Parallel primitive Histogram and use in Cryptography

Dushan Bikov

Faculty of Computer Science, Goce Delchev University, Shtip, Macedonia,

National Seminar in Mathematical Software and Combinatorial Algorithms

December 8, 2020

Motivation

Nowadays parallel programming is stepping up on the big door and slowly surpasses the traditional sequential programming model.

The use of modern Graphics Processing Units (GPUs) has become attractive for scientific computing which is due to its massive parallel processing capability.

- Here is presented implementation of parallel Histogram for computing Walsh distribution (Spectrum) of a Boolean function.
  - Algorithm for computing the Linear (Autocorrelation, Algebraic degree) distribution will take a part in next version of BoolSPLG (CUDA Library)
  - A histogram is an representation of the distribution of data

Boolean function and Linearity

Boolean function f of n variables is a mapping from $\mathbb{F}_2^n$ into $\mathbb{F}_2$, where $\mathbb{F}_2 = \{0, 1\}$ is a field with two element or every function: $f : \mathbb{F}_2^n \to \mathbb{F}_2$.

Truth Table $TT(f)$ is the 2^n-dimensional vector whose coordinates are the function values of f after the lexicographic ordering of the inputs.

Walsh transformation of f is function $f^W : \mathbb{F}_2^n \to \mathbb{Z}$, defined by

$$f^W(a) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus f_a(x)} = \sum_{x \in \mathbb{F}_2^n} (-1)^{f_a(x)} (-1)^{f(x)},$$

where $f_a(x) = a_1 x_1 \oplus a_2 x_2 \oplus \cdots \oplus a_n x_n$.

The values of f^W are called *Walsh coefficients* and the 2^n-tuple $(f^W(\bar{0}), f^W(\bar{1}), \ldots, f^W(\overline{2^n - 1}))$ is called the *Walsh spectrum* $[W_f]$ of the Boolean function f.
The set $\{W_f(i), -2^n \leq i \leq 2^n\}$, where $W_f(i)$ is the number of Walsh coefficients with value i, is called *Walsh distribution* of the function f.

CUDA - Compute Unified Device Architecture

CUDA is a powerful parallel computing platform and API created by NVIDIA for general purpose computing of GPU

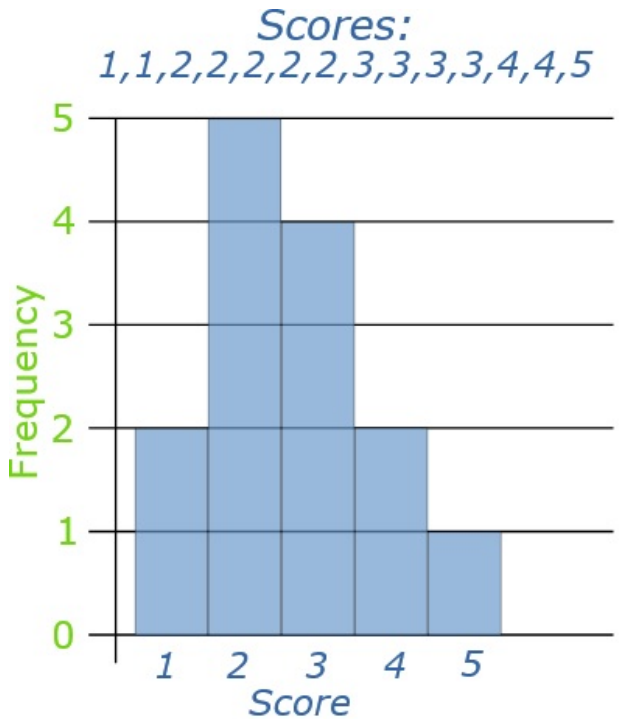
- Platform allows direct interaction with the GPU and speed up computing applications by harnessing parallel computing resources
- On the top level on CUDA application there is master process that is run on CPU who is responsible for:
  - initialization, allocate and moving memory between main memory and GPU memory;
  - launching the kernels on the GPU that perform computation;
  - fetching back the memory once the computation is completed;
  - deallocation of the memory;
  - termination.

Histograms

Histograms are a commonly used analysis tool in image processing and data mining applications.
They show the frequency of occurrence of each data element.

Frequency Histogram

- is a special graph that uses vertical columns to show frequencies (how many times each score occurs):



Parallel implementation of Histogram (1)

Although trivial to compute on the CPU, histograms are traditionally quite difficult to compute efficiently on the GPU.

- Calculating an histogram on a sequential device with single thread of execution is easy (pseudocode):

```
for(int i = 0; i < BIN_COUNT; i++)
    result[i] = 0;

for(int i = 0; i < dataN; i++){
    result[data[i]]++;
```

- Implementing parallel histogram using atomics
  - Histogram kernel using global memory atomic

```
15  // the simplest histogram
16  __global__
17  void copmute_histogram_1(int* hst, const int* data, int numElements)
18  {
19      int x = blockIdx.x * blockDim.x + threadIdx.x;
20
21      if (x >= numElements)
22          return;
23
24      int idx = data[x];
25      atomicAdd(&(hst[idx]), 1);
26  }
27
```

Parallel implementation of Histogram (2)

- Implementing parallel per-thread (local) histogram then reduce (local histograms).
  - Local histogram using Local Memory
  - limitations from local memory, small amount of bins (histogram buckets)
- Implementing parallel histogram by Sort then Reduce by Key

- My approach
  - Sort then Parallel Compact
    - Parallel Compact: given some predicate function remove those elements which return false and "squeeze" the data into the "**required space**".
    - Exclusive Scan is main building block of the parallel compact.

Parallel Prefix Sum (Scan)

Definition: The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns the ordered set

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

Example (Types of Scan): if $\oplus$ is addition, then scan on the input set:

INPUT	1	2	3	4	5	6	7	8

- Exclusive Scan: Output all elements excluding the current

OUTPUT	0	1	3	6	10	15	21	28

- Inclusive Scan: Output all elements including the current

OUTPUT	1	3	6	10	15	21	28	36

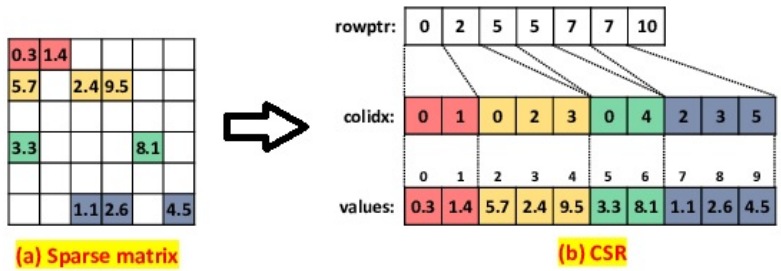
Scan and Sort

- Application of **Scan** - implementation of several parallel algorithms:
  - radix sort, quick sort, String comparison, Lexical analysis, Stream compaction, Polynomial evaluation, Solving recurrences, Tree operations, Histograms, Etc.
- Lots of serial **Sort** algorithms! Fewer Parallel???
  - Goal - find efficient parallel algorithm
    - Keep hardware busy (lots of threads)
    - Limit branch divergence
    - Preferred coalesced memory access
  - Parallel implementation of Sort algorithms
    - quick sort, merge sort, radix sort, bitonic sort, butterfly sort, etc.

Addition remarks about the algorithm

Sparse vector representation (optimize CUDA data transfer)

- Don't store **zeros** - unnecessary data
- Similar to the Sparse Matrix
  - Compressed Sparse Row (CSR) Format, Sparse Matrix representation



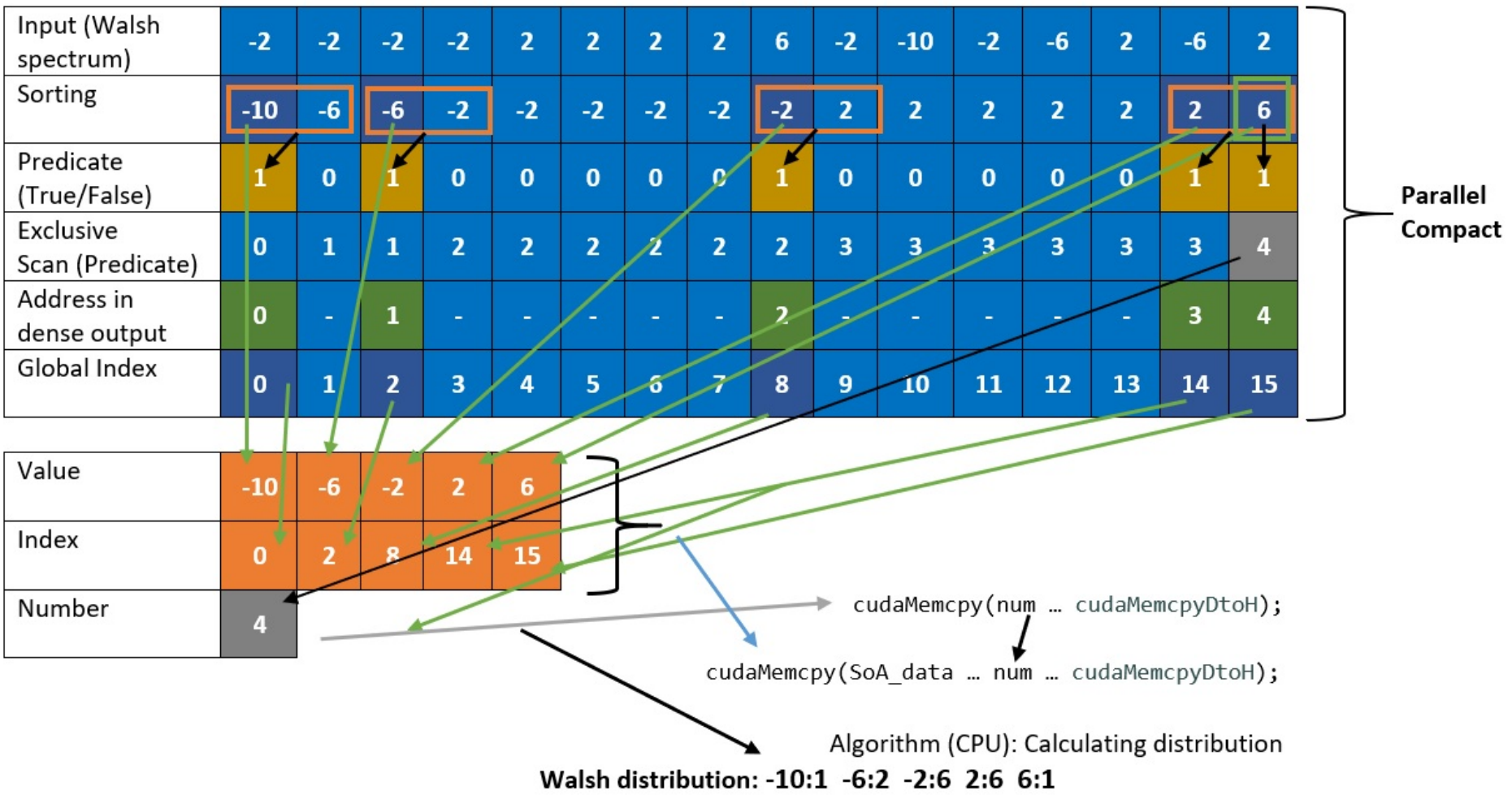
Note: How to store the results (Sparse vector)

- Structure of Arrays (SoA) vs Array of Structures (AoS)

```
// AoS wastes bandwidth
int key1 = AoS_data[i].key;

// SoA efficient use of bandwidth
int key2 = SoA_data.keys[i];
```

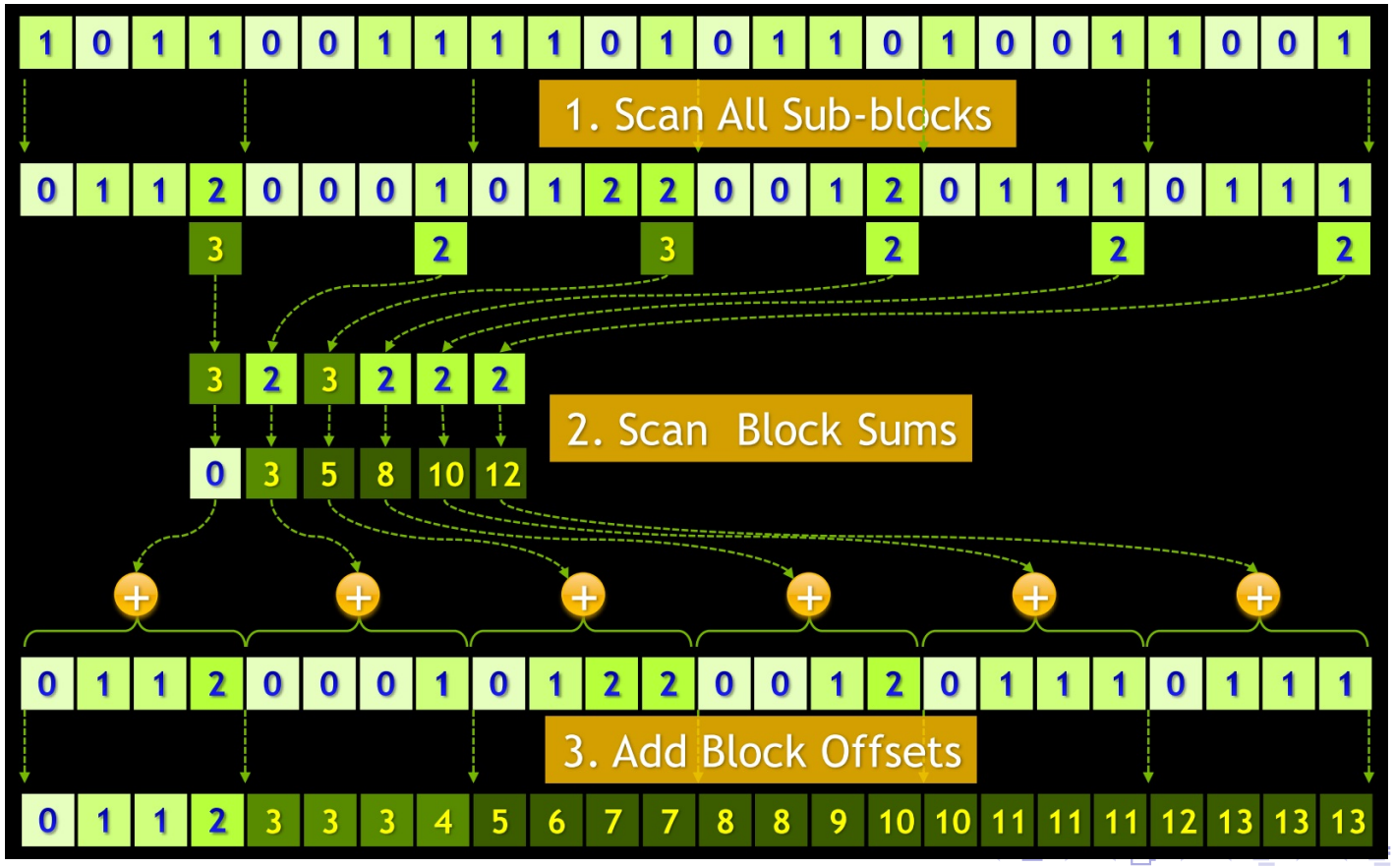
Parallel Algorithm for computing of Walsh distribution



Implementing Scan: Strategies (1)

Scan is recursive

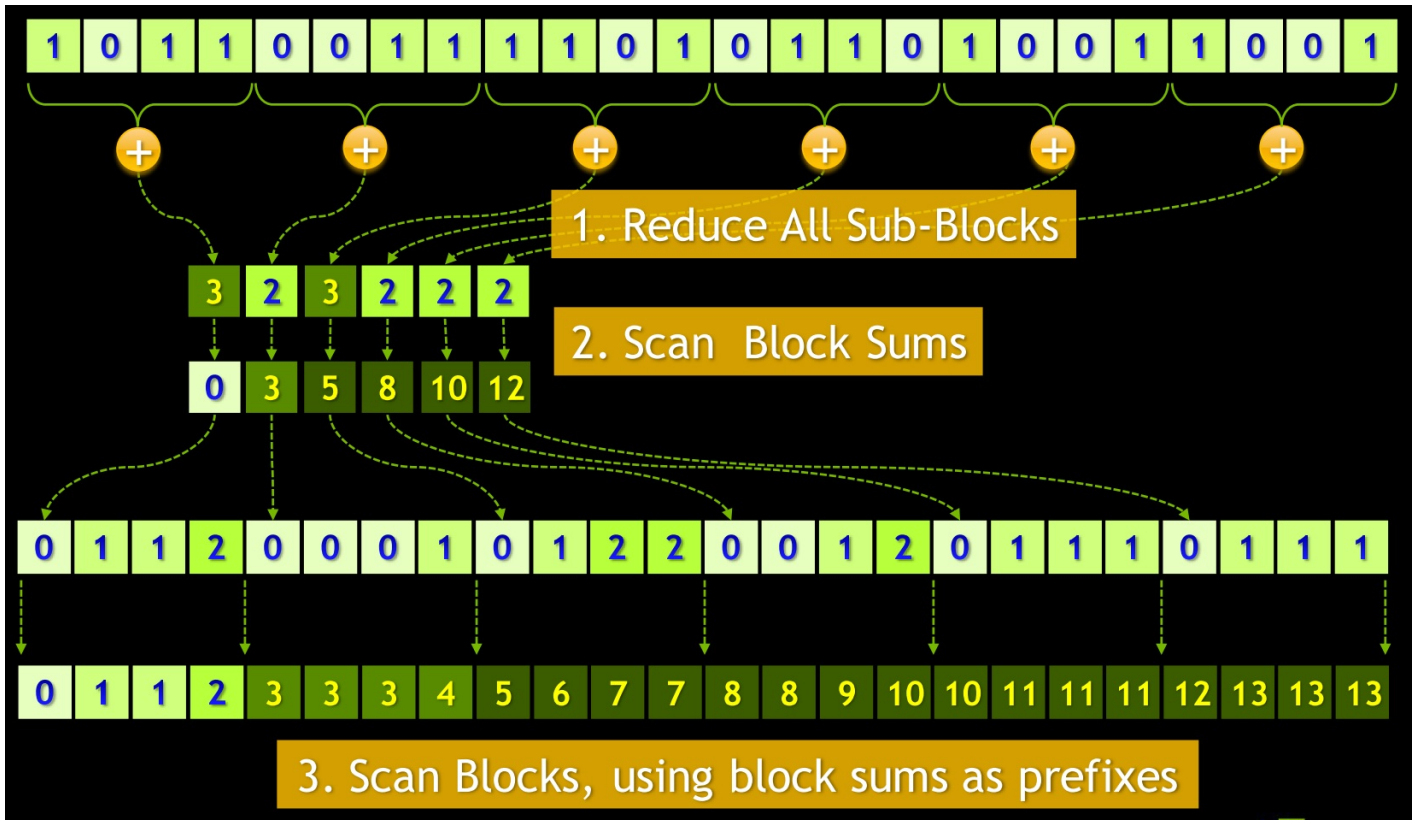
- Build large scans from small ones
  - In CUDA, build array scans from thread block scans
  - Build thread block scans from warp scans
- One approach:-Scan-Scan-Add



Implementing Scan: Strategies (2) - next to tray

Improvement: Reduce-then-Scan

- Scan-Scan-Add requires 2 reads/writes of every element
- Instead, compute block sums first, use as prefix for block scans
  - 25% lower bandwidth



Test Platform

Environment	**Platform**
CPU	Intel i7-8565U, 1.80GHz
Memory	8 GB DDR4 2400 MHz
GPU	GeForce MX150
OS	Windows 10, 64-bit
IDE	MS Visual Studio 2013
CUDA	10.1
GPU Driver	V457.51

Table: Description of the test platforms

In the next experiments - CUDA Time include Compute, copy data GPUtoCPU - Walsh spectrum is already in Global GPU memory

Platform, preliminary results

size	CPU (ms)	GPU (ms)	speed up	# Walsh coefficient
2^{10}	15	0.2	$x75$	48
2^{12}	31	0.4	$x77.5$	106
2^{14}	78	0.6	$x130$	219
2^{16}	156	0.9	$x173$	458
2^{18}	188	7.9	$x23$	459
2^{20}	203	37	$x5.5$	459

Table: Preliminary results

* Input Boolean functions (Walsh spectrum) is random - should be random

Acknowledgment

This research is supported by Bulgarian Science Fund under Contract DN-02-2/13.12.2016.


Q
Q
&
&
A
A