Information Sciences Letters

Volume 11
Issue 3 *May 2022*

Article 7

2022

Generating a Shortest B-Chain using Multi-GPUs

Hatem M. Bahig
Department of Mathematics, Faculty of Science, Ain Shams University, Cairo 11566, Egypt, h.m.bahig@gmail.com

Dieaa I. Nassr
Department of Mathematics, Faculty of Science, Ain Shams University, Cairo 11566, Egypt, h.m.bahig@gmail.com

Information Sciences Letters
An International Journal

http://dx.doi.org/10.18576/isl/110307

Generating a Shortest B-Chain using Multi-GPUs

Hatem M. Bahig and Dieaa I. Nassr*

Department of Mathematics, Faculty of Science, Ain Shams University, Cairo 11566, Egypt

Received: 2 Jan. 2022, Revised: 23 Feb. 2022, Accepted: 2 Mar. 2022
Published online: 1 May 2022

Abstract: Let B be a finite set of binary operations over the set of natural numbers N. A B-chain for a natural number n, denoted by $BC(n)$, is a sequence of numbers $1 = c_0, c_1, \ldots, c_l = n$ such that for each $i > 0, c_i = c_j \circ c_k$, where $0 \leq j,k \leq i-1$ and $\circ$ is an operation of B. Generating a shortest B-chain for n plays an important role in increasing the performance of some cryptosystems and protocols. This paper has two purposes. The first is to propose a generic algorithm to generate a shortest B-chain using a single CPU and a single GPU for any B. The second is to propose two strategies to improve the generation of a shortest B-chain using two (or more) GPUs. Using two GPUs, the experimental study shows that the first strategy improves the performance by about 20%, while the second strategy improves the performance by about $30 \sim 35\%$ in case of $B = \{+\}$. It is also possible to combine both strategies when we have at least four GPUs.

Keywords: B-chain, addition Chain, addition-subtraction chain, addition-multiplication chain, Branch-and-Bound, GPU, CUDA, Depth-First Strategy, Breadth-First Strategy

1 Introduction

Given a natural number n, and an element g in some groups G, computing g^n with the minimal number of operations is equivalent to the problem of finding a sequence of elements such that the sequence starts with 1, which represents g, terminates with n, which represents g^n, and each other element in the sequence comes from two preceding elements (not necessarily different) in G using the binary operation defined on G. Formally, let B be a finite set of binary operations over the set of natural numbers. A B-chain [1] for a natural number n, denoted by $BC(n)$, is a sequence $1 = c_0, c_1, \ldots, c_l = n$, such that for each $i > 0$, $c_i = c_j \circ c_k$, where $0 \leq j,k \leq i-1$ and $\circ$ is an operation of B. The number l is called the length of $BC(n)$. A $BC(n)$ is called a shortest if its length is minimal. Let $\ell^B(n)$ denotes the length of a shortest B-chain.

Designing an efficient algorithm to generate a shortest B-chain plays an important role in increasing the efficiency of some public key cryptosystems and protocols [2,3,4,5] that used the operation g^n in their computations.

A B-chain is considered a mathematical model for studying the complexity of evaluating integers and polynomials [1,6]. The most important types of B-chains are:

1. *addition chain* [7], denoted by AC or BC^+, when $B = \{+\}$. Generating a shortest addition chain for n plays an important role in speeding up modular exponentiation $g^n \bmod m$, where $g \in Z_m$ is an element in the multiplicative group of integers modulo a positive integer m.
For example, computing g^{51} using $BC^+(51) : 1,2,4,8,16,32,48,50,51$ can be done as follows:
$g,\ g^2 = g*g,\ g^4 = g^2*g^2,\ g^8 = g^4*g^4,\ g^{16} = g^8*g^8,\ g^{32} = g^{16}*g^{16},\ g^{48} = g^{32}*g^{16},\ g^{50} = g^{48}*g^2,\ g^{51} = g^{50}*g.$
This computation requires 8 multiplications, while the following computation requires 7 multiplications using the $BC^+(51) : 1,2,3,6,12,24,48,51$.
$g,\ g^2 = g*g,\ g^3 = g^2*g,\ g^6 = g^3*g^3,\ g^{12} = g^6*g^6,\ g^{24} = g^{12}*g^{12},\ g^{48} = g^{24}*g^{24},\ g^{51} = g^{48}*g^3.$

2. *addition-subtraction chains* [7], denoted by ASC or $BC^{\pm}$, when $B = \{+,-\}$, i.e., each element in a chain can be written as summation or subtraction of two previously elements $c_i = c_j \pm c_k, j,k < i$. Addition-subtraction chains [2] are similar to addition chains in that they are used to compute $n \cdot g$, where n is a scalar and g is a point on elliptic curve E over a finite field $\mathbb{F}$. For example, if $n = 63$, then we can find $BC^{\pm}(n) : 1,2,4,8,16,32,64,63$. The last element

* Corresponding author e-mail: h.m.bahig@gmail.com, hmbahig@sci.asu.edu.eg

$c_7 = c_6 - c_0$. Thus, computing $63 \cdot P$ can be done as follows: $P,\ 2 \cdot P,\ 4 \cdot P,\ 8 \cdot P,\ 16 \cdot P,\ 32 \cdot P, 64 \cdot P, 63 \cdot P$, where $63 = 64 - 1$. Note that a shortest $BC^+(63)$ is $1,2,3,6,12,15,30,60,63$, and so $\ell^+(63) \geq \ell^{\pm}(63)$. In general, $\ell^{\pm}(n) \leq \ell^+(n.)$

3. *addition-multiplication chains* [8,9,10], denoted by *AMC* or BC^+_*, when $B = \{+,*\}$, i.e., each element in a chain is a summation or multiplication of two previously elements $c_i = c_j *^+ c_k, j,k < i$. For example, $BC^+_*(63)$: $1,2,3,6,7,9,63$, and so $\ell^+_*(63) \leq \ell^{\pm}(63) \leq \ell^+(63.)$ Clearly, $\ell^+_*(n) \leq \ell^+(n.)$

4. *Euclidean addition chains* [11], denoted by *EAC*. It is a special case of addition chains where $c_2 = 3$, and for $2 \leq i \leq l-1$, if $c_i = c_{i-1} + c_j$ for some $j \leq i-2$, then $c_{i+1} = c_i + c_{i-1}$ or $c_{i+1} = c_i + c_j$. *EAC* has application in performance of some elliptic curve cryptosystem [12]. Herbaut and P. Véron [13] proposed a public key cryptosystem with security based on *EAC*. Efficient generation of *EAC* may lead to the cryptanalysis of Herbaut-Véron cryptosystem.

In general, generating a shortest *B*-chain is NP-hard problem [14,15]. There are two directions to generate *BC*. The first is to generate a short *BC*, while the other is generate a shortest *BC*. In this paper, we concentrate on a shortest *BC*. From practical view, generating a shortest *B*-chain is important when n is not very large or it is a fixed number for a period of time. Otherwise, one can generate a short *B*-chain [3,7,16]

The majority of studies in the literature have focused on generating a shortest BC^+. For examples, Thurber [17] developed a fast branch and bound depth first search (BB-DFS) algorithm to find a BC^+ by presenting three (pruning bounds) bounding sequences and two types of pruning techniques to cut off some elements in the search tree that cannot lead to a BC^+. Bahig [18,19] improved Thurber's work by determining some conditions for a step c_i to be in the form $c_i = c_{i-1} + c_j, j < i$, and the lower bound of j, and k when we generate $c_i = c_j + c_k, j,k < i-1$. Thurber and Clift [20] generalized two purring bounds of Thurber's result [17]. Bahig and AbdElbari [21] proposed a GPU-based algorithm to generate a shortest BC^+.

On other sides, a few works have been done on generating shortest BC^+_* [8,9], and *EAC* [12,13].

Parallel computing [22,23,24] is used to speedup generation of a shortest or short BC^+. Graphics processing units (GPUs) play a main role in parallel computing in different domains, such as cryptanalysis [25], and bioinformatics [26].

The purposes of this paper are (1) uses of GPUs to present a general algorithm to generate any type of *B*-chains with minimal length; and (2) proposing two strategies to speed up the generation using multi-GPUs and multi-threads.

Compared to using a single GPU, the two proposed strategies accelerate the generation by about 20, and $30 \sim 35\%$ respectively.

The remainder of the paper is organized as follows. In Section 2, we present a general algorithm to generate a shortest *B*-chain. It is a generalization of the algorithm developed by Bahig and AbdElbari [21] to generate a shortest addition chain. The proposed algorithm can work on any type of *B*-chain. In Section 3, we propose the first strategy to use multi-GPUs to increase the performance of generating a shortest *B*-chain. In Section 4, we propose the second strategy. Section 5 describes the implementation details of the two strategies. Finally, Section 6 includes the conclusion of the paper.

2 Generating a Shortest B-Chain using GPU

In this section, we present a general algorithm to generate a shortest *B*-chain. It is a generalization of the algorithm proposed by Bahig and AbdElbari [21]. The algorithm starts with computing a lower bound ,lb, of *BC*, and then generating a short *BC*. Thus, the length of the generated short *BC* is the upper bound of the depth of the search tree, i.e., if no a *BC* with length $lB < ub$ is found, then the generated short *BC* is shortest and so the algorithm terminates. The search tree is divided into three parts (subtrees) as shown in Fig. 1:

1. Top tree (**TTree**) which employs the central processing unit, CPU, to perform branch and bound depth first search strategy (**BB-DFS**) see Algorithm 1.
2. Middle tree (**MTree**) which employs the CPU to perform the branch and bound breadth first search algorithm (**BB-BFS**), see Algorithm 2.
3. Bottom tree (**BTree**) which employs the GPU to perform the branch and bound depth first search strategy (**BB-DFS**), see Algorithm 3.

The detailed description of each part is similar to that described in [21] but for $B = \{+\}$. Algorithm 1 describes the search tree to generate a shortest *BC*, where

- B refers to the type of *B*-chains.
- lower-bound-BC(n,B) returns a lower bound of $BC(n)$. For example, if $B = \{+\}$, then we have [7,27]

$$lb \geq \lceil \log_2 n + \log_2 HW(n) - 2.13 \rceil,$$

where HW denotes to the Hamming weight, i.e., number of "1" bit in the binary representation of n. While if $B = \{+,*\}$, then we have [8]

$$lb \geq \log_2 \log_2 n + 1.$$

Similarly for $B = \{+,-\}$, see [28].

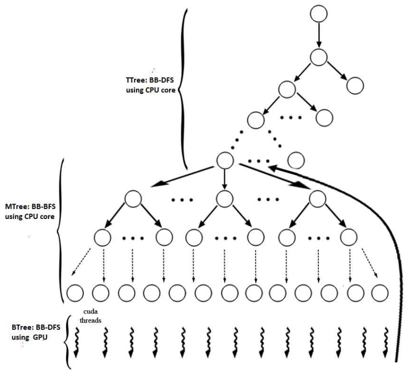

Fig. 1: **GSBC**: General strategy to generate a shortest *BC* using GPU

- upper-bound-BC(n,B) returns a short B-chain, and its length. In case of $B=\{+\}$, we can use one of the methods that generates a short addition chain such as [7,29,30]. Similarly, for $B=\{+,*\}$, we can use the r-ary method [8], while the methods in [2,31] for $B=\{+,-\}$.
- Bounding-sequences(n,B,lb) returns bounding sequences, if exists, for n with length lb, where a bounding sequence (or prune bounds) is a sequence of numbers $\{p_i\}_{i=0}^{lb}$ of length lb to determine a lower bound of each c_i in any B-chain of length lb, i.e., $c_i \geq p_i, 0 \leq i \leq lb$. In the literature, bounding sequences for BC^{+} are proposed by Thurber [17] and improved by Thurber and Clift [20], while Bahig [9] presented a bounding sequence for BC^{+}_{*}. Until now, there is no proposed bounding sequence for $BC^{\pm}$. The main difficulty in finding a bounding sequence for $BC^{\pm}$ is that it is not increasing sequence.
- *DetermineDepthLevel*(n,B,lb) returns the estimated depth of **TTree**, see [21] for example. The estimation of *DepthLevel* should consider the available memory storage, otherwise we need to use another strategy, such as in [25], for **MTree**.
- *DStack* is a stack to hold each element and its level in the search tree using **DFS**.

3 The First Strategy

In this section, we present the first strategy to improve the generation of a shortest B-chain using multi-GPUs. The strategy is based on using two (or more) GPUs at **BTree**. When the number $|QueueElem\_GPU|$ of generated children (paths) using **MTree** is sufficiently large, we distribute the generated elements *QueueElem_GPU* to some or all available GPUs such that each of them has sufficient data to work efficiently. Let α denotes the number of available GPUs, and β denotes the minimum number of elements (paths) to occupy each GPU assuming that all GPUs have the same specification. The

Algorithm 1 **GSBC: Generate a shortest B-chain for** n

Ensure: BC : shortest $BC(n)$.
$CurBC[0] \leftarrow 1$;
$CurBC[1] \leftarrow 2$;
$lb \leftarrow$ lower-bound-BC(n,B);
$BC, ub \leftarrow$ upper-bound-BC(n,B);
while $(lb < ub)$ **do**
  $BS \leftarrow$ Bounding-sequences(n,B,lb); ▷ **TTree: BB-DFS**
  $DepthLevel \leftarrow DetermineDepthLevel(n,B,lb)$;
  $Curlevel \leftarrow 1$;
  loop
    if $CurLevel < DepthLevel$ **then**
      $DStack \leftarrow$ Push all possible children of $CurBC[0..CurLevel]$ (associated with their levels $CurLevel+1$ and) generated by $op \in B$ and retained by BS;
    end if
    if $DStack$ is not empty **then**
      $(CurLevel, CurBC[CurLevel]) \leftarrow$ Pop$(DStack)$;
    else
      Exit the inner loop;
    end if
    if $CurLevel = DepthLevel$ **then** ▷ **MTree: BB-BFS**
      $(QueueElem\_GPU, CurLevel) \leftarrow Generate\_Child\_BBBFS(CurBC, CurLevel, lb, n, BS, B)$;
      if $QueueElem\_GPU$ is not empty **then**
        if $CurLevel = lb$ **then**
          return a shortest $BC(n)$;
        else ▷ **BTree: BB-DFS**
          $GPU\_BBDFS(QueueElem\_GPU, CurLevel, FoundBC, BC, n, BS, B)$
          if $FoundBC = true$ **then**
            return BC
          end if
        end if
      end if
    end if
  end loop
  $lb = lb + 1$;
end while
return BC

Algorithm 2 $Generate\_Child\_BBBFS(CurBC, CurLevel, lb, n, BS, B)$

Ensure: $List\_Paths\_for\_GPU$
$List\_Paths\_for\_GPU \leftarrow insert(CurBC)$
while $CurLevel < lb$ and $List\_Paths\_for\_GPU$ is not empty **do**
  $CurLevel \leftarrow CurLevel + 1$;
  $List\_Paths\_for\_GPU \leftarrow$ Generate all possible children for each path in $List\_Paths\_for\_GPU$
  if the conditions for GPU are satisfied **then**
    return $List\_Paths\_for\_GPU$;
  end if
end while
return $List\_Paths\_for\_GPU$;

Algorithm 3 GPU_BBDFS $(ListPaths\_GPU, CurLevel, FoundBC, BC, n, BS, B)$

Require: FoundBC, BC;
  Each GPU thread has the following local variables:
  $ThrdStack$: stack to hold elements and their levels;
  $ThrdCurPath$: current path;
  $ThrdCurLevel$: current level
start threads sufficient for all elements of $ListPaths\_GPU$;
$ThrdCurPath \leftarrow ListPaths\_GPU[ThreadUniqeID]$;
$ThrdCurLevel \leftarrow CurLevel$;
repeat parallely
  each thread push children of $ThrdCurPath[0..ThrdCurLevel]$ in $ThrdStack$ (associated with ThrdCurLevel+1) ;
  if $ThrdStack$ is not empty (i.e., $ThrdCurLevel > CurLevel$)) **then**
    $(ThrdCurLevel, ThrdCurBC[ThrdCurLevel]) \leftarrow$ Pop$(ThrdStack)$;
  else
    End of the thread ;
  end if
  if $ThrdCurLevel = lb$ **then**
    FoundBC= true;
    Atomically make $BC = ThrdCurBC$;
    Announce all other threads to stop;
  end if
until $ThrdStack$ is empty
FoundBC= false;

QueueElem_GPU generated from **TTree** is distributed to GPUs as follows:

$\gamma \leftarrow \alpha$.
while ($\frac{|QueueElem\_GPU|}{\gamma} < \beta$) do
$\gamma \leftarrow \gamma - 1$
end while
Copy about $\lceil \frac{|QueueElem\_GPU|}{\gamma} \rceil$ to each GPU.

Fig. 2 shows the idea of this strategy when we have two GPUs. We use CPU-Core-1 for both **TTree** and **MTree** using **BB-DFS** and **BB-BFS**, respectively. While we use GPU-1 and GPU-2 for **BTree** using **BB-DFS**.

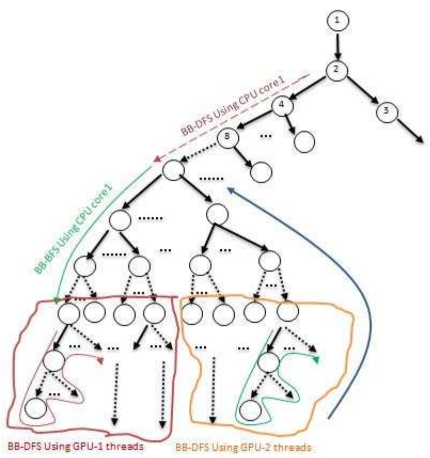

Fig. 2: The first strategy

4 The Second Strategy

In this section, we propose another efficient strategy to improve the generation of a shortest BC using two (or more) GPUs. The strategy is based on our observation that "there is usually a difference between $\ell^B(n)$ and lb". For example, let $n = 170089$. There is no shortest addition chain for n with length $lb = 20$, but there is with the length $lb = 21$. The algorithm **GSBC** tries to find a $BC(n)$ of length lb. If **GSBC** couldn't find a B-chain with the length lb, then it increases lb by 1 and repeats the process. Thus, our strategy is to run **GSBC** using one core and one GPU to find a $BC(n)$ of length lb, and also to run **GSBC** using another core and another GPU to find a $BC(n)$ of length $lb+1$, and so on. Suppose that we have two GPUs. Our strategy is to run at the same time **GSBC** using two cores (say, core-1, and core-2) and two GPUs (say, GPU-1, and GPU-2). The CPU core-1 with GPU-1 tries to find a $BC(n)$ of length lb, while the CPU Core-2 with GPU-2 tries to find a $BC(n)$ of length $lb+1$. There are two cases:

1. If Core-1 (with GPU-1) finished before core-2 (with GPU-2). In this case, we have two subcases:
   (a) Core-1 found a shortest B-chain. Thus, **GSBC** terminates core-2 and returns the shortest B-chain.
   (b) There is no $BC(n)$ of length lb. Thus, **GSBC** waits until core-2 (with GPU-2) ends. If core-2 finds a $BC(n)$ of length $lb+1$, then **GSBC** returns the founded chain as a shortest B-chain. Otherwise, i.e., core-2 didn't find a shortest $BC(n)$, we have to start core-1 with depth $lb+2$, and core-2 with depth $lb+3$, and repeat the process. Note that, in the case where the depth of the search tree is equal to ub, then there is no need to continue the search.
2. Otherwise, core-2 (with GPU-2) finished before core-1. In this case, core-2 should wait until core-1 terminates, and then we have two subcases:
   (a) If core-1 finds a $BC(n)$ of length lb, then it returns the shortest $BC(n)$.
   (b) Otherwise, core-2 returns a shortest $BC(n)$ of length $lb+1$, if one exists. If both cores didn't find a $BC(n$, then repeat the process, i.e., core-1 (with GPU-1) searches for a $BC(n)$ of length $lb+2$, and core-2 (with GPU-2) searches for a $BC(n)$ of length $lb+3$.

Fig. 3 shows our main idea.

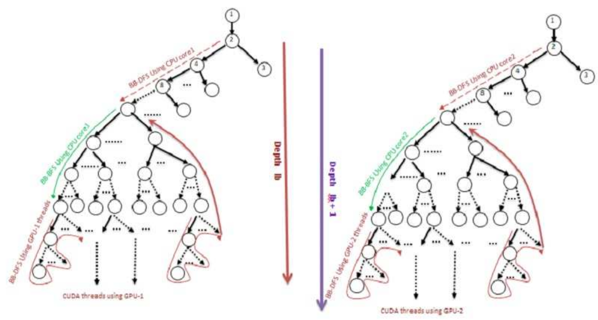

Fig. 3: The second strategy

5 Experimental Results

This section describes our implementation of the two strategies. We choose $B = \{+\}$. The implementation was conducted on a PC with Intel core i7-10870H CPU 2.21 GHz, 16 GB RAM, Windows 10 and Visual Studio 2013. The PC is coupled with two NVIDIA GeForce GTX 3060 [32]. The programs are written in the CUDA C. The CUDA C is an extension of the standard C language that used to access the GPU. We take $\alpha = 2$, $\beta = 1024$.

Table 1 shows the performance of the proposed strategies for 200 random m-bit numbers, where

$m = 16, 18, 20, 22, 24$.

The first strategy speeds up the generation by about $19 \sim 21\%$, while the second strategy speeds up the generation by $30 \sim 35\%$. Both strategies increase with an increasing the number of bits in the input number n, and, in particular, with an increasing Hamming weight of n. The performance of the second strategy increases more than the first since usually the difference between $\ell^B(n)$ and lb increases as n increases and so the second strategy will be more effective.

Table 1: Comparison in times (secs) between using a single GPU and using 2-GPUs with different strategies

Algorithm **GSBC**	m–bits				
	16	18	20	22	24
single GPU	0.88	3.46	27.83	193.47	386.01
2-GPUs + Strategy 1	0.71	2.75	22.01	152.48	304.7
2-GPUs +Strategy 2	0.62	2.39	18.95	129.41	251.17

6 Conclusion

We have presented a general (generic) algorithm for generating a shortest B-chain using GPU. It can be used to generate any type of B-chain. Then, we have suggested two strategies to improve the generation using multi-GPUs. The experimental results show that using two GPUs, the first strategy reduces the average time by about 20%, while the second strategy reduces the average time by about $30 \sim 35\%$ compared to using a single GPU. It is possible to combine the two strategies to get more performance, but this requires two cores and four GPUs.

Acknowledgement

This work was supported financially by the Academy of Scientific Research and Technology (ASRT), Egypt, Grant No 6419, ScienceUp.
The authors are grateful to the anonymous referee for valuable comments.

Competing interests:

The authors declare that they have no competing interests.

References

[1] R. Lipton, D. Dobkin. Complexity measures and hierarchies for the evaluation of integers and polynomials. Theoret. Comput. Sci. 3, 349–357 (1976).
[2] F. Morain, J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. Informatique théorique et applications. 24 (6): 531–543 (1990).
[3] Alfred Menezes, Paul C. van Oorschot and Scott A. Vanstone, Handbook of Applied Cryptography. Ch. 14, CRC Press (1996).
[4] R. Rivest, A. Shamir, L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM. 21(2), 120–126 (1978).
[5] N. Koblitz, Elliptic curve cryptosystems, Mathematics of Computation 48 , 203–209 (1987).
[6] W. De Melo, B.F. Svaiter. The cost of computing integers. Proc.Amer. Math. Soc. 124 (5), 1377–1378 (1996).
[7] D. E. Knuth. Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition) , Addison-Wesley, Reading MA. 461–485 (1997).
[8] Hatem M. Bahig. On a generalization of addition chains: Addition–multiplication chains. Discrete Mathematics 308, 611–616 (2008).
[9] H Bahig, and A Mahran, Efficient generation of shortest addition–multiplication chains, Journal of the Egyptian Mathematical Society, Vol. 26, no. 3, 509–521 (2018)
[10] D. Dobkin, R. Lipton. Addition Chain Methods for the Evaluation of Specific Polynomials. SIAM J. Comput. Vol. 9, 121–125 (1980).
[11] Nicolas Méloni. New Point Addition Formulae for ECC Applications. WAIFI: Workshop on the Arithmetic of Finite Fields, Jun 2007, Madrid, Spain. 189–201 (2007).
[12] F. Herbaut, N. Meloni, P. Véron. Compact variable-base ECC scalar multiplication using Euclidean addition chains. 18th International Conference on Security and Cryptography (SECRYPT 2021), Jul 2021, Online Event, Italy. 531–539 (2021).
[13] F. Herbaut, P. Véron. A public key cryptosystem based upon euclidean addition chains. In C. Carlet and A. Pott, editors, Sequences and Their Applications - SETA 2010 - 6th International Conference, Paris, France, September 13-17, 2010. Proceedings, volume 6338 of Lecture Notes in Computer Science, 284–297. Springer, (2010).
[14] P. Downey, B. Leong, R. Sethi. Computing sequences with addition chains. SIAM J Comput. 10(3), 638–646 (1981).
[15] Hatem M. Bahig, Hazem M. Bahig. A new strategy for generating shortest addition sequences. Computing 91, 285–306 (2011).
[16] D.M. Gordon. A survey of fast exponentiation methods. J Algorithms, Vol. 122, 129–146 (1998)
[17] E.G. Thurber. Efficient generation of minimal length addition chains. SIAM J. Computing. 28, 1247–1263 (1999).
[18] Hatem M. Bahig. Improved generation of minimal addition chains. Computing. 78, 161–172 (2006).
[19] Hatem M. Bahig. Star reduction among minimal length addition chains. Computing. 91, 335–352 (2011).
[20] E.G. Thurber, N.M. Clift. Addition chains, vector chains, and efficient computation. Discrete Mathematics, Vol. 344, No. 2, 112200 (2021).
[21] Hatem M. Bahig, K. A. AbdElbari. A fast GPU-based hybrid algorithm for addition chains. Cluster Comput 21, 2001–2011 (2018).
[22] Hazem M. Bahig. A fast optimal parallel algorithm for a short addition chain. J. Supercomput., 74 (1), 324–333 (2018).

[23] Hazem M. Bahig, Yasser Kotb. An Efficient Multicore Algorithm for Minimal Length Addition Chains. Comput., 8(1), 23, (2019).
[24] K. Fathy, Hazem M. Bahig, A. Ragab. A Fast Parallel Modular Exponentiation Algorithm. Arab J Sci Eng 43, 903–911 (2018).
[25] M. S. Esseissah, A. Bhery, H. M. Bahig. Improving BDD Enumeration for LWE Problem Using GPU. IEEE Access, Vol. 8, 19737–19749 (2020).
[26] J.L. Payne, N.A. Sinnott-Armstrong, J. H. Moore. Exploiting graphics processing units for computational biology and bioinformatics. Interdiscip Sci Comput Life Sci 2, 213–220 (2010).
[27] A. Schnhage. A lower bound for the length of addition chains. Theoretical Computer Science. 1, 1–12 (1975).
[28] Hugo Volger. Some results on addition/subtraction chains. Information Processing Letters, 20 (3), 155–160 (1985).
[29] F. Bergeron, J. Berstel, S. Brlek, C. Duboc. Addition chains using continued fractions. Journal of Algorithms, 10(3), 403–412 (1989).
[30] Hazem M. Bahig, Khaled A. Alutaibi, Mohammed A. Mahdi, Amer AlGhadhban and Hatem M. Bahig. An Evolutionary Algorithm for Short Addition Chains International Journal of Advanced Computer Science and Applications(IJACSA), 11(12), (2020).
[31] Raveen R. Goundar, Ken-ichi Shiota, Masahiko Toyonaga, New Strategy for Doubling-Free Short Addition-Subtraction Chain, Applied Mathematics and Information Sciences, an international journal, Vol. 2, No. 2 , 123–133 (2008).
[32] NVIDIA. NVIDIA CUDA C Programming Guide, CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/index.html