts them so they are A -orthogonal to W, as well. Without the V, term, rank(Vi+l) would be bounded by rank(*AWi*) ≤ rank(Vi), and would soon drop to zero.

## 2.4 Simplifying Block Lanczos

We can optimize the computations of *Vi+1*. Consider the product in the coefficient term *Ci+1,j*.

If j < i, then

= ()()

= (*AWj*)TA

= (*Vj+1 − Vj* + a linear combination of *Wk*)T*A*

= *A* −*A*

= *A* (2.14)

If Sj+1 = IN(so that Wj+1 = vj+1) and if j < i−1, then the above term will become zero. So we have coefficients Ci+1,j = 0 for j ≤(i − 3). This means,Vj+1 should be A−orthogonal to Wj+3 through Wm. For this we require all vectors in Vj+1 be used either in Wj+1 or Wj+2.

From this, we have the equation computing Vj+1 in (14) rewritten as

*Vj+1*= *AWi* + *Vi*−*Wi*C*i+1,i* −*Wi−1 Ci+1,i−1* −*Wi−2 Ci+1,i−2* (i ≥ 2)

(2.15)

The above equation remains valid for i = 0 and i = 1 if we define Vj = 0 and Wj = 0

for j < 0.

In order to have a nice framework for computing the terms Vi+1, let us substitute

the values of coefficients in (2.15)

= + − C*i+1,i* − *Vi−1Si−1Ci+1,i−1* − *Vi−2Si−2Ci+1,i−2*

+ − *A*(*A*+)−−*A*2

(2.16)

where

= ()−1= ( )−1 (2.17)

By expressing the inner products and in terms of and , the equation further simplifies to

(2.18)

for i ≥ 0, where

)( (2.19)

We take Wjinvand Vj to be 0, Sj to be IN for j < 0. The pseudo-code for the algorithm is given as Algorithm 1.

Algorithm 1: Block Lanczos

Input:Matrices B of size n1 × n2 and Y of size n2 × N

Output:The matrices X and Vm

Cmt: The algorithm BT (BX) = V0 = BT

1: Initialization: X = 0

2: V0 = AY = BT *\** (BY )

3: = = (BTB)V0 = (BV0)T \*BV0

4: Compute AV0 = BT \*(BV0)

5: i = 0

6: while ≠0 do

7: compute = ()*T* *\** ()

8: [ , *S* ] = FindInverse (, *S*,N)

9: X = X + Vi *\** ( *\** ( *\** ))

10: = () *\** (*S*) +

11: ()

12: ( *\** *S*)

13: (*IN* − +)() *S*

14:

15: compute B and *A* = BT *\** B

16: = = (B)T *\** (B)

17: i = i + 1

18: end while

19: Return X and Vm

## 2.5 Finding the Null Space of the Sieve Matrix

The sieving process of the NFS algorithm produces an n1 × n2 (n1 < n2) matrix B called the sieve matrix. Each row represents a prime from the factor base and columns correspond to relations. We need to find dependencies among the relations. The Lanczos algorithm requires the matrix to be symmetric. So we define a new symmetric matrix A = BTB. Any solution of BX = 0 will satisfy AX = 0. So the nullspace of B is contained in the null space of A. We run the algorithm on A as follows.

Select an n2 ×N random matrix Y over GF (2), compute V0 = AY. If V0 = 0, then Y is the solution. Otherwise we continue the algorithm with V0 and try to find a matrix X that satisfies AX = V0 = AY. The main idea is that if we succeed in finding such an X, then, since A(X−Y) = 0, the column vectors of X−Y will be random vectors in the nullspace of A.

We perform the Lanczos iterations until we get the matrix Vi such that . The algorithm ends in the following two cases.

(Suppose at step i = m, the algorithm terminates)

*  and = 0
*  and ≠0

Let V0, V1, . . . Vm−1 be the generated matrices during the Lanczos iterations and

let Wi be the orthogonal subspaces generated by column vectors of corresponding Vi

and span(Vm) =Wm.

Denote

*W = W1 +W2 + . . . +Wm−1*

*Wm = < Vm >* (2.20)

Then Wm is orthogonal to itself and to *W* . Compute

(2.21)

By construction, we have Ax − V0 W +Wm.

case 1:

This case is trivial one. If Vm = 0, then AX= V0 = AY. So X − Y is the solution. The probability of this case is low. Often the algorithm terminates in case 2.

case 2:

In this case, the nonzero matrix Vm is A−orthogonal not only to itself but to Wj for j < m. In practice, this Vm has small rank (perhaps two). The generated matrices Vj and Wj are contained in the Krylov sequence

V =< V0 > + < AV0 > + < A2V0 > +. . . (2.22)

Then V is the direct sum of W and of < Vm >= Wm and we have *AWm*  *C* *AV C V = W + Wm*. Suppose w0 + w1 + . . . + wm ∈ AWm, where each wj ∈ Wj . Then wesee that

*wjT AWj = (w0 + w1 + . . . + wm)T AWj C( AWm)T AWj*

*= A(AWj) C A (W +Wm)*

= 0 (2.23)

Since wj ∈ Wj and Wj is A−invertible for j < m, we have wj = 0.

So the column vectors of A(X−Y) and Vm are in *Wm*. The total rank of X−Yand Vm is at most 2N. Using Gaussian elimination, one can take linear combinations of X−Y and Vm to get the nullspace of A. The same construction can be used to get the nullspace of B. Define a augmented matrix Z = [X − Y : Vm]. Compute BZ and take the column operations on this matrix to get zero columns. Apply the same column operations on Z to get the null space of B. The pseudo-code for the main algorithm is given as Algorithm 2, below.

Algorithm 2 Finding NullSpace of B

Input: The matrix B

Output: Nullspace X

Cmt: The algorithm finds a part of nullspace of B

1: Choose a Y n2×N matrix

2: Get [X, Vm] = BlockLanczos(B, Y,N, n1, n2)

3: Z = [X − Y, Vm]

4: BZ = [B \* (X − Y ),B\* Vm]

5: U = Nullspace(BZ)

6: ZU = Z \* U

7: Return a basis matrix of ZU.

## 2.6 Basic operations with bitwise operators

In this subsection, we will show how the running time of the algorithm can be decreased by using the fact that every computational operation of the whole process is over the field GF(2). Nowadays, computer operations are executed in the binary number system. Therefore, the basic unit of information, called a bit, can take one of two possible values, 0 and 1.

This leads us to the idea to represent each vector component in the Block Lanczos algorithm (which works over GF(2)) by a bit.

Let N denote the word size of our computer, namely the number of bits needed to represent an unsigned integer. Each unsigned integer has a unique combination of N bits.

Applying the idea to store the elements as bits converts vectors to numbers and certain matrices to vectors, as follows: if we have an N-vector with components from GF(2), this can be stored in one integer, where every bit of this integer corresponds to a component of the vector.

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | & | ^ |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Table 1: Bitwise operators acting on 1-bit variables.

This method can be extended to store matrices of size n X N or N X n as an n-vector. For example, let C be a matrix of size n X N over the field GF(2). This matrix has in every row N elements so that each row can be stored in one integer. Because C has n rows, the total number of integers is equal to n and therefore we obtain an n-vector from a matrix. Hence, the bit decomposition of the ith component in this vector corresponds with the ith row of the matrix.

A given matrix D of size N X n can be stored in the same way as the matrix C with the only difference that here every integer of the vector contains one column of D.

This way of storing data has two advantages. The algorithm needs less memory storage, because every element of the matrices is a bit. Moreover, we can use bitwise operators over integers, which decreases the running time tremendously. The basic bitwise operations we will need are & (AND) and ^ (XOR), which are defined in Table 1. The appealing property of these operators is that they can be used bitwise over integers. We would like to emphasize that this works faster than performing an N-loop and calculating each bit operation separately (even though the result is the same).

# Chapter 3 GPUs

A graphics processing unit or GPU (also occasionally called visual processing unit or VPU) is a specialized circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.

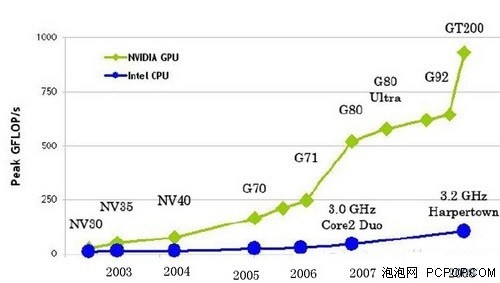
## 3.1 History

Most software developers have relied on the advances in hardware to increase the speed of their applications under the hood; the same software simply runs faster as each new generation of processors is introduced. This drive, however, has slowed since 2003 due to energy consumption

and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Virtually all microprocessor vendors have switched to models where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power.

Many-core processors, especially the GPUs, have led the race of floating-point performance

since 2003. This phenomenon is illustrated in Figure 1.1. While the performance improvement of general-purpose microprocessors has slowed significantly, the GPUs have continued to improve relentlessly. As of 2009, the ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 10 to 1. These are not necessarily achievable application speeds but are merely the raw speed that the execution resources can potentially support in these chips: 1 teraflops (1000 gigaflops) versus 100 gigaflops in 2009.



#### Figure 3.1: Enlarging performance gap between GPUs and CPUs.

## 3.2 Design

GPUs have more space provided for Arithmetic units than for data cache and control. The differences is in the fundamental design philosophies between the two types of processors , CPU and GPU, as illustrated in figure 2. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed.

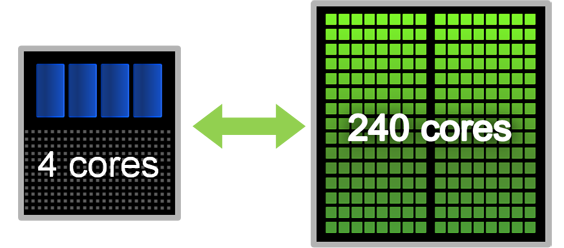


#### Figure 3.2: CPUs and GPUs have fundamentally different design philosophies.

Because of frame buffer requirements and the relaxed memory model—the way various system software, applications, and input/output (I/O) devices expect their memory accesses to work—general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. In contrast, with simpler memory models and fewer legacy constraints, the GPU designers

can more easily achieve higher memory bandwidth. The more recent NVIDIA GT200 chip supports about 150 GB/s.

The design philosophy of the GPUs is shaped by the fast growing video game industry, which exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced games. This demand motivates the GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The prevailing solution to date is to optimize for the execution throughput of massive numbers of threads. The hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, thus minimizing the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications so multiple threads that access the same memory data do not need to all go to the DRAM. As a result, much more chip area is dedicated to the floating-point calculations.



#### Figure 3.3: Comparison of number of cores available on CPU and GPU.

GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well; therefore, one should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs. In comparison to the central processor’s traditional data processing pipeline, performing general-purpose computations on a graphics processing unit (GPU) is a new concept.

Thus, the level of parallelism supported by GPU hardware is very high and still increasing quickly.

# Chapter 4 CUDA

We are using CUDA C programming language for the implementation of Block Lanczos on GPU. CUDA (Compute Unified Device Architecture) programming model, introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.

CUDA has a Heterogeneous programming model. The CPU and GPU are separate devices with separate memory spaces. The CPU is called as Host and GPU is called Device. CPU code is standard C/C++ while GPU code is Subset of C with extensions of CUDA libraries. CUDA helps us to scale GPU code to 100s of cores, 1000s of parallel threads and facilitate heterogeneous computing.

## 4.1 Threads and Kernels

Parallel portions of an application are executed on the device as kernels with one kernel executing at a time and many threads executing each kernel. Kernel is launched as

Kernel<<<blocks,threads>>>( )

Where,

Blocks =number of blocks in a grid that we intend to use,

Threads =number of threads in each block.

Differences between CUDA and CPU threads:

* CUDA threads are extremely lightweight
  + Very little creation overhead
  + Fast switching
* CUDA uses 1000s of threads to achieve efficiency
  + Multi-core CPUs can use only a few

A CUDA kernel is executed by an array of threads. All threads run the same code. Each thread has an ID that it uses to compute memory addresses and make control decisions.

### 4.1.1 Thread Cooperation

Thread cooperation is a powerful feature of CUDA. Thread cooperation is valuable as it can be used to share results to avoid redundant computation. Also share memory accesses results in Bandwidth reduction.

Threads can be used to perform same computation on different data by passing chunks of input data to each thread. In this the same copy of kernel is executed by different threads in different blocks. But there may arise a situation in some applications where threads may need to cooperate between them while do computations. This can be achieved by using shared memory on GPU.

A shared memory is available on each block and all threads in a block can use it. The CUDA C compiler treats variables in shared memory differently than typical variables. It creates a copy of the variable for each block that you launch on the GPU. Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks. This provides an excellent means by which threads within a block can communicate and collaborate on computations. Furthermore, shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM. Because of this, the latency to access shared memory tends to be far lower than typical buffers, making shared memory effective as a per-block, software managed cache or scratch-pad. The threads can be synchronized to avoid race condition. This allows programs to transparently scale to different GPUs.

## 4.2 Memory Types

The different memory types available on GPUs are as follows:-

* *Registers* :- They are on chip with read/write access and each thread has its own registers. Thus their lifetime is lifetime of the thread in which they reside.
* *Local* :- This is off-chip memory with read/write access for each thread.
* *Shared* :- This is on-chip memory with read/write access for each block. All threads in a block can access and use this memory.
* *Global* :- This is off-chip memory with read/write access for all threads and also the host. The lifetime of global memory is the lifetime of an application.
* *Constant* :- This is off-chip cached memory with only read access for all threads and host and has a lifetime of an application.
* *Texture* :- This is off-chip cached memory with only read access for all threads and host and has a lifetime of an application.

The constant memory and texture memory are regions in memory used for special purposes. We use constant memory for data that will not change over the course of a kernel execution. The constant memory is used if a certain number of threads access the same memory location. There are two reasons why reading from the 64KB of constant memory can save bandwidth over standard reads of global memory:

* A single read from constant memory can be broadcast to other “nearby” threads, effectively saving up to 15 reads.
* Constant memory is cached, so consecutive reads of the same address will not incur any additional memory traffic.

In the CUDA Architecture, a warp refers to a collection of 32 threads that are “woven together” and get executed in lockstep. At every line in your program, each thread in a warp executes the same instruction on different data. When it comes to handling constant memory, NVIDIA hardware can broadcast a single memory read to each half-warp i.e. 16 threads. If every thread in a half-warp requests data from the same address in constant memory, the GPU will generate only a single read request and subsequently broadcast the data to every thread. If you are reading a lot of data from constant memory, you will generate only 1/16 (roughly 6 percent) of the memory traffic as you would when using global memory.

Like constant memory, texture memory is another variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. In a computing application, this roughly implies that a thread is likely to read from an address “near” the address that nearby threads read.

## 4.3 Atomics

The execution of some operations cannot be broken into smaller parts by other threads. The operations that satisfy this constraint are known as atomic operations.

Trying to cope with potentially tens of thousands of threads simultaneously modifying the same memory addresses is a common situation where a massively parallel machine can seem burdensome. Fortunately, we have hardware-supported atomic operations available to help ease this pain. CUDA C supports several atomic operations that allow user to operate safely on memory, even when thousands of threads are potentially competing for access. Some of them are atomicAdd( ), atomicMin( ) and bitwise operations like atomicXor( ).

## 4.4 Page-Locked Host Memory

Memory can be allocated on host by using cudaHostAlloc( ) call instead of C malloc( ) call. This allocates a buffer of page-locked (pinned) host memory, i.e., the operating system guarantees us that it will never page this memory out to disk, which ensures its residency in physical memory. It becomes safe for the OS to allow an application access to the physical address of the memory, since the buffer will not be evicted or relocated.

Knowing the physical address of a buffer, the GPU can then use direct memory access (DMA) to copy data to or from the host. Since DMA copies proceed without intervention from the CPU, it also means that the CPU could be simultaneously paging these buffers out to disk or relocating their physical address by updating the operating system’s page-tables.

Page-locked host memory enjoys roughly a twofold performance advantage over standard pageable memory when used for copying data between the GPU and the host.

The disadvantage of using page-locked host memory is that the computer running the application needs to have available physical memory for every page-locked buffer, since these buffers can never be swapped out to disk. This means that the system will run out of memory much faster than it would if you stuck to standard malloc( ) calls.

### 4.4.1 Zero Copy Host Memory

Memory can also be allocated on host from device using cudaHostAlloc( ) and passing flag cudaHostAllocMapped. This is called Zero-Copy host memory as it does not require copy to and from GPU. Since the task of copying is not required anymore the application runs faster and enhances performance. It has the same disadvantage as page-locked host memory in that the computer running the application needs to have available physical memory for every page-locked buffer, since these buffers can never be swapped out to disk.

CUDA helps to achieve great performance on GPU hardware by data-parallel computations by following a few simple guidelines:

* Using parallelism efficiently
* Coalescing memory accesses if possible
* Taking advantage of shared memory
* Exploring other memory spaces
  + Texture
  + Constant

# Chapter 5 Implementation of Block Lanczos on GPU

The Block Lanczos algorithm is an optimal candidate for parallelization. Since the size of the matrix A is so large, the main bottleneck in the algorithm is the calculation of AX.. However, this is easily parallelized by assigning a different portion of the matrix to each available device, in such a way that the whole matrix is accounted for.

## 5.1 Data Distribution

The typical matrices which we wish to apply to this algorithm are very large and mostly sparse. We can take advantage of the latter and store the matrix 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 we 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 our typical N may be two to twenty times larger than this, increasing the RAM requirement substantially.

First of all, the matrix corresponding to the system of equations that we get 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 columns are called dense since they have relatively many nonzero entries. Once sparsity increases, it will be more worthwhile to store the locations of these entries rather than storing all the particular entries.

The sieving matrix that generated in the NFS is a huge sparse matrix over the field GF(2). For example, in the RSA-512 factorization, the matrix has 131 million rows and 79 million columns. Typically the number of nonzero entries per column is 40 to 60. We assume that the matrix is stored in column major order form. So the non-zero row indices will be stored. As each entry occupies 4 bytes, the total size of the matrix is 79x106x40x4 which is around 12GB. In the factorization of RSA-640, the output matrix had 166x107 relations. So the size of the sieve matrix is 107x107x40x4 bytes, which is around 171GB.

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 we may need to keep the matrix on several devices. 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 computations uniformly.

We have stored the matrix in column major order form with each column giving the indices of rows with non-zero elements. The matrix is being stored in global memory on GPU. The matrix has then divided into strips of sizes which depend on shared memory restrictions of GPU. The offsets of these strips have also been stored in global memory on GPU. These offsets are used for calculating transpose of matrix B.

## 5.2 Operations

We now discuss the fine points of our implementation. Recall the equation of finding the next V,

The matrices D,E and F involve the computation of the matrices *VTA2V = (AV)TAV* and *VTAV = (BV)TBV* and a few multiplication operations on N×N matrices. The computation of involve 4 matrix-data products. All these computations are classified as the following three kinds of outer-products.

* The vector u of order n×N and the vector v of order N ×N. The outer-product uv. It is denoted by op1.
* The vector u of order n×N and the vector v of size n×N . The outer-product uT v is N × N. It is denoted by op2.
* The vector u of order N×N and the vector v of order N×N. The outer-product is uv. It is denoted by op3.

All matrix-data products in the above equation are op1 products. The matrices *VTA2V* and *VTAV* are op2 products.

### 5.2.1 Computation of Outer Product(op1)

If the product uv is computed in a straight forward manner, then for every row of u we requires N2 AND operations, N(N − 1) XOR operations. The total cost of the algorithm will be O(nN(2N − 1)). We can perform the computation efficiently in a different way taking advantage of packed representation. We describe the method below. In the following sections, the term word refers to an integer or a long integer depending on the block size N =32 or 64 respectively.

In practice, the binary matrices u and v are stored in a packed form. The matrix u is stored as an array of n words. The matrix v is stored as an array of N words. Using this packed representation, the operations on N bits can be done at the same time. For the ith row of the matrix u, we take the word u[i] and extract the N bits out of it and take XOR operations on entries of the matrix v. If the kth (0 ≤ k ≤ N −1) bit is equal to one, then perform the operation uv[i] = uv[i] ^ v[k]. The pseudo-code is given as algorithm 3.

Algorithm 3 Outer-Product (op 1)

Input : Matrix u of size n × N and v of size N × N

Output : The matrix uv of size n × N

Comment : u,v and uv are stored as arrays

1: uv[i] = 0 for 1≤ i ≤ n − 1

2: for j = 0 to n − 1 do

3: data = u[i].

4: for k = 0 to N − 1 do

5: b = ExtractBit(data,k) (kth bit is extracted from data)

6: if b = 1 then

7: uv[j] = uv[j] ^ v[k]

8: end if

9: end for

10: end for

For every word of the matrix u, the function ExtractBit requires N AND operations. If all bits of u[i] are equal to one , then N XOR operations are required to compute one word of the matrix uv. Suppose the cost of a single XOR or AND operation is t cycles. The maximum total cost for computing op1 is 2n1Nt cycles. The computations of outer-product op3 are done in the same way. The maximum cost for computing the product op3 is 2N2t cycles.

### 5.2.2 Computation of Outer Product(op2)

Let the matrices u and v be of size n1 ×N. With ordinary multiplication, the computation uT v requires N2n2 AND operations, (n2−1)N2 XOR operations and few extra memory accesses for swapping operations to get transpose of u. Exploiting a packed representation and structure of uT ,we can perform the computations in a much better way.

The size of uT is N × n1.We study the multiplication of every row of uT with col 0 of v. The element uT [0][0] is the MSB of u[0]. While taking the InnerProduct( row 0,col 0), this bit is AND-ed with MSB of v[0].Likewise, in InnerProduct(row i,col 0) the element uT [i][0] ((N − i)th bit of u[i]) is AND-ed with the MSB of v[0].In general, the InnerProduct(row i,col j) requires the element uT [i][0] to be AND-ed with jth bit of v[0].

The bit uT [i][0](0 ≤ i ≤ n1−1) is operated on all bits of integer v[0]. It is clear that every bit of u[0] is operated on integer v[0]. So the bitwise operations on individual bits can be replaced by XOR operations over integers. The rows of outer product are computed as follows.

We take u[i](0 ≤ i ≤ n1 − 1). If the kth bit of u[i] is one, then we manipulate the entry uT v[k] = uT v[k] ^ v[i] (0 ≤ k ≤ N −1). The entry uT v[k] represents the kth row of outerproduct uT v.

The Pseudocode for computing the outer product op2 is given as Algorithm 4.

Algorithm 4 Outer Product uT v

Input : Matrix u of size n1 × N and v of size N × N

Output : The matrix uTv of size n1 × N

Cmt : u,v and uTv are stored as arrays

1: uT v[i] = 0 for 0 ≤ i ≤ N

2: for j = 1 to n1 do

3: bdata = u[j].

4: for k = 1 to N do

5: b = ExtractBit(data,k) (kth bit is extracted from data)

6: if b = 1 then

7: uT v[j] = uT v[j] ^ v[k]

8: end if

9: end for

10: end for

Extracting the bits out of one u[i] requires N AND operations. If all bits of u[i] are equal to 1, then manipulating the entries of uT v needs N XOR operations. The total cost of computations is 2n1Nt cycles.

## 5.3 Implementation

### 5.3.1 Random Vector Y

The initial vector Y of size n2xN is randomly generated on host and stored in page-locked memory on host. Each row of Y is of N bits so each row can represented by a single word. Thus Y contains n2 rows of words. So Y is stored as vector of n2 words where words represent the word-size N of the system. We are using page-locked memory as the transfers from page-locked memory to device is roughly two times faster than normal copy from host to device.

### 5.3.2 Computation of *V0*

The computation of *V0*= *BT \*(BY)* is carried as follows. We allocate zero-copy memory for *V0* as it used in each iteration. We calculate *BY* first by copying B and Y in global memory on device and starting the kernel function for multiplication with 16 blocks with 512 threads each. The computation of BY is a multiplication of a n1xn2 matrix and a vector of n2 words. We allocate shared memory in each block to store result of computation of each block. These results are copied back to host zero-copy memory where they are XOR-ed to get final result. We similarly multiply BT with the result to get final *V0*.

### 5.3.3 Computation of *Ci*

The computation of = = (B)T *\** (B) is same as computation of outer product op2 as discussed above.

### 5.3.4 Computation of *ViTA2Vi*

The computation of = ()*T* *\** () is same as outer product op2. The value of is computed as  *BT \*(B).* This is computed in same way as *V0* just replacing Y with .

### 5.3.5 Computation of V*i+1*

The computation of consists of 4 multiplications of type op1.

## 5.4 Experimental results

# References

1. A Block Lanczos Algorithm for Finding Dependencies over GF (2) by *Peter L. Montgomery.*
2. Parallel Computations for the Matrix stage of Integer factorization by *M. Ramanjulu.*
3. A New Parallel Approach to the Block Lanczos Algorithm for Finding Nullspaces over GF(2) by *Ildik’o Flesch*
4. Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm —a case study by *Rob H. Bisseling, Ildik´o Flesch*
5. Parallel Block Lanczos for Solving Large Binary Systems by *Michael Peterson*
6. Computational Aspects of Discrete Logarithms by *Rob Lambert*
7. CUDA by Example-An Introduction to General Purpose GPU programming by *Jason Sanders* and *Edward Kandrot*.
8. NVidia CUDA C Programming Guide.
9. CUDA-GDB The NVidia CUDA Debugger – User Manual.
10. Programming Massively Parallel Processors- A Hands on Approach by *David Kirk* and *Wen-mei W. Hwu.*
11. Efficient Sparse Matrix-Vector Multiplication on CUDA by *Nathan Bell* and *Michael Garland.*
12. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors by *Nathan Bell* and *Michael Garland* (NVidia Research).