**About BoolSPL: A library with parallel algorithms for Boolean functions and S-boxes for GPU**

***Current release: v0.2***

***What is BoolSPL?***

BoolSPL (Boolean S-box parallel library for GPU) provides, reusable software components for every layer of the CUDA programming model [4]. BoolSPLG is a library consisting procedures for analysis and compute cryptographic properties of Boolean and Vector Boolean function (S-box). Our procedures have function for auto grid conﬁguration. Most of the functions are designed to compute the data in registers because they oﬀer the highest bandwidth.

**Overview of BoolSPLG Basic Procedures**

The proposed library implement algorithm as composition of basic function into one parameterized kernel, without care about details of implementation. The building function can be classiﬁed into computation (Butterﬂy (FWT, IFWT, FMT, bitwise FMT, min-max), DDT, AlgebraicDegree, ComponentFunction, PowerInt), reordering operations (Copy, MemoryPatern) and support operation reduction (min, max).

Figure 1 presents a scheme with the classiﬁcation of the functions used to build procedures for computing the cryptographic properties of Boolean and Vector Boolean function. The solid line indicates a dependency while the dashed line represents an optional component.

A screenshot of a cell phone

Description automatically generated

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

Our library contains the following butterﬂy algorithms: binary Fast Walsh Transforms (FWT), binary Inverse Fast Walsh Transforms (IFWT), binary Fast Mobius Transforms (FMT), Bitwise binary Fast Mobius Transform (bitwise FMT) and butterfly Min-Max. There are additional algorithms and function for computing DDT, algebraic normal form, component function and auxiliary function reduction for maintaining necessary operations ﬁgure 1.

File BoolSPLG\_v02 contain all declaration of host and device functions and procedures. On the end of declaration there is comment for the library version of the function/procedure. Included procedure in BoolSPLG compute next cryptographic properties: Wf(f) (Walsh spectra of Boolean function), Lin(f) (Linearity of Boolean function), LAT(S) (Linear Approximation Table of S-box), Lin(S) (Linearity of S-box), rf(f) (Autocorrelation Spectrum of Boolean function), AC(f) (Autocorrelation of Boolean function), ACT(S) (Autocorrelation spectrum of S-box), AC(S) (Autocorrelation of S-box), ANF(f) (Algebraic Normal Form of Boolean function), ANF(S) (Algebraic Normal Form of S-box), Deg(f) (Algebraic Degree of Boolean function), Deg(S) (Algebraic Degree of S-box), DDT(S) (Difference Distribution Table), δ (Diﬀerential uniformity) and Sb (Component function of S-box) [1].

**Device functions**

//Declaration for Boolean GPU device functions

//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

//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

//GPU compute 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

//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

extern \_\_global\_\_ void ifmt\_kernel\_shfl\_xor\_SM\_Sbox(int \* VectorValue, int \* VectorValueRez, int step); //0.1

//Declaration for S-box GPU device functions

//GPU Bitwise Fast Mobius 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 compute 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

**Blocks passing data**

Hardware limitation resources (memory, number of thread per block) inﬂuence the design of algorithms. If dimension of input array bigger from 210 entries, in some point it is need rearranges of data between memory from diﬀerent blocks. The memory pattern (use in [2, 3]) rearranges the shared memory in such a way that the memory elements from diﬀerent blocks (intermediate results) are set in order to perform butterﬂy algorithms from the beginning. Rearranges of shared memory data between diﬀerent block is made by pointers, without worrying about the number of blocks cooperating.

**S-box, component function**

In order to study the cryptographic properties of a vectorial Boolean function we need to consider all non-zero linear combinations of the coordinates of the vectorial Boolean function [1]. We implement two similar algorithms for computing component functions. The ﬁrst one (ComponentFnAll kernel) compute all component function at once (n ≤ 10) and the second one (ComponentFnVec\_kernel) compute component function one after another (n > 10). This separation is caused by the hardware resource limitation (memory, number of thread per block).

First device function (ComponentFnAll kernel), use two array and integer block size. Input array contain vector Boolean function. Every one of the element represent integer which binary representation is column from vector Boolean matrix representation. Output 2n × 2n array contain all component function. Number of component function, number of blocks, size of component function and the block size are equal. Threads from one block computed one component function. Every block, have copy from the vector Boolean function, in fact values from the vector Boolean function is write in threads local register value. The output array CF\_out contain sequence of component function and it is 2n × 2n dimensional Figure 2.

A screenshot of a cell phone

Description generated with very high confidence

Figure 2. S-box, array contain all component function

**Function support operations, reduction**

In some of the algorithms there are need maximum or minimum value depending from the examined properties. General parallel algorithm suitable for problem of this type are known as reduction. Very important to notice that we need to ﬁnd absolute maximum or minimum (minimum will be implemented in next library release) value. This is the reason for our modiﬁed reduction implementation. But we will not give detail of implementation because algorithm is well known.

//Declaration for Max - Min Reduction function

int runReductionMax(int size, int \*d\_idata); //0.1

int runReductionMin(int size, int \*d\_idata); //v0.2

//Declaration for Butterfly max function

int Butterfly\_max\_kernel(int sizeSbox, int \*device\_data); //v0.2

**Grid conﬁguration**

One of the basic requirements for to obtain optimal performance is assignment of maximum parallelism that we control with conﬁguration of the grid. Optimal setting and conﬁguration of the parallel function’s grid is limited by the available hardware resources. Our procedures automatically adjust the grid. Resource that impact performing are numbers of registers per thread, shared memory per block, number of running block per SM and number of threads per block.

//Function: Set GRID

inline void setgrid(int size); //0.1

inline void setgridBitwise(int size); //0.2

**BoolSPLG procedures**

The procedures are combination of functions that unite sequence of diﬀerent algorithms, and at the same time they maintain the correct parallel distribution of work. Introduced device function give possibility for design diﬀerent compact algorithm. This main parallel function beside combining function, perform grid conﬁguration.

//Declaration for Boolean procedures

// Wf(f) Walsh spectra, return Lin(f) Linearity of Boolean function

int WalshSpecTranBoolGPU(int \*device\_Vect, int \*device\_Vect\_rez, int size, bool returnMaxReduction); //0.1 BoolFWT\_compute

// ANF(f) Algebraic Normal Form of Boolean function

void MobiusTranBoolGPU(int \*device\_Vect, int \*device\_Vect\_rez, int size); //0.1 BoolFMT\_compute

//return - deg(f) Algebraic Degree of Boolean function

int AlgebraicDegreeBoolGPU(int \*device\_Vect, int \*device\_Vect\_rez, int size); //0.1 BoolAD\_compute

// rf(f) Autocorrelation Spectrum, return AC(f) Autocorrelation of Boolean function

int AutocorrelationTranBoolGPU(int \*device\_Vect, int \*device\_Vect\_rez, int size, bool returnMaxReduction); //0.1BoolAC\_compute

// Wf(f) Walsh spectra, return Lin(f) Linearity of Boolean function (Butterfly max)

int WalshSpecTranBoolGPU\_ButterflyMax(int \*device\_Vect, int \*device\_Vect\_rez, int size, bool returnMaxReduction); //v0.2 BoolFWT\_compute

//return - deg(f) Algebraic Degree of Boolean function (Butterfly max)

int AlgebraicDegreeBoolGPU\_ButterflyMax(int \*device\_Vect, int \*device\_Vect\_rez, int size); //v0.2 BoolAD\_compute

// rf(f) Autocorrelation Spectrum, return AC(f) Autocorrelation of Boolean function (Butterfly max)

int AutocorrelationTranBoolGPU\_ButterflyMax(int \*device\_Vect, int \*device\_Vect\_rez, int size, bool returnMaxReduction); //v0.2 BoolAC\_compute

//Declaration for Bitwise Boolean procedures

// ANF(f) Algebraic Normal Form of Boolean function

void BitwiseMobiusTranBoolGPU(unsigned long long int \*device\_Vect, unsigned long long int \*device\_Vect\_rez, int size); //v0.2 Bitwise BoolFMT\_compute

//return - deg(f) Algebraic Degree of Boolean function (Butterfly max)

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

//Declaration for S-box procedures

// LAT(S) Linear Approximation Table, return Lin(S) Linearity of S-box

int WalshSpecTranSboxGPU(int \*device\_Sbox, int \*device\_CF, int \*device\_LAT, int sizeSbox); //0.1

// ANF(S) Algebraic Normal Form of S-box and return deg(S) Algebraic Degree of S-box

int MobiusTranSboxADGPU(int \*device\_Sbox, int \*device\_CF, int \*device\_ANF, int sizeSbox); //0.1

// ACT(S) Autocorrelation Table, return AC(S) Autocorrelation of S-box

int AutocorrelationTranSboxGPU(int \*device\_Sbox, int \*device\_CF, int \*device\_ACT, int sizeSbox); //0.1

// DDT(S) Difference Distribution Table, return δ Differential uniformity of S-box

int DDTSboxGPU(int \*device\_Sbox, int \*device\_DDT, int sizeSbox); //0.1

// LAT(S) Linear Approximation Table, return Lin(S) Linearity of S-box (Butterfly max)

int WalshSpecTranSboxGPU\_ButterflyMax(int \*device\_Sbox, int \*device\_CF, int \*device\_LAT, int sizeSbox, bool returnMax); //v0.2

// ANF(S) Algebraic Normal Form of S-box

void MobiusTranSboxGPU(int \*device\_Sbox, int \*device\_CF, int \*device\_ANF, int sizeSbox); //v0.2

// return deg(S) Algebraic Degree of S-box (Butterfly max)

int AlgebraicDegreeSboxGPU\_ButterflyMax(int \*device\_Sbox, int \*device\_CF, int \*device\_ANF, int sizeSbox); //v0.2

// ACT(S) Autocorrelation Table, return AC(S) Autocorrelation of S-box (Butterfly max)

int AutocorrelationTranSboxGPU\_ButterflyMax(int \*device\_Sbox, int \*device\_CF, int \*device\_ACT, int sizeSbox, bool returnMax); //v0.2

// DDT(S) Difference Distribution Table, return δ Differential uniformity of S-box (Butterfly max)

int DDTSboxGPU\_ButterflyMax(int \*device\_Sbox, int \*device\_DDT, int sizeSbox, bool returnMax); //v0.2

//Declaration for Bitwise S-box procedures

// ANF(f) Algebraic Normal Form of S-box

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

//return - deg(f) Algebraic Degree of S-box (Butterfly max)

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

**How do I get started using BoolSPL?**

BoolSPL is implemented as a C++ header library. There is no need to “build” BoolSPL separately. To use BoolSPL primitives in your code, simply:

1. Download and unzip the latest BoolSPL distribution from the Downloads section and extract the contents of the zip file to a directory. You need to install (copy) only “BoolSPL” directory from the main BoolSPL-vx.x directory. We suggest installing BoolSPL to the CUDA include directory, which is usually:
   * /usr/local/cuda/include/ on a Linux and Mac OSX;
   * C:\CUDA\include\ on a Windows system.

Example: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\include\;

If you are unable to install BoolSPL to the CUDA include directory, then you can place BoolSPL somewhere in your home directory, for example: /home/nathan/libraries/.

1. #include the "umbrella" <[BoolSPL/BoolSPLG\_v0.cuh](https://nvlabs.github.io/cub/cub_8cuh.html)> header file in your CUDA C++ sources;
2. Compile your program with NVIDIA's nvcc CUDA compiler, specifying a -I<path-to- BoolSPL> include-path flag to reference the location of the BoolSPL header library.

**Examples**

BoolSPL distribution directory contain “examples” directory with examples (Boolean and S-box) programs. For the examples to work there is need to include (add) the additional header files from the directory “help and additional header files”. This additional header files contain CPU Boolean and S-box function used for comparison and checking the obtained results from GPU functions.

**Reference and Publications related to the BoolSPL library**

[1] Bikov D., and I. Bouyukliev, BoolSPLG: A library with parallel algorithms for Boolean functions and S-boxes for GPU, preprint.

[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, 21–43 (2018).

[3] Bikov D., and I. Bouyukliev, Parallel Fast Mobius (Reed-Muller) Transform and its Implementation with CUDA on GPUs, Proceedings of PASCO 2017, Kaiserslautern, Germany, Germany — July 23 - 24, 2017, ISBN: 978-1-4503-5288-8 (**improvement presented in this publication are implemented in current v0.2 BoolSPL** **library**)

[4] CUDA homepage, Availaible on: <http://www.nvidia.com/object/cuda_home_new.html>

**Additional - Reference and Publications related to the BoolSPL library**

[1] I. Bouyukliev, D, Bikov, Applications of the binary representation of integers in algorithms for boolean functions, Proceedings of the Forty Fourth Spring Conference of the Union of Bulgarian Mathematicians SOK “Kamchia”, (2015), pp.161-166, ISSN: 1313-3330

[2] D. Bikov, I. Bouyukliev, Walsh Transform Algorithm and its Parallel Implementation with CUDA on GPUs, Proceedings of 25 YEARS FACULTY OF MATHEMATICS AND INFORMATICS, Veliko Tarnovo, Bulgaria, (2015), pp. 29-34, ISBN: 978-619-00-0419-6

[3] D. Bikov, I. Bouyukliev, A. Stojanova, Beneﬁt of Using Shared Memory in Implementation of Parallel FWT Algorithm with CUDA C on GPUs, Proceedings of 7th International Conference Information Technologies and Education Development, Zrenjanin, Serbia, (2016) pp.250-256, ISBN 978-86-7672-285-3

[4] I. Bouyukliev, D. Bikov, S. Bouyuklieva, S-Boxes from Binary Quasi-Cyclic Codes, Electronic Notes in Discrete Mathematics Volume 57, (2017), pp. 67–72, SJR 0.320

[5] D. Bikov, I. Bouyukliev, S. Bouyuklieva, Bijective S-boxes of Different Sizes Obtained from Quasi-Cyclic Codes, submitted.

**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 Bulgarian Science Fund under Contract DN-02-2/13.12.2016