User Guide for BoolSPLG CUDA Library

Dushan Bikov *
dusan.bikov@ugd.edu.mk

Iliya Bouyukliev †
iliyab@math.bas.bg

VERSION 0.3, updated March 13, 2023

Abstract

This manual accompanies the article entitled "BoolSPLG: A library with parallel algorithms for Boolean functions and S-boxes for GPU". Here we present a library with parallel functions for computing some of the most important cryptographic characteristics of Boolean and vectorial Boolean functions. These functions can be used to develop efficient algorithms. The library is implemented using CUDA parallel programming model for recent NVIDIA GPU architectures.

Keywords Mathematical software, CUDA C, CUDA Library, GPU, Butterfly algorithms, Cryptographic properties, Boolean function, Vectorial Boolean function, Parallel Algorithms and Computing.

Contents

1 Introduction

The main objects which we consider are Boolean and vectorial Boolean functions (S-boxes) with good cryptographic properties. There is a lot of research on S-boxes in eight or fewer variables [13], but not much is known about larger S-boxes, despite the interest in them [8], [5], [9]. One of the reasons for this is the computationally difficult evaluation of their cryptographic properties. Construction of such kind of objects is very important but in most of the cases computationally expensive problem.

Cryptographic parameters which we investigate are nonlinearity, algebraic degree, autocorrelation, differential uniformity. The computation of these parameters is connected to Fourier-related transformation like Walsh-Hadamard, Reed-Muller(Möbius) Transform and Inverse Walsh-Hadamard [1]. The algorithms, known as butterfly algorithms, that implement these fast discrete transforms are very efficient [6]. Moreover, these algorithms are suitable for parallelization in SIMD (Single Instruction, Multiple Data) computer architectures. Nowadays using of modern graphics processing units (GPUs) and CUDA (Compute Unified Device Architecture)

*Faculty of Computer Science, Goce Delchev University, Shtip, Macedonia
†Institute of Mathematics and Informatics, Veliko Tarnovo, Bulgaria

[12] for this type of parallelization is naturale and very effective. Contemporary NVIDIA GPUs are powerful computing platform developed for general purpose computing using CUDA.

For our research, we develop a library BoolSPLG with C/C++ functions. It can be used to study important cryptographic properties of Boolean functions of n variables and bijective $n \times n$ S-boxes for $n \leq 20$. All basic functions have two versions - sequential and parallel. All these features can be used for project development by anyone who knows the C/C++ programming language but is not so familiar with CUDA C. The structure of BoolSPLG is presented in fig. 1.

In addition, BoolSPLG can be used for training research, to view the characteristics of the video card and to compare parallel and sequential performance. In short, the way BoolSPLG works is presented in the examples that are part of the library distribution.

2 Installing BoolSPLG

BoolSPLG library is implemented as a C++ header library. The current version of the BoolSPLG Library requires CUDA 9.x and above. There is no need to "build" BoolSPLG library separately. To be able to use BoolSPLG primitives in your code, simply:

1. Clone the repository from *GitHub* to your local computer. Cloning the BoolSPLG-v0.3 library resource folder:

   $cd "directory"
   $ git clone https://github.com/BoolSPLG/BoolSPLG-v0.3
   ...
   $

   The newly created BoolSPLG-v0.3 folder in your HOME is the place where is install the BoolSPLG library resources.

   You need to install (copy) only "BoolSPLG" subdirectory from the main BoolSPLG-vx.x directory to the CUDA include directory. CUDA include directory usually is:

   - "/usr/include/" on a Linux (Ubuntu);
   - "C:\CUDA\include\" on a Windows.

   If you are unable to install BoolSPLG to the CUDA include directory, then you can place BoolSPLG somewhere in your home directory.

2. #include the "umbrella" <BoolSPLG/BoolSPLG_v03cuh>header file in your CUDA C++ sources;

3. Compile your program with NVIDIA nvcc CUDA compiler, specifying a -I<path-to- BoolSPLG>include-path flag to reference the location of the BoolSPLG header library.

2.1 BoolSPLG and CMake

CMake is free open-source BSD license software [4], powerful cross-platform system of tools designed to build, automate, test, automate, test, package, and install software using a compiler-independent manner [7]. With CMake, the entire BoolSPLG library installation process is automated. In addition to installing the library, CMake can also be used to build the "Examples" programs. The *CMakeLists.txt* configuration placed in each source directory, is used to generate standard compilation files in a different platform environment. The use the latest version of CMake is recommended, or at least CMake 3.21.

The basic CMake setting requires to select the configuration file *CMakeLists.txt* and to choose where to build the binaries. In the main configuration file *CMakeLists.txt* the required version of CUDA is set to 10.0. The necessary version can be changed by changing the argument of the configuration command (CUDA **version** REQUIRED). Note that CMake does not always recognize the correct GPU architecture, this may also be subject of adjustment. By pressing the button "Configure" from the graphic interface the program CMake will show whether there are any configuration problems. If there are no configuration problems, the user may proceed with the generating native *makefiles* or project files by pressing the button "Generate".

In the document **CmakeBoolSPLG.pdf** (in directory docs) the user can read more details about how to use CMake, to install BoolSPLG header library, and to run the Examples.

3 Overview of BoolSPLG Procedures

BoolSPLG (Boolean functions and S-box Parallel Library for GPU) is a library containing functions for analysis and calculation of the most important cryptographic parameters and characteristics of Boolean and vectorial Boolean functions. There are also other additional auxiliary functions.

The functions are organized in several layers. The functions in the first layer provide information about the CUDA textit nvcc compiler and the parameters of the hardware used. Developed functions automatically configure the required grid of threads.

The main algorithms in the library are implemented using a parameterized kernel. There are several types of library functions. The most important of them are for calculating the basic parameters of the considered functions (Butterfly (FWT, IFWT, FMT, bitwise FMT, min-max), DDT, AlgebraicDegree, ComponentFunction, PowerInt). The second type of functions refers to the efficient use of different types of memory (Copy, MemoryPatern). The third type is related to summarizing and analyzing the results or the so-called reduction functions (min, max, sum).

Figure 1 presents a diagram of the different types of functions used in the library and the relationships between them. The solid line indicates dependence, while the dotted line is an optional component.

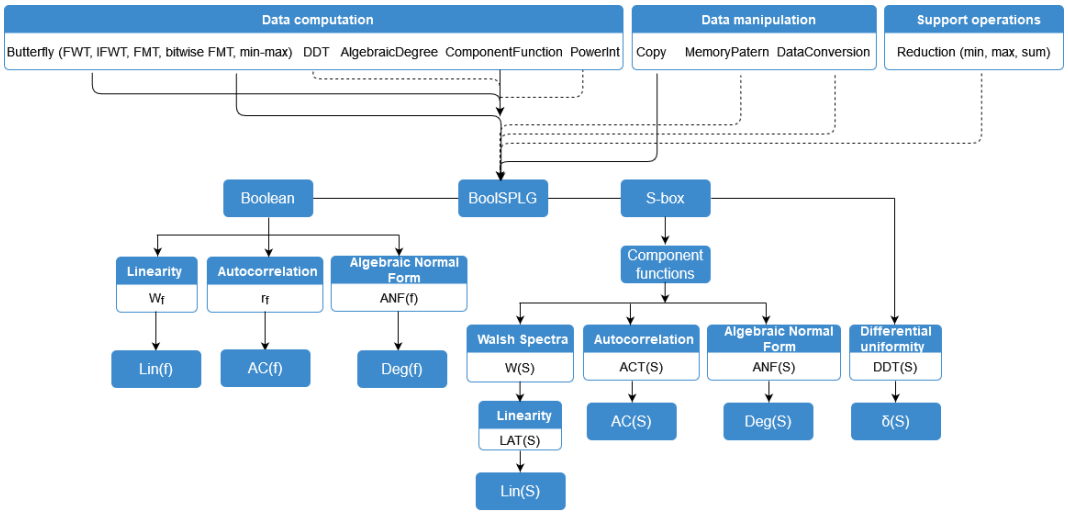

Figure 1: Classification and module dependencies of the building blocks involved in the library

Our library contains the following butterfly algorithms: Fast Walsh Transforms (FWT), Inverse Fast Walsh Transforms (IFWT), Fast Möbius Transform (FMT), Bitwise Fast Möbius Transform (bitwise FMT) and butterfly Min-Max. There are additional functions for computing DDT, component functions and auxiliary functions.

The file BoolSPLG_v03 contains all declarations of host and device functions. At the end of the declaration there is a comment on the version of the library. BoolSPLG computes the following cryptographic parameters of the Boolean and vectorial Boolean functions: $W_f(f)$ (Walsh spectrum of a Boolean function), $Lin(f)$ (Linearity of a Boolean function), $W(S)$ (Walsh Spectrum of an S-box), $LAT(S)$ (Linear Approximation Table of an S-box), $Lin(S)$ (Linearity of an S-box), $r_f(f)$ (Autocorrelation Spectrum of a Boolean function), $AC(f)$ (Autocorrelation of a Boolean function), $ACT(S)$ (Autocorrelation spectrum of an S-box), $AC(S)$ (Autocorrelation of an S-box), $ANF(f)$ (Algebraic Normal Form of a Boolean function), $ANF(S)$ (Algebraic Normal Form of an S-box), $deg(f)$ (Algebraic Degree of a Boolean function), $deg(S)$ (Algebraic Degree), $DDT(S)$ (Difference Distribution Table of an S-box), $\delta(S)$ (Differential uniformity of an S-box) and S_b (a component function of an S-box).

3.1 S-boxes: component functions and tables

In order to study the cryptographic properties of a vectorial Boolean function (S-box) related to the linearity, algebraic degree and autocorrelation, we need to consider all non-zero linear combinations (figure 2) of the coordinate functions of the considered S-box S, denoted by

$$S_b = b \cdot G_S = b_1 f_1 \oplus \cdots \oplus b_m f_m,$$

where $b = (b_1, \ldots, b_m) \in \mathbb{F}_2^m$. These are the component functions of S.

One way to represent the $n \times n$ vectorial Boolean function S that we use is through the Truth Tables of its coordinate functions. Therefore, an $n \times 2^n$ matrix is needed. For convenience, we use the integers corresponding to the binary representation of the columns of this matrix, so S is defined by the integer vector $(s_0, s_1, \ldots, s_{2^n-1})$.

This representation has several advantages: data from the main memory is transferred to GPU memory much faster, and the value of the function S for the input vector $v \in \mathbb{F}_2^n$ corresponds to the v-th coordinate of the array.

Any vector $a \in \mathbb{F}_2^n$ defines the component function S_a as a linear combination of rows of the matrix corresponding to S. The Truth Table of the component function S_a can be found in the following way:

$$TT(S_a) = (a \cdot s_0, a \cdot s_1, \ldots, a \cdot s_{2^n-1}).$$

The functions in the library generate the following tables related to S-boxes: Walsh spectrum table WST(S), linear approximation table LAT(S), autocorrelation table ACT(S), a table with the algebraic degrees of the monomials in the component Boolean functions ADT(S) and difference distribution table (DDT).

The columns of LAT(S), ACT(S) and the rows of ADT(S) are obtained by transforming the Truth Tables or ANF of the component functions. When generating LAT(S), ACT(S) on S-boxes of less than 11 variables, all component functions are generated simultaneously (ComponentFnAll kernels). The necessary transformation is performed simultaneously on the Truth Tables of all component functions considered as one vector. To obtain the LAT(S) and ACT(S), it is necessary to rearrange the data (see figure 2). A similar approach is used to obtain the ADT(S) of an S-box with no more than 10 variables, using the algebraic normal form instead of the Truth Table for each component function. For S-boxes with more variables, the component functions are generated one after the other (ComponentFnVec_kernels).

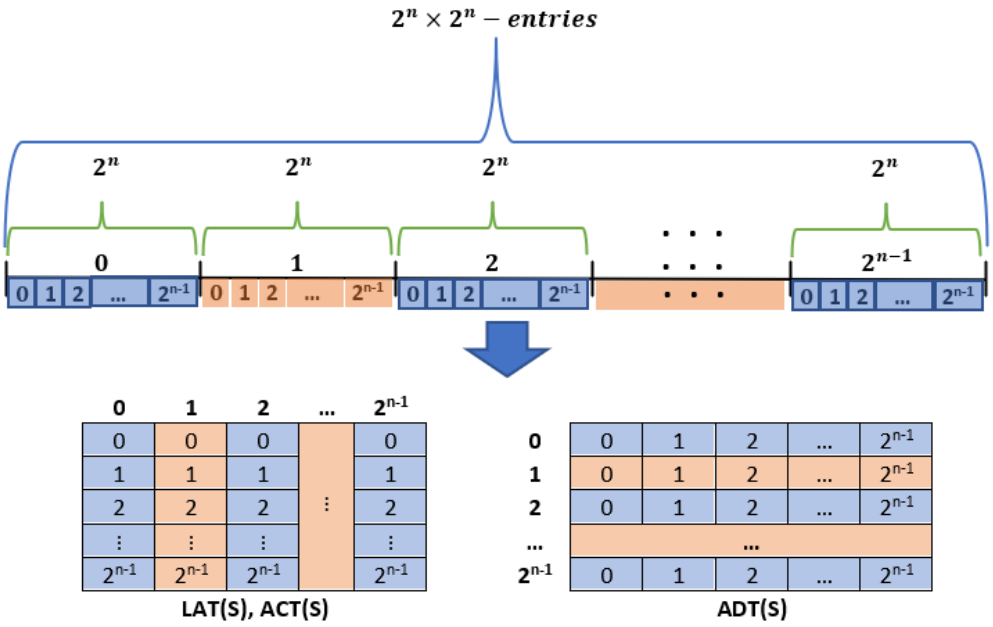

Figure 2: S-box, represented by a memory array consisting of all elements of the component functions, and the necessary redistribution to obtain the tables W(S), LAT(S), and ACT(S)

3.2 Memory Transfer (v0.3)

One of the biggest problems in the CUDA programming model is minimizing data transfer between the host's main memory and the device's global memory [3]. To optimize data transfer, True Table ($TT(f)$) can be considered as a binary representation of unsigned integers. Therefore, the input vector is divided into parts of 32 (or 64) elements which are considered as unsigned integers.

3.3 BoolSPLG basis device functions

Any function is a sequence of kernels that perform different parallel calculations. Appropriate grid configuration is also performed through them.

```
BOOLEAN FUNCTION
================
================
Boolean function f with True Table TT(f),  Walsh Spectrum WS(f), Linearity Lin(f), Algebraic Normal
Form ANF(f), Algebraic Degree AD(f), Algebraic Degrees of the monomials of the Boolean functions ADT(f),
```

Autocorrelation Spectrum AS(f), Autocorrelation AC(f).

```
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect   TT(f) //size 2^n
         bool returnMaxReduction - false or true
//OUTPUT
         int *device_Vect_rez   WS(f) //size 2^n
RETURN Lin(f) if returnMaxReduction is true
//FUNCTION:
int WalshSpecTranBoolGPU(int *device_Vect, int *device_Vect_rez, int size, bool returnMaxReduction); //0.1
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) or ANF(f) //size 2^n
//OUTPUT
         int *device_Vect_rez   ANF(f) or TT(f)  //size 2^n
//FUNCTION:
void MobiusTranBoolGPU(int *device_Vect, int *device_Vect_rez, int size); //0.1
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f)  //size 2^n
//OUTPUT
         int *device_Vect_rez   ADT(f) //size 2^n
RETURN AD(f)
//FUNCTION:
int AlgebraicDegreeBoolGPU(int *device_Vect, int *device_Vect_rez, int size); //0.1
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) //size 2^n
         bool returnMaxReduction - false or true
//OUTPUT
         int *device_Vect_rez   AS(f) //size 2^n
RETURN (AS(f)) if returnMaxReduction is true
//FUNCTION:
int AutocorrelationTranBoolGPU(int *device_Vect, int *device_Vect_rez, int size, bool returnMaxReduction); //0.1
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) //size 2^n
         bool returnMaxReduction  - false or true
//OUTPUT
         int *device_Vect_rez   WS(f) //size 2^n
RETURN Lin(f) if returnMaxReduction is true
//COMMENT Butterfly reduction
//FUNCTION:
int WalshSpecTranBoolGPU_ButterflyMax
(int *device_Vect, int *device_Vect_rez, int size, bool returnMaxReduction); //v0.2
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f)  //size 2^n
//OUTPUT
         int *device_Vect_rez   ADT(f) //size 2^n
RETURN AD(f)
//COMMENT Butterfly reduction
//FUNCTION:
int AlgebraicDegreeBoolGPU_ButterflyMax(int *device_Vect, int *device_Vect_rez, int size); //v0.2
=============================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) //size 2^n
         bool returnMaxReduction - false or true
//OUTPUT
        int *device_Vect_rez   AS(f) //size 2^n
RETURN (AS(f)) if returnMaxReduction is true
//COMMENT Butterfly reduction
//FUNCTION:
int AutocorrelationTranBoolGPU_ButterflyMax
(int *device_Vect, int *device_Vect_rez, int size, bool returnMaxReduction); //v0.2
```

```
================================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) //size 2^n/32
         bool returnMaxReduction - false or true
//TEMPORARY
         T* device_Vect_Max   (short int or int)  //size 2^n
//OUTPUT
         T* device_Vect_rez   (short int or int) AS(f) //size 2^n
RETURN (Lin(f)) if returnMaxReduction is true
//COMMENT use template
//FUNCTION:
template <class T>
int WalshSpecTranBoolGPU_ButterflyMax_v03
(int* device_Vect, T* device_Vect_rez, T* device_Vect_Max, int size, bool returnMaxReduction); //v0.3
================================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) or ANF(f) //size 2^n/32
//OUTPUT
         T* device_Vect_rez   (short int or int) ANF(f) or TT(f) //size 2^n
//COMMENT use template
//FUNCTION:
template <class T>
void MobiusTranBoolGPU_v03(int* device_Vect, T* device_Vect_rez, int size); //v0.3
================================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) //size 2^n/32
//OUTPUT
         T* device_Vect_rez - (short int or int) ADT(f) //size 2^n
RETURN AD(f)
//COMMENT use template
template <class T>
int AlgebraicDegreeBoolGPU_ButterflyMax_v03(int* device_Vect, T* device_Vect_rez, int size); //v0.3
================================================================================================================
//INPUT  int size - 2^n
         int *device_Vect - TT(f) //size 2^n/32
         bool returnMaxReduction - false or true
//TEMPORARY
         T* device_Vect_Max   (short int or int)  //size 2^n
//OUTPUT
         T* device_Vect_rez   (short int or int) AS(f) //size 2^n
RETURN (AC(f)) if returnMaxReduction is true
//COMMENT use template
//FUNCTION:
template <class T>
int AutocorrelationTranBoolGPU_ButterflyMax_v03
(int* device_Vect, T* device_Vect_rez, T* device_Vect_Max, int size, bool returnMaxReduction); //v0.3
================================================================================================================
//INPUT  int size - 2^n
         unsigned long long int *device_Vect - TT(f) or ANF(f) //size 2^n/64
//OUTPUT
         unsigned long long int *device_Vect_rez   ANF(f) or TT(f)  //size 2^n/64
//COMMENT Bitwise
//FUNCTION:
void BitwiseMobiusTranBoolGPU(unsigned long long int *device_Vect,
unsigned long long int *device_Vect_rez, int size); //v0.2
================================================================================================================
//INPUT  int size - 2^n
         unsigned long long int *device_Vect - TT(f) //size 2^n/64
//TEMPORARY
         int *device_Vec_max_values // size 2^n/64
         int *host_Vec_max_values   // (if 2^n/64 < 256), size 2^n/64
//OUTPUT
         unsigned long long int *device_Vect_rez  ANF(f) //size 2^n/64
RETURN AD(f)
//COMMENT Bitwise
//FUNCTION:
```

```
int BitwiseAlgebraicDegreeBoolGPU_ButterflyMax(unsigned long long int *device_Vect,
unsigned long long int *device_Vect_rez, int *device_Vec_max_values, int *host_Vec_max_values, int size); //v0.2
=============================================================================================================

VECTOR BOOLEAN FUNCTION
=======================
=======================
Vector boolean function S with matrix of coordinate function MCorF(S)
(integer array of size 2^n - each column is a binary representation of the corresponding integer),
Walsh Spectrum Table WST(S, Linear Approximation Table LAT(S), Linearity Lin(S),
Algebraic Normal Form of S-box ANF(S), Algebraic Degree of S-box ADmax(S); ADmin(S),
Algebraic Degrees Table ADT(S), Difference Distribution Table DDT(S), Differential Uniformity DU(S),
Autocorrelation Table ACT(S), Autocorrelation AC(S), Table of Component functions TCom(S).

=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_WST   WST(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_WST    //size 2^n
         int *device_CF     //size 2^n
RETURN Lin(S)
//FUNCTION:
int WalshSpecTranSboxGPU(int *device_Sbox, int *device_CF, int *device_WST, int sizeSbox); //0.1
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_ANF   ANF(S)   //size 2^n x 2^n
         int *device_CF    TCom(S)  //size 2^n x 2^n
    ELSE
         int *device_ANF    //size 2^n
         int *device_CF     //size 2^n
RETURN ADmax(S)
//FUNCTION:
int MobiusTranSboxADGPU(int *device_Sbox, int *device_CF, int *device_ANF, int sizeSbox); //0.1
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_ACT   ACT(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_ACT    //size 2^n
         int *device_CF     //size 2^n
RETURN AC(S)
//FUNCTION:
int AutocorrelationTranSboxGPU(int* device_Sbox, int* device_CF, int* device_ACT, int sizeSbox); //0.1
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
         bool returnMax - false or true
//OUTPUT
//IF n<11 THEN
         int *device_LAT   WST(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_WST    //size 2^n
         int *device_CF     //size 2^n
RETURN IF (returnMax) Lin(S)
//COMMENT Butterfly reduction
```

```
//FUNCTION:
int WalshSpecTranSboxGPU_ButterflyMax
(int *device_Sbox, int *device_CF, int *device_WST, int sizeSbox, bool returnMax); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
         bool returnMax - false or true
//OUTPUT
//IF n<11 THEN
         int* device_WST   WST(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_WST   //size 2^n
         int *device_CF    //size 2^n
RETURN IF (returnMax) Lin(S)
//COMMENT Butterfly reduction
//FUNCTION:
int WalshSpecTranSboxGPU_ButterflyMax_v03
(int* device_Sbox, int* device_CF, int* device_WST, int sizeSbox, bool returnMax); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_LAT   LAT(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         printf("The output data is to big.\n");

//FUNCTION:
void LATSboxGPU_v03(int* device_Sbox, int* device_CF, int* device_LAT, int sizeSbox); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_ANF   ANF(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_ANF    //size 2^n
         int *device_CF     //size 2^n
//FUNCTION:
void MobiusTranSboxGPU(int *device_Sbox, int *device_CF, int *device_ANF, int sizeSbox); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
         bool returnMax - false or true
//OUTPUT
//IF n<11 THEN
         int *device_ACT   ACT(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_ACT    //size 2^n
         int *device_CF     //size 2^n
RETURN IF (returnMax)  AC(S)
//COMMENT Butterfly reduction
//FUNCTION:
int AutocorrelationTranSboxGPU_ButterflyMax(int *device_Sbox, int *device_CF, int *device_ACT,
int sizeSbox, bool returnMax); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int* device_DDT   DDT(S) //size 2^n x 2^n
    ELSE
```

```
         int* device_DDT    //size 2^n
RETURN DU(S)
//FUNCTION:
int DDTSboxGPU(int* device_Sbox, int* device_DDT, int sizeSbox); //0.1
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
         bool returnMax - false or true
//OUTPUT
//IF n<11 THEN
         int* device_DDT   DDT(S) //size 2^n x 2^n
    ELSE
         int* device_DDT    //size 2^n
RETURN IF (returnMax) DU(S)
//COMMENT Butterfly reduction
//FUNCTION:
int DDTSboxGPU_ButterflyMax(int* device_Sbox, int* device_DDT, int sizeSbox, bool returnMax); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
         bool returnMax - false or true
//OUTPUT
//IF n<11 THEN
         int* device_DDT   DDT(S) //size 2^n x 2^n
    ELSE
         int* device_DDT    //size 2^n
RETURN IF (returnMax) DU(S)
//COMMENT Butterfly reduction
//FUNCTION:
int DDTSboxGPU_ButterflyMax_v03(int* device_Sbox, int* device_DDT, int sizeSbox, bool returnMax); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
         bool returnMax - false or true
//OUTPUT
//IF n<15 THEN
         int* device_DDT   DDT(S) //size 2^n x 2^n
RETURN IF (returnMax) DU(S)
//COMMENT Butterfly reduction
//FUNCTION:
int DDTSboxGPU_ButterflyMax_v03_expand(int* device_Sbox, int* device_DDT, int sizeSbox, bool returnMax); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_ANF   ADT(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_ANF    //size 2^n
         int *device_CF     //size 2^n
RETURN  ADmax(S)
//COMMENT Butterfly reduction
//FUNCTION:
int AlgebraicDegreeSboxGPU_ButterflyMax(int *device_Sbox, int *device_CF, int *device_ANF, int sizeSbox); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_ANF   ADT(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         int *device_ANF    //size 2^n
         int *device_CF     //size 2^n
RETURN  ADmin(S)
```

```
//COMMENT Butterfly reduction
//FUNCTION:
int AlgebraicDegreeSboxGPU_ButterflyMin(int* device_Sbox, int* device_CF, int* device_ANF, int sizeSbox); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* device_Sbox_ANF - ANF(S) //size 2^n
//TEMPORARY
  //IF n<11 THEN
          int* device_CF //size 2^n x 2^n
    ELSE
          int* device_CF //size 2^n
RETURN ADmax(S)
//FUNCTION:
int AlgebraicDegreeSboxGPU_in_ANF_ButterflyMax(int* device_Sbox_ANF, int* device_CF, int sizeSbox); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* device_Sbox_ANF //size 2^n
//TEMPORARY
  //IF n<11 THEN
          int* device_CF //size, 2^n x 2^n
    ELSE
          int* device_CF //size, 2^n
RETURN ADmin(S)
//FUNCTION:
int AlgebraicDegreeSboxGPU_in_ANF_ButterflyMin(int* device_Sbox_ANF, int* device_CF, int sizeSbox); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *device_Sbox - MCorF(S) //size 2^n
//OUTPUT
//IF n<11 THEN
         int *device_ADT   ADT(S)  //size 2^n x 2^n
         int *device_CF    TCom(S) //size 2^n x 2^n
    ELSE
         printf("The output data is to big.\n");

//FUNCTION:
void AlgebraicDegreeTableSboxGPU(int* device_Sbox, int* device_CF, int* device_ADT, int sizeSbox); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *host_Sbox - MCorF(S) //size 2^n
//TEMPORARY
//IF n<14 THEN
          int *host_Vect_CF  //size 2^n x 2^n
          unsigned long long int *host_NumIntVecCF    //size (2^n x 2^n)/64
          unsigned long long int *device_NumIntVecCF  //size (2^n x 2^n)/64
//OUTPUT
//IF n<14 THEN
         unsigned long long int *device_NumIntVecANF   ANF(S) //size (2^n x 2^n)/64
//COMMENT Bitwise
//FUNCTION:
void BitwiseMobiusTranSboxGPU(int *host_Sbox, int *host_Vect_CF, unsigned long long int *host_NumIntVecCF,
unsigned long long int *device_NumIntVecCF, unsigned long long int *device_NumIntVecANF, int sizeSbox); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int *host_Sbox - MCorF(S) //size 2^n
//TEMPORARY
//IF n<14 THEN
          int *host_Vect_CF  //size 2^n x 2^n
          unsigned long long int *host_NumIntVecCF    //size (2^n x 2^n)/64
          unsigned long long int *device_NumIntVecCF  //size (2^n x 2^n)/64
          int* host_max_values //(if (2^n x 2^n)/64 < 256), size (2^n x 2^n)/64
          int* device_Vec_max_values  //size (2^n x 2^n)/64
    ELSE
          int *host_Vect_CF  //size 2^n
          unsigned long long int *host_NumIntVecCF    //size 2^n/64
          unsigned long long int *device_NumIntVecCF  //size 2^n/64
```

```
          int* device_Vec_max_values  //size 2^n/64
//OUTPUT
//IF n<14 THEN
         unsigned long long int *device_NumIntVecANF   ANF(S) //size (2^n x 2^n)/64
    ELSE
         unsigned long long int *device_NumIntVecANF  //size 2^n/64
RETURN ADmax(S)
//COMMENT Bitwise
//FUNCTION:
int BitwiseAlgebraicDegreeSboxGPU_ButterflyMax(int* host_Sbox, int* host_Vect_CF, int* host_max_values,
unsigned long long int* host_NumIntVecCF, unsigned long long int* device_NumIntVecCF, unsigned long long int*
device_NumIntVecANF, int* device_Vec_max_values, int sizeSbox); //v0.2
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* device_Sbox - MCorF(S) //size 2^n
//TEMPORARY
//IF n<14 THEN
          int* device_CF  //size 2^n x 2^n
          unsigned long long int *device_NumIntVecCF  // size (2^n x 2^n)/64
          int* device_Vec_max_values  //size (2^n x 2^n)/64
    ELSE
          int *host_Vect_CF // 2^n
          unsigned long long int *device_NumIntVecCF  // size 2^n/64
          int* device_Vec_max_values  // size 2^n/64
//OUTPUT
//IF n<14 THEN
         unsigned long long int *device_NumIntVecANF   ANF(S) //size (2^n x 2^n)/64
    ELSE
         unsigned long long int *device_NumIntVecANF  // size 2^n/64
RETURN ADmax(S)
//COMMENT Bitwise
//FUNCTION:
int BitwiseAlgebraicDegreeSboxGPU_ButterflyMax_v03(unsigned long long int* device_NumIntVecCF, unsigned long
long int* device_NumIntVecANF, int* device_Vec_max_values, int* device_Sbox, int* device_CF, int sizeSbox);//v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* device_Sbox_ANF //size 2^n
//TEMPORARY
  //IF n<14 THEN
          int* device_CF  //size 2^n x 2^n
          unsigned long long int* device_NumIntVecCF  //size (2^n x 2^n)/64
          int* device_Vec_max_values  //size (2^n)/64
    ELSE
          int* device_CF  //size 2^n
          unsigned long long int* device_NumIntVecCF  //size (2^n)/64
          int* device_Vec_max_values  //size (2^n)/64
RETURN ADmax(S)
//FUNCTION:
int BitwiseAlgebraicDegreeSboxGPU_ANF_in_ButterflyMax_v03(unsigned long long int* device_NumIntVecCF,
int* device_Vec_max_values, int* device_Sbox_ANF, int* device_CF, int sizeSbox); //v0.3
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* device_Sbox_ANF  //size 2^n
//TEMPORARY
  //IF n<14 THEN
          int* device_CF  //size 2^n x 2^n
          unsigned long long int* device_NumIntVecCF  //size (2^n x 2^n)/64
          int* device_Vec_max_values  //size (2^n)/64
    ELSE
          int* device_CF  //size 2^n
          unsigned long long int* device_NumIntVecCF  //size (2^n)/64
          int* device_Vec_max_values  //size (2^n)/64
RETURN ADmin(S)
//COMMENT Bitwise
//FUNCTION:
int BitwiseAlgebraicDegreeSboxGPU_ANF_in_ButterflyMin_v03(unsigned long long int* device_NumIntVecCF, int*
```

```
device_Vec_max_values, int* device_Sbox_ANF, int* device_CF, int sizeSbox); //v0.3
=============================================================================================================
```

3.4 Function support operations, reduction, grid configuration

In some cases, a maximum or minimum value in an array is required depending on the properties studied. The general parallel algorithm suitable for a task of this type is known as parallel reduction. It is very important to note that we need to find an absolute maximum or minimum value. The algorithm for parallel reduction is also implemented, which returns a sum from the input array. This is the reason for our modified application of parallel reduction. This algorithm is well known. In addition, the butterfly algorithm for finding the absolute maximum and minimum value of the input array is implemented here.

```
REDUCTION FUNCTION
==================
==================
int* d_idata input data array, sum of all values from input data array SUM,
maximum value from the input data array MAX, minimal value from input data array MIN.

=============================================================================================================
//INPUT  int size - 2^n
         int* d_idata  //size 2^n
RETURN SUM
//FUNCTION:
int runReduction(int size, int* d_idata); //v0.3
=============================================================================================================
//INPUT  int size - 2^n
         int* d_idata  //size 2^n
RETURN MAX
//FUNCTION:
int runReductionMax(int size, int* d_idata); //0.1
=============================================================================================================
//INPUT  int size - 2^n
         int* d_idata  //size 2^n
RETURN MIN
//FUNCTION:
int runReductionMin(int size, int* d_idata); //v0.2
=============================================================================================================

MIN-MAX BUTTERFLY FUNCTION
==========================
==========================
Rearrange input data array, int* device_data.

=============================================================================================================
//INPUT  int size - 2^n
         int* device_data  //size 2^n
//OUTPUT
         int* device_data  //size 2^n
            device_data[0] - MAX value
            device_data[(2^n)-1] - MIN value
//COMMENT min-max butterfly
//FUNCTION:
int Butterfly_max_kernel(int sizeSbox, int* device_data); //v0.2
=============================================================================================================
```

The optimal grid configuration of the parallel function is determined by the available hardware resources. Functions in the library automatically generate the required grid.

```
GRID configuration FUNCTION
===========================
===========================

=============================================================================================================
//INPUT
         int size - 2^n or 2^n x 2^n
```

```
         int NumInt - 2^n/64
         int NumOfBits - 64
//COMMENT initialise global variables
//FUNCTIONS:
inline void setgrid(int size); //v0.1
inline void setgridBitwise(int size); //v0.2
inline void setgrid_BinVecToDec(int size, int NumInt); //v0.3
inline void setgrid_fmt_ad(int sizeSbox, int NumInt, int NumOfBits); //v0.3
=============================================================================================================
```

3.5 Host basic functions

Apart of the BoolSPLG functions the library contains C/C++ functions which are executed consecutive on CPU. These functions can be used for computing the listed cryptographic parameters.

```
HOST BOOLEAN FUNCTION
=====================
=====================
Boolean function f with True Table TT(f), Boolean function f with Polarity True Table PTT(f),
Walsh Spectrum WS(f), Linearity Lin(f), Algebraic Normal Form ANF(f), Algebraic Degree AD(f),
Algebraic Degrees of the monomials in the Boolean functions ADT(f),
Autocorrelation Spectrum AS(f), Autocorrelation AC(f).

=============================================================================================================
//INPUT  int size - 2^n
         int *BoolSbox - PTT(f) //size 2^n
//OUTPUT
         int *walshvec   WS(f) //size 2^n
//FUNCTION:
void FastWalshTrans(int size, int* BoolSbox, int* walshvec);
=============================================================================================================
//INPUT  int size - 2^n
         int *walshvec   //size 2^n
//OUTPUT
         int *walshvec   //size 2^n
//COMMENT inverse walsh transformation
//FUNCTION:
void FastWalshTransInv(int size, int* walshvec);
=============================================================================================================
//INPUT  int size - 2^n
         int* TT - TT(f) or ANF(f) //size 2^n
//OUTPUT
         int* ANF  ANF(f) or TT(f) //size 2^n
//FUNCTION:
void FastMobiushTrans(int size, int* TT, int* ANF);
=============================================================================================================
//INPUT  int size - 2^n
         int* ANF_CPU - ANF(F) //size 2^n
RETURN AD(f)
//FUNCTION:
int FindMaxDeg(int size, int* ANF_CPU);
=============================================================================================================
//INPUT  int NumInt - 2^n/64
         unsigned long long int* NumIntVec - TT(f) //size 2^n/64
//OUTPUT
         unsigned long long int* NumIntVecANF  ANF(S) //size 2^n/64
//COMMENT Bitwise
//FUNCTION:
void CPU_FMT_bitwise(unsigned long long int* NumIntVec, unsigned long long int* NumIntVecANF, int NumInt);
=============================================================================================================
//INPUT  int NumOfBits - 64
         int NumInt - 2^n/64
         unsigned long long int* NumIntVec  ANF(f) //size 2^n/64
RETURN ADmax(S)
//COMMENT Bitwise
```

```
//FUNCTION:
int DecVecToBin_maxDeg(int NumOfBits, unsigned long long int* NumIntVec, int NumInt);
=============================================================================================================
//INPUT  int nvals - 2^n
         int* rf  AS(f) //size 2^n
RETURN AC(f)
//FUNCTION:
int reduceCPU_AC(int nvals, int* rf);
=============================================================================================================

HOST VECTOR BOOLEAN FUNCTION
============================
============================
Vector boolean function S with matrix of coordinate function MCorF(S)
(integer array of size 2^n - each column is a binary representation of the corresponding integer),
Component function Com(S), Table of Component functions TCom(S), Algebraic Normal Form of S-box ANF(S).

=============================================================================================================
//INPUT  int sizeSbox - size 2^n
         int binary - integer n
         int* sbox - S-box integers //size 2^n
//TEMPORARY
         int* binary_num  //size n
//OUTPUT
         int** STT  MCorF(S) //size 2^n x n
//COMMENT use 2D array
//FUNCTION:
void SetSTT(int* sbox, int** STT, int* binary_num, int sizeSbox, int binary);
=============================================================================================================
//INPUT  int sizeSbox - size 2^n
         int binary - integer n
         int* sbox - S-box integers //size 2^n
//TEMPORARY
         int* binary_num  //size n
//OUTPUT
         int** SboxElemet_ANF  ANF(S) //size 2^n x n
//COMMENT use 2D array
//FUNCTION:
void SetS_ANF(int* sbox, int* SboxElemet_ANF, int sizeSbox, int binary);
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* Sbox - MCorF(S) //size 2^n
         int dx - number of MCorF(S) row
//OUTPUT
RETURN DU(f)
//COMMENT return DU(S) of row
//FUNCTION:
int DDT_vector(int sizeSbox, int* sbox, int dx);
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* ANF_CPU  ANF(Com(S)) //size 2^n
//OUTPUT
RETURN ADmax(Com(S))
//COMMENT return ADmax of component functions
//FUNCTION:
int AlgDegMax(int sizeSbox, int* ANF_CPU);
=============================================================================================================
//INPUT  int sizeSbox - 2^n
         int* ANF_CPU  ANF(Com(S)) //size 2^n
//OUTPUT
RETURN ADmin(Com(S))
//COMMENT return ADmin of component functions
//FUNCTION:
int AlgDegMin(int sizeSbox, int* ANF_CPU);
=============================================================================================================
```

```
HOST SUPPORT FUNCTION
============================
============================

=============================================================================================================
//INPUT  int base (integer value)
         int exp  (integer value)
//OUTPUT
RETURN base^exp
//FUNCTION:
int ipow(int base, int exp);
=============================================================================================================
//INPUT  int size - 2^n
         int* vec - //array, size 2^n
//OUTPUT
         vec^2 //size 2^n
//FUNCTION:
void fun_pow2(int size, int* vec);
=============================================================================================================
//INPUT  int nvals - 2^n
         int* vals  //array, size 2^n
//OUTPUT
RETURN max value
//FUNCTION:
int reduceCPU_max(int* vals, int nvals);
=============================================================================================================
//INPUT  int size - 2^n
         int* InputVector    //size 2^n
         string NameSpectrum //string message
//OUTPUT
         int* HistogramVector //Histogram array, size 2^n
//FUNCTION:
void HistogramSpectrum(int size, int* HistohramVector, int* InputVector, string NameSpectrum);
=============================================================================================================
//COMMENT Allocate 2D Dynamic Array
//FUNCTION:
template <typename T>
T** AllocateDynamicArray(int nRows, int nCols); //Allocate 2D Dynamic Array
=============================================================================================================
//COMMENT Free 2D Dynamic Array
//FUNCTION:
template <typename T>
void FreeDynamicArray(T** dArray); //Delete 2D Dynamic Array
=============================================================================================================
//INPUT  int n - size 2^n
//OUTPUT
         int* vector_TT   TT(f)  //size 2^n
         int* vector_PTT  PTT(f) //size 2^n
//COMMENT generate random arrays
//FUNCTION:
void Fill_dp_vector(int n, int* vector_TT, int* vector_PTT);
=============================================================================================================
//INPUT  int binary - n, number of bits
         int number - integer
//OUTPUT
         int* binary_num
//COMMENT integer binary conversion
//FUNCTION:
void binary1(int number, int* binary_num, int binary);
=============================================================================================================
//INPUT  int size - 2^n or 2^n x 2^n
         int* vector1  //size 2^n or 2^n x 2^n
         int* vector2  //size 2^n or 2^n x 2^n
//OUTPUT
//IF true
         printf("\nCheck: True\n");
```

```
ELSE
        printf("\nCheck: False\n");
//COMMENT validate input arrays
//FUNCTION:
void check_rez(int size, int* vector1, int* vector2);
=============================================================================================================
//INPUT  int size - 2^n/64
        unsigned long long int* vector1  //size 2^n/64
        unsigned long long int* vector2  //size 2^n/64
//OUTPUT
//IF true
        printf("\nCheck: True\n");
ELSE
        printf("\nCheck: False\n");
//COMMENT validate input arrays
//FUNCTION:
void check_rez_int(int size, unsigned long long int* vector1, unsigned long long int* vector2);
=============================================================================================================
//INPUT  int size  0 - 2^n
        int* result  //size 2^n
//OUTPUT
       for (0 to size)
              printf("%d, ", result[j]);
//COMMENT print function
//FUNCTION:
void Print_Result(int size, int* result);
=============================================================================================================
//INPUT  int size - 2^n
        T* vector1 - short int or int  //size 2^n
        int* vector2  //size 2^n
//OUTPUT
//IF true
        printf("\nCheck: True\n");
ELSE
        printf("\nCheck: False\n");
//FUNCTION:
template <class T>
void check_rez03(int size, T* vector1, int* vector2);
=============================================================================================================
```

4 Program examples

BoolSPLG distribution directory contains "Examples" directory with examples. All its subdirectories contain several header files and a base file with a "*.cu*" extension. The base file calls the main function. There are additional header files in the corresponding directory that contain CPU functions that are used for comparison and checking the obtained results. The example directory "ExampleSboxBoolSPLG" apart the main and additional header files contains "sbox" file with sample S-boxes.

There is also separate directory "help and additional header files". It contains all additional header files and files with sample S-boxes of different sizes.

4.1 Example of a program code for computing the linearity of a Boolean function

In this subsection a code example (directory "Examples/CodeExample", file *kernel.cu*) for using the BoolSPLG library is presented. The code below shows how to use CPU and GPU functions to compute the linearity $Lin(f)$ of a Boolean function f.

```
//Include Standard library
#include <stdio.h>
#include <iostream>
#include <chrono>

//CUDA runtime.
#include "cuda_runtime.h"
```

```
//Main Library header file
#include <BoolSPLG/BoolSPLG_v03.cuh>

using namespace std;
////////////////////////////////////////////////////////////////////////////////////////////////
//@@ Example using of GPU function for computing Linearity of Boolean function
////////////////////////////////////////////////////////////////////////////////////////////////
int main()
{
        cout << "=========================================================";
        printf("\nExample using of GPU function for computing Linearity of Boolean function BoolSPLG Lin(f).\n");
        cout << "=========================================================";
        ////////////////////////////////////////////////////////////////////////////////////////////
        //@@BoolSPLG Properties Library, Function to check if GPU fulfill BoolSPLG CUDA-capable requires
        BoolSPLGMinimalRequires();
        ////////////////////////////////////////////////////////////////////////////////////////////
        cout << "=========================================================\n";
        printf("Example with random generate Boolean function. Length according to the input exponent.\n");
        //@@Set size of Boolean vector
        cout << "\nInput 'exponent' for power function (base is 2). \n";
        cout << " ==> The input exponent can be between 6 - 20.\n";
        cout << "\nInput exponent:";
        int size, exponent;
        cin >> exponent;
        size = (int)(pow(2, exponent));
        printf("The size length is: 2^%d=:%d\n", exponent, size);
        ////////////////////////////////////////////////////////////////////////////////////////////
        //Declarations, host_Vect_CPU - vector for CPU computation, host_Vect_GPU -vector for GPU computation

        //Allocate memory block. Allocates a block of size bytes of memory
        int *host_Vect_PTT = (int*)malloc(sizeof(int) * size);
        int *walshvec_cpu = (int*)malloc(sizeof(int) * size);
        int *host_Vect_rez = (int*)malloc(sizeof(int) * size);
        ////////////////////////////////////////////////////////////////////////////////////////////
        //Function: Fill random input vector for computation
        Fill_dp_vector(size, host_Vect_PTT, host_Vect_PTT);
        ////////////////////////////////////////////////////////////////////////////////////////////

        ////////////////////////////////////////////////////////////////////////////////////////////
        //@@Start measuring time CPU
        auto startFWTCPU = chrono::steady_clock::now();
        ////////////////////////////////////////////////////////////////////////////////////////////
        //Function: Fast Walsh Transformation function CPU (W_f(f))
        FastWalshTrans(size, host_Vect_PTT, walshvec_cpu);
        int Lin_cpu = reduceCPU_max(walshvec_cpu, size);
        ////////////////////////////////////////////////////////////////////////////////////////////
        //@@ Stop measuring time and calculate the elapsed time
        auto endFWTCPU = chrono::steady_clock::now();
        cout << "=========================================================";
        cout << "\nCPU Elapsed time in milliseconds (CPU - Lin(f)): ";
        cout << chrono::duration_cast<chrono::milliseconds>(endFWTCPU - startFWTCPU).count() << " ms" << endl;
        cout << "CPU Elapsed time in nanoseconds (CPU - Lin(f)): ";
        cout << chrono::duration_cast<chrono::nanoseconds>(endFWTCPU - startFWTCPU).count() << " ns" << endl;
        ////////////////////////////////////////////////////////////////////////////////////////////
        //@@Set Device size array
        int sizeBoolean = sizeof(int) * size;
        //Declaration device vectors
        int* device_Vect, * device_Vect_rez;

        ////////////////////////////////////////////////////////////////////////////////////////////
        //Set and Call Boolean Fast Walsh Transform
        ////////////////////////////////////////////////////////////////////////////////////////////
        //@@ Allocate GPU memory here

        //check CUDA component status
        cudaError_t cudaStatus;
```

```
//input device vector
cudaStatus = cudaMalloc((void**)&device_Vect, sizeBoolean);
if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        //goto Error;
        exit(EXIT_FAILURE);
}
//output device vector
cudaStatus = cudaMalloc((void**)&device_Vect_rez, sizeBoolean);
if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        //goto Error;
        exit(EXIT_FAILURE);
}
////////////////////////////////////////////////////////////////////////////////////////////////
//@@Start mesure time HostToDevice
cudaEvent_t startHToD_PTT, stopHToD_PTT;
cudaEventCreate(&startHToD_PTT);
cudaEventCreate(&stopHToD_PTT);
cudaEventRecord(startHToD_PTT);
////////////////////////////////////////////////////////////////////////////////////////////////
// Copy input vectors from host memory to GPU buffers.
cudaStatus = cudaMemcpy(device_Vect, host_Vect_PTT, sizeBoolean, cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        exit(EXIT_FAILURE);
}
////////////////////////////////////////////////////////////////////////////////////////////////
//@@Stop mesure time HostToDevice
cudaEventRecord(stopHToD_PTT);
cudaEventSynchronize(stopHToD_PTT);
float elapsedTimeHToD_PTT = 0;
cudaEventElapsedTime(&elapsedTimeHToD_PTT, startHToD_PTT, stopHToD_PTT);
cout << "\n=========================================\n";
printf("\n(GPU HostToDevice) Time taken to copy PTT(f) (int): %3.6f ms \n", elapsedTimeHToD_PTT);
////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
//Call BoolSPLG Library function for FWT(f) calculation
////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
//@@Start mesure time compute FWT GPU
cudaEvent_t startTimeFWT, stopTimeFWT;
cudaEventCreate(&startTimeFWT);
cudaEventCreate(&stopTimeFWT);
cudaEventRecord(startTimeFWT);
////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
//Call BoolSPLG Library, Boolean Fast Walsh Transform: return Walsh Spectra and Lin(f)
////////////////////////////////////////////////////////////////////////////////////////////////
int Lin_gpu = WalshSpecTranBoolGPU_ButterflyMax(device_Vect, device_Vect_rez, size, true);
////////////////////////////////////////////////////////////////////////////////////////////////
//@@Stop mesure time compute FWT GPU
cudaEventRecord(stopTimeFWT);
cudaEventSynchronize(stopTimeFWT);
float elapsedTimeFWT = 0;
cudaEventElapsedTime(&elapsedTimeFWT, startTimeFWT, stopTimeFWT);
cout << "\n=========================================\n";
printf("\n(GPU) Time taken to Compute Lin(S): %3.6f ms \n", elapsedTimeFWT);
////////////////////////////////////////////////////////////////////////////////////////////////
//cudaDeviceSynchronize waits for the kernel to finish, and
//returns any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
```

```
        if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
        exit(EXIT_FAILURE);
        }
        //Copy output vector from GPU buffer to host memory.
        cudaStatus = cudaMemcpy(host_Vect_rez, device_Vect_rez, sizeBoolean, cudaMemcpyDeviceToHost);
        if (cudaStatus != cudaSuccess) {
                fprintf(stderr, "cudaMemcpy failed!");
                exit(EXIT_FAILURE);
        }
        ////////////////////////////////////////////////////////////////////////////////////////////////
        //Compare, Check result
        cout << "\nCheck result FWT(f) -> CPU vs. GPU:";
        check_rez(host_Vect_rez, walshvec_cpu, size);
        ////////////////////////////////////////////////////////////////////////////////////////////////
        cout << "\nLin(f)_cpu:" << Lin_cpu << "\n";
        cout << "Lin(f)_gpu:" << Lin_gpu << "\n";
        cout << "\n=========================================\n";
        ////////////////////////////////////////////////////////////////////////////////////////////////
        printf("\n   --- End Example, Boolean function BoolSPLG. ---\n\n");
        ////////////////////////////////////////////////////////////////////////////////////////////////
        //@Free memory
        cudaFree(device_Vect);
        cudaFree(device_Vect_rez);
        free(host_Vect_PTT);
        free(host_Vect_rez);
        free(walshvec_cpu);

        return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////
```

4.2 Output from code execution

Below is given command prompt output information and result from the code execution. The output refers to calculating the Walsh spectrum and finding the linearity of a randomly generated Boolean function of 18 variables on a NVIDIA GeForce MX150 GPU.

```
=============================================================
Example using of GPU function for computing Linearity of Boolean function BoolSPLG.
=============================================================
Minimal requires compute capability 3.0 or later to run the Library BoolSPLG version 0.3;

Fulfilled minimal requires to run Library BooSPLG version 0.3:
  CUDA Capability Major/Minor version number:    6.1

 Running on...
  Device: NVIDIA GeForce MX150
=============================================================
The example Boolean function will randomly generate with length according to the input exponent.

Input 'exponent' for power function (base is 2).
 ==> The input exponent can be between 6 - 20.

Input exponent:18
The size length is: 2^18=:262144
=============================================================
CPU Elapsed time in milliseconds (CPU - Lin(f)): 3 ms
CPU Elapsed time in nanoseconds (CPU - Lin(f)): 3096200 ns
=============================================================
(GPU HostToDevice) Time taken to copy PTT(f) (int): 0.465920 ms
=============================================================
(GPU) Time taken to Compute Lin(S): 0.887808 ms

Check result FWT(f) -> CPU vs. GPU:
Check: True
```

```
Lin(f)_cpu:4040
Lin(f)_gpu:4040
============================================================
   --- End Example, Boolean function BoolSPLG. ---
```

5 Experimental results

The platforms used for the experiments are described in Table 1. A server (Table 1) with CPU Intel Xeon E5-2640 processor is used to run the experiments and this server contains two graphic cards. The first graphic card NVIDIA TITAN X [10], has 3584 cores running at 1.41 GHz and 480 (GB/sec) memory bandwidth. The second graphic card NVIDIA GeForce GTX TITAN [11], has 2688 cores running at 876 MHz and 288.38 (GB/sec) memory bandwidth. We use CUDA Toolkit 10.2 and development environment MS Visual Studio 2019. Programs are build / executed in Active solution configuration-Release, and Active solution platform-$x64$ mode. The GPU NVIDIA TITAN X is denote as Device 0 and the GPU GeForce GTX TITAN as Device 1.

Table 1: Description of the test platform table

Environment:	Platform	
CPU	Intel Xeon E5-2640, 2.50GHz	
Memory	48 GB DDR3 1333 MHz	
OS	Windows 7, 64-bit	
IDE/Compiler	MSVC 2019	
CUDA SDK	10.2	
GPU Driver	V 471.96	
GPU	Nvidia TITAN X (Pascal)	GeForce GTX TITAN
Notation	Device 0	Device 1
Architecture	Pascal	Kepler
CUDA Cores	3584	2688
Boost Clock	1531 MHz	876 MHz
Memory Speed	10 Gbps	6 Gbps
Global Memory	12 GB GDDR5X	6 GB GDDR5
Memory Bandwidth	480 (GB/sec)	288.38 (GB/sec)

The experiments are performed on a specific predefined input data set of 100 Boolean functions and 100 S-boxes. To measure the execution time, the program is executed under the same conditions and the time is obtained by using timers specially placed on suitable positions. Results in the tables represent the average execution time related with the input data set. This means that we compute a given cryptographic parameter with a predefined input data set of 100 objects and as result we take the average time. However, in case of repetition of the experiment a deviation of $\pm 5\%$ may occur. The input data set have determined structure which consists of integer arrays with 2^n entries ($n = 8, \ldots, 20$).

One of the biggest problems in the CUDA programming model is data transfer. Table 2 gives the time to copy an array specifying the S-box from host to device.

Table 2: (GPU) Time for transfer the S-box from the Host to Device

Size	Device 0 (ms)	Device 1 (ms)
2^8 (256)	0.06348	0.085984
2^9 (512)	0.06564	0.070912
2^{10} (1024)	0.06348	0.087520
2^{11} (2048)	0.06244	0.120224
2^{12} (4096)	0.11776	0.172787
2^{13} (8192)	0.118784	0.119040
2^{14} (16384)	0.151552	0.173984
2^{15} (32768)	0.160768	0.190496
2^{16} (65536)	0.348192	0.481696
2^{17} (131072)	0.478272	0.668800
2^{18} (262144)	0.821384	0.902624
2^{19} (524288)	1.412128	1.545760
2^{20} (1048576)	2.388416	2.627904

The efficiency of the parallel realization is measured with the achieved *speedup*, given by the formula:

$$S_P = \frac{T_0(n)}{T_p(n)}.$$

Here n is the size of the input data (in our case the number of variables of the Boolean function), $T_0(n)$ is the execution time of the sequential algorithm, and $T_p(n)$ is the execution time of the parallel algorithm.

In the next tables we present the obtained experimental results from the cryptographic parameters computation. For calculating the linearity, algebraic degree, autocorrelation of a vectorial Boolean function S, we need all its component functions S_b (not only the coordinate functions). Table 3 shows the time for calculating the S-box component functions.

Table 3: Computing Component functions S_b of the S-box

Size	CPU(ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2^8 (256)	0,17	0,009216	x18.47	0.9683	-
2^9 (512)	1.442	0.012288	x118.19	0.3705	x3.8
2^{10} (1024)	2.975	0.019456	x156.57	1.645	x1.8
2^{11} (2048)	7.052	12.402	/	15	/
2^{12} (4096)	26.871	30.215	/	38.85	/
2^{13} (8192)	118.456	51.885	x2.28	79.55	x1.4
2^{14} (16384)	445.29	139.61	x3.18	125.91	x3.56
2^{15} (32768)	1871.98	243.36	x7.7	269.92	x6.9
2^{16} (65536)	7088.38	502.42	x14.1	544.5	x13
2^{17} (131072)	28685.2	875.96	x32.74	1125	x25.5
2^{18} (262144)	114278	1823.68	x62.68	3810.62	x30
2^{19} (524288)	436793	6480.8	x67.4	12448.54	x35
2^{20} (1048576)	1655729	24285	x68.1	44858.15	x36.9

Tables 4, 5, 6 and 7 present the time required and speedup to calculate linearity $Lin(S)$, autocorrelation $AC(S)$, algebraic degree $deg(S)$ and differential uniformity $\delta(S)$ of S-boxes in the performed experiments.

Tables 8, 9 and 10 present experimental results from the calculation of linearity $Lin(f)$, autocorrelation $AC(f)$ and algebraic degree $deg(f)$ of a Boolean functions.

A Appendix A Device functions

This subsection provides declarations for Boolean functions and S-boxes for GPU devices (function name, return type and parameters). The comment at the end of the function declaration shows the version of the BoolSPLG library in which the function was added.

These functions (kernels) for parallel execution are the building blocks of the basic functions. In most cases, their names and accompanying comments indicate which feature it is. The names of the parameters show their meaning (for those familiar with the topic). The following is a description of the device function declarations.

Table 4: Computing Linearity $Lin(S)$ of S-box

Size	CPU(ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2^8 (256)	0.863	0.223	x3.86	0.14336	x6
2^9 (512)	7.654	0.340	x22.51	0.3576	x21.4
2^{10} (1024)	15.855	0.318	x49.85	1.1442	x13.8
2^{11} (2048)	68.343	62.774	/	72	/
2^{12} (4096)	314.927	149.562	x2.1	186.106	x1.68
2^{13} (8192)	1257.97	287.688	x4.83	347.72	x3.6
2^{14} (16384)	5686.84	622.29	x9.13	759.53	x7.5
2^{15} (32768)	24083.8	1239.54	x19.42	2141.55	x11.2
2^{16} (65536)	99800.3	2446.03	x40.8	6289.8	x15.86
2^{17} (131072)	442678	6165.59	x71.8	22772	x19.43
2^{18} (262144)	1677921	22029.27	x76.1	90089	x18.62
2^{19} (524288)	7562247	90786.66	x83.29	375519	x20.13
2^{20} (1048576)	29638868	418942	x70.7	1618402	x18.3

Table 5: Computing Autocorrelation $AC(S)$ of S-box

Size	CPU(ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2^8 (256)	2.020	0.206	x9.8	0.319	x6.3
2^9 (512)	14.212	0.435	X32.67	0.393	x36.13
2^{10} (1024)	37.940	0.463	X81.94	1.462	x25.95
2^{11} (2048)	185.673	132.284	X1.4	144.06	x1.28
2^{12} (4096)	771.527	209.130	X3.68	253.18	x3
2^{13} (8192)	3142.98	503.77	X6.23	418.54	x7.5
2^{14} (16384)	13555.2	866.02	X15.65	1094.63	x12.3
2^{15} (32768)	57324.5	1931.21	X29.68	3189.85	x18
2^{16} (65536)	238216.4	3813.45	X62.46	9621.8	x24.76
2^{17} (131072)	1060294	9396.38	x112.84	34810.46	x30.45
2^{18} (262144)	3832308	33814.94	x113.33	133629.15	x28.67
2^{19} (524288)	16860299	138108.15	x122.1	566914	x29.7
2^{20} (1048576)	68227870	629515	x108.38	2411828	x28.2

Table 6: Computing Algebraic Degree $deg(S)$ of S-box

Size	CPU(ms)	Device 0 base(ms)	Device 0 bitwise	CPUvsDev0 base	CPUvsDev0 bitwise	Device 1 base(ms)	Device 1 bitwise	CPUvsDev1 base	CPUvsDev1 bitwise
2^8	1.127	0.122	0.115	x9.23	x9.8	0.3193	0.2692	x3.5	x14.1
2^9	10.099	0.201	0.202	x49.9	x49	0.7321	0.3087	x13.6	x32.46
2^{10}	20.396	0.318	0.142	x64.1	x143.6	1.1814	0.7277	x17.2	x28
2^{11}	102.390	76.042	12.447	x1.3	x8.2	78.66	23.71	x1.3	x4.3
2^{12}	369.589	162.987	28.963	x2.2	x12.7	220.58	42.9	x1.6	x8.6
2^{13}	1754.26	307.870	54.385	x5.69	x32.2	366	65.68	x4.8	x26.9
2^{14}	6822.06	720.525	279.5	x9.4	x24.45	822.66	767.35	x8.3	x8.9
2^{15}	28395.3	1410.75	489.42	x20.1	x58	2221	2658.13	x12.78	x10.6
2^{16}	117070.8	2961.87	1593.99	x39.5	x73.5	6400	9964	x18.29	x11.74
2^{17}	492150.2	6421.09	6470	x76.6	x76	22974	26110	x21.42	x18.84
2^{18}	1699950	22824.7	12011.3	x74.48	x141.53	84706	56085	x20	x30.1
2^{19}	7327083	90603.8	32384.5	x80.87	x226.2	356180	122500	x20.5	x59.8
2^{20}	29481099	416376	83943	x70.8	x351.2	1530321	478612	x19.2	x61.59

Table 7: Computing Differential uniformity $\delta(S)$ of S-box

Size	CPU(ms)	Device 0 (ms)	CPU vs. Dev 0	Device 1 (ms)	CPU vs. Dev 1
2^8 (256)	0.282	0.208	/	0.136	x2
2^9 (512)	1.908	0.432	x4.416	0.351	x5.4
2^{10} (1024)	3.591	0.366	x9.811	1.053	x3.4
2^{11} (2048)	14.269	0.705	x20.23	1.313	x10.8
2^{12} (4096)	69.932	1.710	x40.89	2.914	x24
2^{13} (8192)	245.719	5.773	x42.56	11.27	x22
2^{14} (16384)	998.174	21.022	x47.48	37.64	x27
2^{15} (32768)	4059.71	990.88	x4.1	1497.6	x2.7
2^{16} (65536)	18307.9	1924.91	x9.5	4345	x4.2
2^{17} (131072)	86983.9	4206.78	x20.68	14988.47	x5.8
2^{18} (262144)	453136	14319.73	x31.64	55269.61	x8.2
2^{19} (524288)	1925444	61572	x31.27	283692	x6.7
2^{20} (1048576)	9312785	730660	x12.7	1531534	x6

Table 8: Computing Linearity $Lin(f)$ of Boolean function

Size	CPU(ms)	Device 0 v1(ms)	Device 0 v3(ms)	CPU vs. Dev 0
2^{12} (4096)	0.081	0.081	0.102	/
2^{13} (8192)	0.177	0.079	0.169	x2.1
2^{14} (16384)	0.336	0.080	0.105	x4.2
2^{15} (32768)	0.77	0.082	0.111	x9.3
2^{16} (65536)	1.438	0.096	0.092	x15.8
2^{17} (131072)	3.19	0.122	0.147	x26
2^{18} (262144)	6.83	0.150	0.203	x45
2^{19} (524288)	14.74	0.242	0.251	x61
2^{20} (1048576)	30.36ll	0.376	0.489	x82

Table 9: Computing Autocorrelation $AC(f)$ of Boolean function

Size	CPU(ms)	Device 0 v1(ms)	Device 0 v3(ms)	CPU vs. Dev 0
2^{12} (4096)	0.156	0.19	0.241	/
2^{13} (8192)	0.371	0.19	0.21	x2
2^{14} (16384)	0.802	0.16	0.2	x5
2^{15} (32768)	1.775	0.144	0.204	X12.32
2^{16} (65536)	3.42	0.135	0.164	x26
2^{17} (131072)	7.75	0.169	0.238	x45.8
2^{18} (262144)	16.51	0.234	0.238	x70.5
2^{19} (524288)	35.69	0.316	0.361	x112.9
2^{20} (1048576)	73.8	0.525	0.647	x141

Table 10: Computing Algebraic Degree $deg(S)$ of Boolean function

Size	CPU(ms) base(ms)	CPU(ms) bitwise(ms)	Device 0 base(ms)	Device 0 bitwise(ms)	CPU vs Dev 0 base	CPU vs Dev 0 bitwise
2^{12} (4096)	0.084	0.002	0.13	0.09	/	/
2^{13} (8192)	0.204	0.047	0.13	0.09	x1.5	x2
2^{14} (16384)	0.369	0.087	0.138	0.091	x2.8	x4
2^{15} (32768)	0.839	0.17	0.115	0.079	x7.6	x10
2^{16} (65536)	1.617	0.36	0.123	0.089	x13.5	x18
2^{17} (131072)	3.40	0.688	0.140	0.123	x24	x28
2^{18} (262144)	7.06	1.31	0.176	0.112	x41	x64
2^{19} (524288)	15.84	2.74	0.259	0.110	x63	x144
2^{20} (1048576)	32.05	5.57	0.409	0.121	x80.12	x267

```
DEVICE BOOLEAN FUNCTION
=======================
=======================

=============================================================================================================
//GPU Fast Walsh Transform
extern __global__ void fwt_kernel_shfl_xor_SM(int *VectorValue, int *VectorValueRez, int step); //0.1
extern __global__ void fwt_kernel_shfl_xor_SM_MP(int *VectorValue, int fsize, int fsize1); //0.1

extern __global__ void fwt_kernel_shfl_xor_SM_LAT(int* VectorValue, int* VectorValueRez, int step); //v0.3

template <class T>
__global__ void fwt_kernel_shfl_xor_SM_v03(int* VectorValue, T* VectorValueRez, int step); //v0.3
template <class T>
__global__ void fwt_kernel_shfl_xor_SM_MP_v03(T* VectorValue, int fsize, int fsize1); //v0.3

//GPU Fast Mobius Transform
extern __global__ void fmt_kernel_shfl_xor_SM(int * VectorValue, int * VectorRez, int sizefor); //0.1
extern __global__ void fmt_kernel_shfl_xor_SM_MP(int * VectorValue, int fsize, int fsize1); //0.1

//GPU Bitwise Fast Mobius Transform
extern __global__ void fmt_bitwise_kernel_shfl_xor_SM
(unsigned long long int *vect, unsigned long long int *vect_out, int sizefor, int sizefor1); //v0.2

extern __global__ void fmt_bitwise_kernel_shfl_xor_SM_MP
(unsigned long long int * VectorValue, int fsize, int fsize1); //v0.2

template <class T>
__global__ void fmt_kernel_shfl_xor_SM_v03(int* VectorValue, T* VectorRez, int sizefor); //v0.3
template <class T>
__global__ void fmt_kernel_shfl_xor_SM_MP_v03(T* VectorValue, int fsize, int fsize1); //v0.3

//GPU Algebraic Degree
extern __global__ void kernel_AD(int *Vec); //0.1

extern __global__ void kernel_bitwise_AD
(unsigned long long int *NumIntVec, int *Vec_max_values, int NumOfBits); //v0.2

//GPU Inverse Fast Walsh Transform
extern __global__ void ifmt_kernel_shfl_xor_SM(int * VectorValue, int * VectorValueRez, int step); //0.1
extern __global__ void ifmt_kernel_shfl_xor_SM_MP(int * VectorValue, int fsize, int fsize1); //0.1
template <class T>
__global__ void ifmt_kernel_shfl_xor_SM_v03(T* VectorValue, T* VectorValueRez, int step); //v0.3
template <class T>
__global__ void ifmt_kernel_shfl_xor_SM_MP_v03(T* VectorValue, int fsize, int fsize1); //v0.3

//GPU Min-Max Butterfly
extern __global__ void Butterfly_max_min_kernel_shfl_xor_SM
(int *VectorValue, int *VectorValueRez, int step); //v0.2

extern __global__ void Butterfly_max_min_kernel_shfl_xor_SM_MP
(int * VectorValue, int fsize, int fsize1); //v0.2

template <class T>
__global__ void Butterfly_max_min_kernel_shfl_xor_SM_v03(T* VectorValue, T* VectorValueRez, int step); //v0.3
template <class T>
__global__ void Butterfly_max_min_kernel_shfl_xor_SM_MP_v03(T* VectorValue, int fsize, int fsize1); //v0.3
=============================================================================================================

DEVICE VECTOR BOOLEAN FUNCTION
==============================
==============================

=============================================================================================================
//Inverse Fast Walsh Transforms for S-box
```

```
extern __global__ void ifmt_kernel_shfl_xor_SM_Sbox(int * VectorValue, int * VectorValueRez, int step); //0.1

//GPU Bitwise Fast M\"{o}bius Transform
extern __global__ void fmt_bitwise_kernel_shfl_xor_SM_Sbox
(unsigned long long int *vect, unsigned long long int *vect_out, int sizefor, int sizefor1); //v0.2

//GPU Algebraic Degree
extern __global__ void kernel_AD_Sbox(int *Vec); //0.1

extern __global__ void kernel_bitwise_AD_Sbox
(unsigned long long int *NumIntVecANF, int *max_values, int NumOfBits); //v0.2

//GPU Difference Distribution Table
extern __global__ void DDTFnAll_kernel(int *Sbox_in, int *DDT_out, int n); //0.1
extern __global__ void DDTFnVec_kernel(int *Sbox_in, int *DDT_out, int row); //0.1

//GPU S-box Component functions
extern __global__ void ComponentFnAll_kernel(int *Sbox_in, int *CF_out, int n); //0.1
extern __global__ void ComponentFnVec_kernel(int *Sbox_in, int *CF_out, int row); //0.1
```

A Boolean function in n variables is represented by its Truth Table, Polarity Truth Table or Algebraic Normal Form (as a vector of the coefficient), which is an integer vector with 2^n entries. A vectorial Boolean function (S-box) is also represented by a vector with 2^n integer coordinates, but in this case the binary representation of the i-th coordinate corresponds to the i-th column in the matrix G_S that generates the S-box.

Acknowledgments

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

The development of the library is supported by the Bulgarian National Science Fund under Contract No KP-06-N32/2-2019 and Contract No KP-06-N62/2/13.12.2022.

References

[1] Beauchamp, K. G. (1985). Applications of walsh and related functions. With an introduction to sequence theory: Academic Press, Inc., Microelectronics and Signal Processing Series, 24–28 Oval Road, London NW1 7DX and Orlando, FL 32887, 1984, xvi+ 308 pp., ISBN 0-12-084180-0.

[2] Bikov, D., Bouyukliev, I.: Parallel Fast Walsh Transform Algorithm and its implementation with CUDA on GPUs. Cybernetics and Information Technologies, Cybernetics and Information Technologies 18 (5), 21–43 (2018)

[3] Bikov D., Pashinska M., Parallel Programming Strategies for Computing Walsh Spectra of Boolean Functions, In International Conference on ICT Innovations, pp. 138-152. Springer, Cham, 2020.

[4] BSD Licenses CMake, Availaible on:
`https://gitlab.kitware.com/cmake/cmake/-/tree/master/Licenses`

[5] Canteaut, A., Duval, S., Leurent, G., Naya-Plasencia, M., Perrin, L., Pornin, T., & Schrottenloher, A. (2019). Saturnin: a suite of lightweight symmetric algorithms for post-quantum security.

[6] Carlet, C., Crama, Y., & Hammer, P. L. (2010). Boolean Functions for Cryptography and Error-Correcting Codes.

[7] CMake official web site, Availaible on:
`https://cmake.org/`

[8] Kelly, M., Kaminsky, A., Kurdziel, M., Łukowiak, M., & Radziszowski, S. (2015, October). Customizable sponge-based authenticated encryption using 16-bit s-boxes. In MILCOM 2015-2015 IEEE Military Communications Conference (pp. 43-48). IEEE.

[9] Matsui, M. (1997, January). New block encryption algorithm MISTY. In International Workshop on Fast Software Encryption (pp. 54-68). Springer, Berlin, Heidelberg.

[10] NVIDIA TITAN X Specification, Availaible on:
https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/ Last accessed 01 March 2022

[11] NVIDIA GeForce GTX TITAN Specification, Availaible on:
https://www.techpowerup.com/gpu-specs/geforce-gtx-titan.c1996 Last accessed 01 March 2022

[12] Sanders, J., & Kandrot, E. (2010). CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.

[13] Shetty, V. S., Anusha, R., MJ, D. K., & Hegde, P. (2020, February). A survey on performance analysis of block cipher algorithms. In 2020 International Conference on Inventive Computation Technologies (ICICT) (pp. 167-174). IEEE.