Multi-GPU Approximation Methods for Silent Data Corruption of AN Codes

Matthias Werner* Till Kolditz† Tomas Karnagel† Dirk Habich†
Wolfgang Lehner†
* Center for Information Services and High Performance Computing
† Database Technology Group
Technische Universität Dresden, Germany
* Matthias.Werner1@tu-dresden.de † {firstname.lastname}@tu-dresden.de

Abstract

Multi-bit flip rates are assumed to increase dramatically with future transistor technologies, especially in computer DRAM. We previously showed that employing AN codes in main-memory database systems is a feasible choice in terms of performance overhead and memory consumption [1]. The silent data corruption probability of a code is determined by its distance distribution, whose computational complexity is exponential for non-linear codes like AN coding. We provide exact and approximation algorithms for computing the distance distribution on GPUs for AN codes.

1 Introduction

Error detecting / correcting codes (EDC / ECC) have been widely studied in theory and applied in practice [2–4]. Linear codes like Hamming are easy to implement, offer low coding overhead and constant decoding time. They have well-defined, but limited detection and correction capabilities due to their algebraically determined structure, where the linear code represents a vector space over a Galois field GF(q) ($q = 2$ for Boolean space) [3, 4].

Efficient decoding algorithms have been implemented in hardware and single error correcting, double error detecting Hamming is nowadays used in server-grade main memory (ECC DRAM). AN-coding [5], being one representative of non-linear codes, only offers efficient error detection capabilities. Since AN codes are non-systematic, error correction would have to be done in a brute force manner and more efficient correction is an open problem. However, some non-linear codes provide better reliability than any linear code with the same parameters [6]. While linear codes are typically used in communication systems, AN codes have been used to detect errors in hardware itself [4, 7, 8] and current research advocates to use it in main-memory database systems for bit flip detection as well [1].

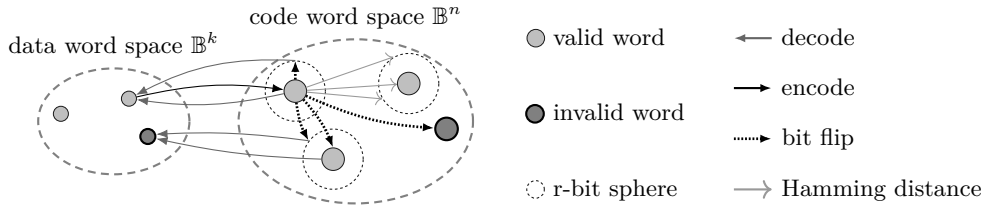

Figure 1: Encoding of data words and decoding of code words.

The reliability of a (non-)linear code depends on how many bit flips can be detected or even corrected after an error-prone transmission of a code word. With the Boolean set $\mathbb{B} = \{0, 1\}$ an injective function $\phi : \mathbb{B}^k \to \mathbb{B}^n$, $n{>}k{>}1$, defines the *code* by $C = \mathrm{Im}(\phi) \subseteq \mathbb{B}^n$ [9]. A given *data word* $x \in \mathbb{B}^k$ is mapped to a *code word* $u \in C$, which can be decoded back to the original data word as illustrated in Figure 1.

The *r-bit sphere* of a code word vector $u \in C$ contains all vectors $v \in \mathbb{B}^n$ with a Hamming distance (see Definition 1) $d_H(u, v) \leq r$, [3, p. 11]. The r-bit spheres do not overlap, if the minimum distance d between all different code words is $d > 2r$ (for case $d = 2r$ see [3, p. 10]). If $\leq r$ bit flips occur during a transmission, an ECC is able to correct such an invalid code word. For instance, Hamming [10] as a linear code with $d = 3$ is able to detect double bit flips and to correct single bit flips. When $> r$ bits flipped, this results either in an uncorrectable invalid code word or, even worse, in a code word within a bit sphere of another code word. The latter results in a decoding error where the decoder returns a data word without being aware of the error, which is referred to as *silent data corruption* (SDC).

Recent studies suggest that the *single event upset* (SEU) error model will become obsolete while multi-bit flips already lead to uncorrectable data corruption [11–14]. It is assumed that both rate and number of bits flipped will increase with future transistor technologies [11]. In the context of the database domain, where users typically expect accurate results, we see that SDC has severe impact on all tasks of database systems. For query processing as example, joins may be incomplete when tuples' join attributes are altered, or filtered scans may have missing or even additional tuples when the filtered values contain bit flips. In short, SDC leads to false positives and false negatives in query (intermediate) results.

In contrast to typical channel-based considerations of EDCs or ECCs, we try to understand robustness of codes being used for storing data in a computer system's main memory within the scope of in-memory database systems. There, all important business data is stored in main memory (DRAM). The problem here is that the probability of errors depends at least on the hardware (DRAM) technology, *each* of the memory cells' susceptibility to external influences[1], each cell's degradation due to aging, as well as the time between successive writes and reads. Furthermore, it is not yet clear whether bit flips will be independent or not (burst errors, ...). Like Forin [5] we argue that, currently, it is not feasible to assume a specific error model.

We earlier proposed AN codes as a software coding approach for in-memory database systems [1], showing that it comes at little or no costs in terms of throughput. We envision AN codes as a complementary, or even alternative, to hardware ECC with Hamming codes in the area of database systems to better detect multi-bit flips in main memory. It can be an alternative, because database systems inherently store all data redundantly anyways (materialized views and indexes, recovery log, ...).

AN codes map data to code words by multiplying a factor A. To find good factors A the probabilities of SDC have to be taken into account. The biggest problem is that, for each data width, each A behaves differently. Here, brute force attempts were made for data widths up to 16 bits [7, 8], where basically all code words are checked against all possible error patterns. These previous approaches are insufficient: On the one hand, in-memory database systems operate on the larger native data widths of currently up to 64 bits. On the other hand, the computational effort can be reduced, if only closest pairs of code words are examined. For AN codes the complexity of brute force is $\mathcal{O}(2^{2^k})$ [9]. An instance with $k = 32$ would take 213 days assuming a modern multi-GPU node with 10^{12} operations/s.

In this paper, we use sampling methods like Monte-Carlo or lattice points to reduce runtime at reasonable costs of accuracy. Since this task is highly parallelizable, GPUs are used for further acceleration. Value ranges of A and their probabilities of SDC for data widths up to 24 bits are examined. Runtimes and computation of optimal values of A are shown for data widths up to 32 bits using approximation and exact algorithms where possible.

The remaining paper is structured the following way: In Section 2 we introduce AN codes and our measure of SDC probability. In Section 3, we show how the distance distribution is computed for AN codes and elaborate on the GPU implementation details. Then, the results are discussed in Section 4 and, finally, we give a conclusion in Section 5.

2 Prerequisites

Regardless of a specific error model we provide a methodology for determining the probability of SDC for coding schemes in general. For describing and evaluating the algorithms, the following definitions are introduced.

[1] like cosmic rays, heat, voltage fluctuations, electrical crosstalk, etc.

Definition 1. *The bit width of data words is assumed to be* $k \geq 2$.

- *Let be* $\mathbb{B} = \{0, 1\}$ *the Boolean set.* $\mathbb{B}^k$ *is the* k*-fold Cartesian product of* $\mathbb{B}$.
- *For* $\alpha \in \mathbb{B}^k$ $\mathrm{wt}(\alpha)$ *returns the number of bits with value 1 (weight of* α*).*
- $d_H(\alpha, \beta) = \mathrm{wt}(\alpha \oplus \beta)$ *is the Hamming distance with the bitwise XOR operator* $\oplus$.
- $\delta_b(x, y)$ *defines the indicator function:*

$$\delta_b(x,y) = \begin{cases} 1, & \text{if } d_H(x,y) = b, \\ 0, & \text{if } d_H(x,y) \neq b, \end{cases} \qquad 0 \leq b \leq n\,.$$

- *The AN code is defined by a constant generator* A, $h{=}n{-}k{=}\lceil \log_2(A) \rceil$, *and by*

$$C_A = \{A \cdot x \in \mathbb{B}^n : x \in \mathbb{B}^k\}\ .$$

A is chosen to be an odd number, since even numbers are left shifted odd numbers [7, p. 94]. The AN code C_A preserves the code words with respect to addition, but not for multiplication as $Ax_1 \cdot Ax_2 = A^2(x_1 \cdot x_2) \neq A(x_1 \cdot x_2)$ [4, p. 103]. AN codes are both non-linear and non-systematic. A code word pair (u, v) with $d_H(u, v) = b$ describes an undetectable b-bit flip, since $u, v \in C_A$. For computing the SDC probability, the Hamming distances between all code word pairs have to be enumerated yielding the distance distribution [9].

Definition 2. c_b^A *denotes the distance distribution of code* C_A *by*

$$c_b^A = \left|\{(u, v) \in C_A^2 : d_H(u, v) = b\}\right|$$

For a code word of length n there can be $\binom{n}{b}$ different b-bit flips, because the order of how the bits flip is unimportant. Each c_b^A is related to the number $2^k \cdot \binom{n}{b}$, which represents all possible b-bit patterns over all code words in C_A. This yields the SDC probability p_b^A for b-bit flips:

$$p_b^A = \frac{c_b^A}{2^k \cdot \binom{n}{b}} \tag{1}$$

If $A = 1$ or no encoding takes place, then $n = k$, $c_b = 2^k\binom{k}{b}$ and $p_b = 1$ are obtained.

3 Computing Distance Distribution of AN Codes

By Definition 1 the distance distribution can be written as:

$$c_b^A = \sum_{\alpha \in \mathbb{B}^k} \sum_{\beta \in \mathbb{B}^k} \delta_b(A\alpha, A\beta) \tag{2}$$

AN codes are non-linear and $(A\alpha) \oplus (A\beta) \neq A(\alpha \oplus \beta)$. The two sums cannot simply be reduced to a single sum unlike Hamming, where the distance distribution can be computed directly from the weight enumerator [3, 125ff.]. The complexity of Equation 2 is $\mathcal{O}(4^k)$. For parameter optimization the algorithm might run thousands of times for different values of A. To approximate the distance distribution, only certain parts of the sum iterations are computed, which is restricted by the desired number of iterations M. The Monte-Carlo method achieves this by using random samples over a domain Ω to estimate the definite integral of a function f:

$$\frac{V}{M} \sum_{i=1}^{M} f(\vec{x_i}) \approx \int_\Omega f(\vec{x}) \mathrm{d}\vec{x}, \qquad V = \int_\Omega \mathrm{d}\vec{x}$$

The sample points $\vec{x_i}$ shall cover Ω as the number of iterations M grows.

$$\hat{c}_b^A = \frac{2^k \cdot 2^k}{N \cdot M} \sum_{r=1}^{N} \sum_{s=1}^{M} \delta_b(A \cdot \sigma_1(r), A \cdot \sigma_2(s)) \approx c_b^A \tag{3}$$

Equation 3 estimates c_b^A by using sample streams σ_1 and σ_2. The samples are generated either by σ_{pseudo} of pseudo-random numbers or by σ_{quasi} of quasi-random numbers. Pseudo-random numbers are prone to clustering, while quasi-random numbers fill the space more uniformly. The probabilistic error of Monte-Carlo is known to be $\mathcal{O}(\frac{1}{\sqrt{M}})$ and for quasi-Monte-Carlo it is $\mathcal{O}(\frac{(\log M)^q}{M})$ with number of dimensions q [15]. We also examine regularly aligned samples, also called lattice or grid points, which are given by $\sigma_{\text{grid}}(r) = \frac{2^k \cdot r}{M}$. It will turn out that M should be an odd value for better convergence. If $M = 2^k$, then the grid sampling yields the correct result, while the random numbers still miss the solution due to collisions and gaps.

Algorithm 1 AN code distance distribution – basic algorithm

Input: $k \geq 2$
Input: Value $A > 0$, $n = k + h$, $h = \lceil \log_2(A) \rceil$
Input: Initial distance distribution $c_b^A = 0$, $b = 0, \ldots, n$
Output: Distance distribution c^A of code C_A

```
1: for α = 0, ..., (2^k − 1) do          ▷ outer loop is parallelized on GPU[s]
2:     for β = 0, ..., (2^k − 1) do      ▷ inner loop is processed by each thread
3:         b ← d_H(Aα, Aβ)
4:         c_b^A ← c_b^A + 1
5:     end for
6: end for
7: return c^A
```

Algorithm 1 shows the basic procedure for enumerating the Hamming distances of all code word pairs. For 1D sampling, line 2 is replaced by $\beta = \sigma(r), r = 0, \ldots, M$. For 2D sampling[2], line 1 is then replaced by $\alpha = \sigma(s), s = 0, \ldots, M_2$. As a simplification, the exact algorithm exploits symmetry and starts at $\beta = \alpha + 1$ in line 2 (then line 4 counts twice and $c_0^A = 2^k$).

The outer loop is mapped to equal workloads onto the GPUs. If symmetry in line 2 is used, then the inner loop iterations would decrease linearly in each step. To distribute equal workloads across all GPUs, the workload size of GPU i is computed by:

$$\lceil 2^k \omega_{i+1} \rceil - \lceil 2^k \omega_i \rceil, \quad w_i = 1 - \sqrt{1 - \frac{i}{N}}, \quad 0 \leq i < N = \text{number of GPUs} \tag{4}$$

ω_i is the solution of $\int_i^{i+1} 1 - x \, \mathrm{d}x = \frac{1}{N}$ for equal work size areas. The maximal relative error of the estimation $\hat{c}_b^A$ is given by $\Delta = \max_{b>0} \frac{|c_b^A - \hat{c}_b^A|}{c_b^A}$ ($b = 0$ is omitted due to $c_0^A = 2^k$).

Algorithm 1 can be parallelized on GPU, since the Hamming distances of two code words can be computed independently. We use the CUDA C/C++ for programming Nvidia GPUs. Listing 1 shows the CUDA C/C++ implementation of the Algorithm 1. `UINT` is a template type for unsigned 32-bit or 64-bit integers. **`threadIdx.x`** is the thread index in a thread block, e.g. $0 - 127$. **`blockDim.x`** is the number of threads per block, e.g. 128. **`blockIdx.x`** is the block index. `gridDim.x` is the number of blocks.

As registers of GPUs are 32-bit wide, the multi-GPU implementation uses 32-bit integers as long as the array elements in a thread do not overflow. With $\max_b c_b^A \leq \max_b 2^k \binom{n}{b} = 2^k \binom{n}{n/2}$ from Equation 1 the following upper bound determines when to use to the 32-bit integers:

$$c_{b,\text{thread}}^A \leq \frac{2^k \binom{n}{n/2}}{\text{threads}} < 2^{32} \ .$$

It should be possible to use $\frac{2^k \binom{k}{k/2}}{\text{threads}}$ as a lower border, since higher values of A make the distance distribution more wide and flat to keep the total sum equal to 4^k. If 4 GPUs are used each with 64 000 threads, then the 64-bit implementation is used when $k \geq 27$. As the GPU uses 64-bit for the global array, the highest bit width for running the GPU algorithm is $k = 33$.

`distr` is the distance distribution array, `offset`=$\lceil 2^k \omega_i \rceil$, `end`=$\lceil 2^k \omega_{i+1} \rceil$ from Equation 4 and `Aend`=$A \cdot (2^k - 1)$. `distr_local` is a thread local array with `DISTR_SIZE` elements. Since local array indexing is dynamic and non-uniform, it cannot be stored into the fast registers, as

[2]For 2D sampling only σ_{grid} has been implemented

Listing 1: CUDA Implementation

```
1 template<uint32 DISTR_SIZE, typename UINT>
2 __global__
3 void ancoding(UINT A, uint64* distr, UINT offset, UINT end, UINT Aend) {
4   /// stored in LocalMemory
5   UINT distr_local[DISTR_SIZE] = { 0 };
6   UINT v, w;
7   /// grid-striding loop of data words
8   for (UINT i = blockIdx.x * blockDim.x + threadIdx.x + offset;
9        i < end;
10       i += blockDim.x * gridDim.x) {
11     /// encode data word i to code word w
12     w = A*i;
13     /// loop succeeding data words
14     for(v=w+A; v<Aend; v+=A) {
15       /// hamming distance processed by local histogram
16       ++distr_local[ hamming_distance(w, v) ];
17     }
18   }
19   /// add local histogram to global memory
20   for(uint_t c=0; c<DISTR_SIZE; ++c)
21     atomicAdd(distr+c, distr_local[c]);
22 }
```

they are not addressable at runtime. Hence, `distr_local` has to be stored in local memory, which is L1 cached thread-private global memory. The L1 cache configuration is maximized by `cudaDeviceSetCacheConfig(cudaFuncCachePreferL1)`, which can double the speedup on pre-Maxwell GPUs, where shared memory and L1 cache are sharing storage.

To get scalable and flexible kernels, the outer loop strides by the size of a CUDA grid (threads per block × blocks per grid). The kernel is called with `blocks`=32 · numberOfMultiprocessors and 128 threads per block. After the local histogram is filled, atomic operations are used to add the values to the global distance distribution.

The probabilities p_b^A decrease with increasing h as the total sum $\sum_{b=0}^{k+h} c_b^A = 4^k$ holds for all h (cp. Equation 2) and $\frac{\text{number of code words}}{\text{number of all words}} = \frac{2^k}{2^{k+h}} = 2^{-h}$ [7, p. 94]. We assume that few bit flips occur more likely than many bit flips at once. Hence, our definition of an optimal A, also called "super" A [8], is to minimize the first positive histogram value for a given h:

$$A_h^* = \underset{\substack{2^{h-1}<A<2^h \\ A \text{ is odd} \\ b\to\max}}{\arg\min} \; c_b^A \,|_{c_1^A,\ldots,c_{b-1}^A=0} \; . \tag{5}$$

For the database domain it is important to control the storage overhead, which is why we want to find an optimal A for each width h. The implementation[3] also provides 2D grid point sampling and supports code word widths up to 64 bits.

4 Results

Table 1 provides the average runtimes for computing the distance distribution of AN codes for a single A with value 61. In column "exact" t denotes the time to solution on CPU or GPU(s). t_M is the runtime depending on the number of iterations M. Δ_M is the maximal relative error after M iteration steps.

The runtime of the (multi-)GPU implementation scales about linearly with the number of GPUs up to $k = 32$. The 24 Haswell CPU cores do not use vector instructions like AVX and are almost 4× slower than a single K80 GPU. The GPU performance is memory bound, which is a result of the dynamic non-uniform array access of c_b^A in line 4 of Algorithm 1.

[3] `https://github.com/tuddbresilience/coding_reliability`

Table 1: Computing distance distributions of AN codes for A=61

	exact			$\sigma_{\text{grid,1D,1×GPU}}$		$\sigma_{\text{grid,1D,4×GPU}}$		
k	t_{CPU}	$t_{1\times\text{GPU}}$	$t_{4\times\text{GPU}}$	t_M	Δ_M	t_M	Δ_M	M
8	7 ms	1 ms	3 ms	11 ms	0.0232	6 ms	0.0232	101
16	376 ms	130 ms	41 ms	26 ms	0.0031	11 ms	0.0031	1001
24	382 min	99 min	27 min	1261 ms	0.0053	354 ms	0.0053	1001
32	–	–	–	19 min	–	5 min	–	1001

Average values after 5 runs on the Bull HPC-Cluster at TU Dresden
CPU: 2×E5-2680 v3 Haswell 12-core 2.50 GHz, gcc5.3, OpenMP 4.0
GPU: NVIDIA Tesla K80, CUDA 7.5

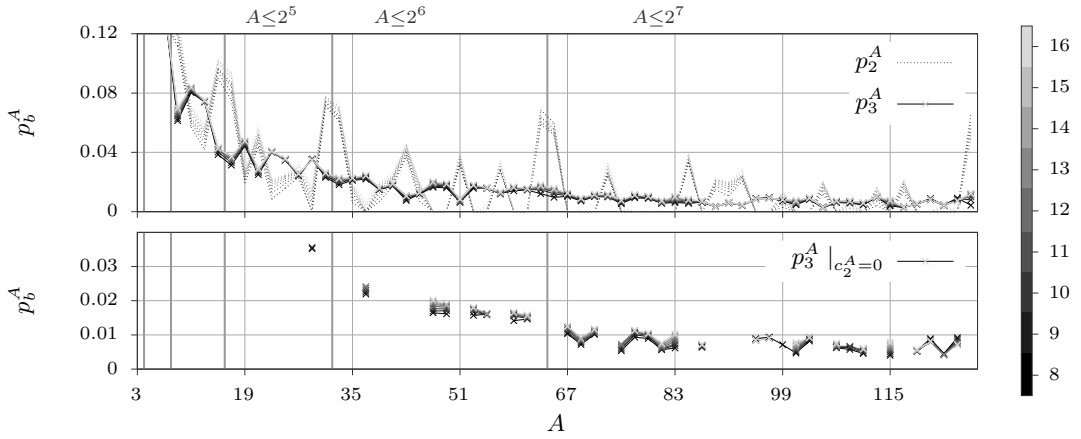

Figure 2: Finding optimal A's (only odd values) by Equation 5 for $k \in \{8, \ldots, 16\}$

These local arrays cannot be stored in GPU registers, because they cannot be addressed dynamically at runtime[4]. The array is stored in local memory instead (thread-private global memory), where L1 cache becomes the bottleneck. If the 64-bit GPU implementation is used, then the runtime took about 3-4× longer than using 32-bit on a single K80 GPU.

For $k = 16$ and $A = 61$ the Nvidia profiler shows that the load/store units of the GPU multiprocessors are utilized by almost 90 %. The L1 local memory cache hit rate reaches almost 99 %. However, there is an average of 6 local load/store transactions per request. As a consequence, the load/store request of a CUDA warp (32 threads, SIMD) must be replayed 6 times due to uncoalesced memory access, until all the requests have been served. This coincides with a benchmark kernel using registers, which showed a 7× better performance than using local memory.

Figure 2 shows several things: firstly, how an increasing value of A gradually decreases the SDC probability p_2^A and p_3^A, and secondly, that p_b^A differs for varying k. The lines are color coded representing different k's. Dashed (solid) lines show the probability of silent data corruption under 2-bit (3-bit) flips (respectively). The upper graph shows all probabilities, whereas the lower graph emphasizes p_3^A under the condition $c_2^A = 0$ (cf. Definition 2 and Equation 1). The graph also shows how generator A creates relatively high probabilities of SDC, if $A = 2^i - 1$ or $A = 2^i + 1$ for $i = \{2, \ldots, 7\}$. Although the curves are mostly almost equal among the values of k, irregular peaks for different A are observed.

Figure 3 shows the convergence of the three different 1D sampling methods for $A = 61$ and $k = 16$ (left) and $k = 24$ (right). In almost all cases the grid points (*grid*) outperform the random numbers (*pseudo*, *quasi*), generated with the cuRAND library, in terms of both runtime (t) and relative error (Δ). The left graph furthermore shows that for $M = 2^{16}$ grid points compute the solution with some overhead in runtime. The 1D grid approximation is directly influenced by the value M, where odd values lead to much smaller errors.

[4]Workaround with if/switch-case branches would be even more costly here.

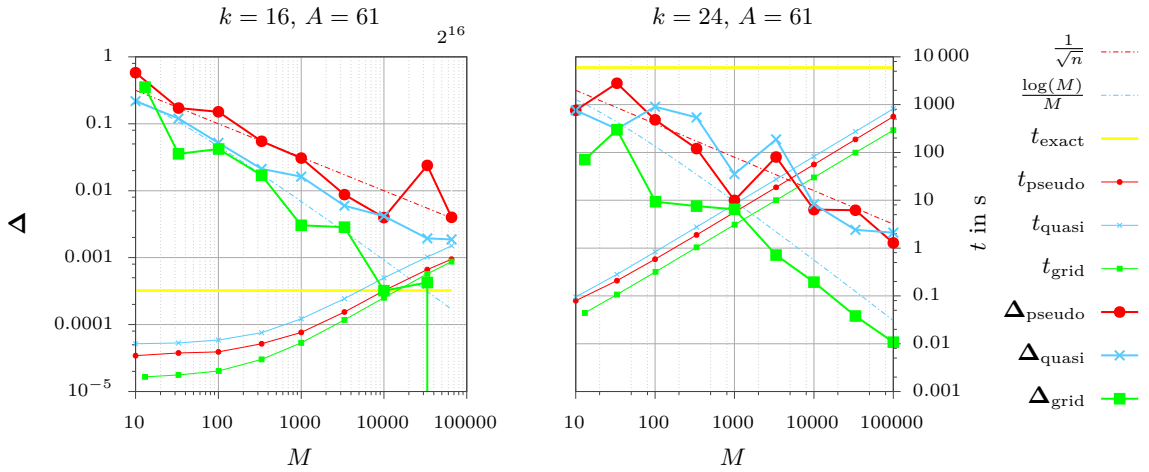

Figure 3: Convergence of maximum relative error Δ for 1D sampling according to the number of iterations M. Run on a single K80 GPU.

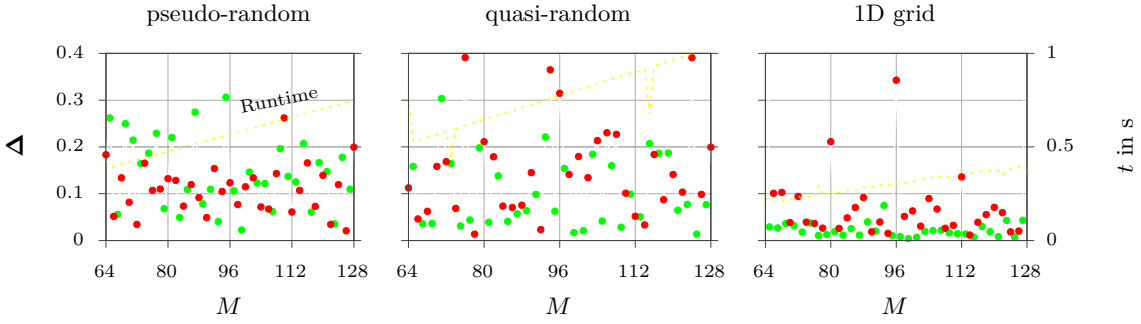

Figure 4: Influence of number of iterations M on runtime t and relative error Δ ($k = 24$, $A = 61$). Filled (empty) circles denote even (odd) Ms. Run on a single K80 GPU.

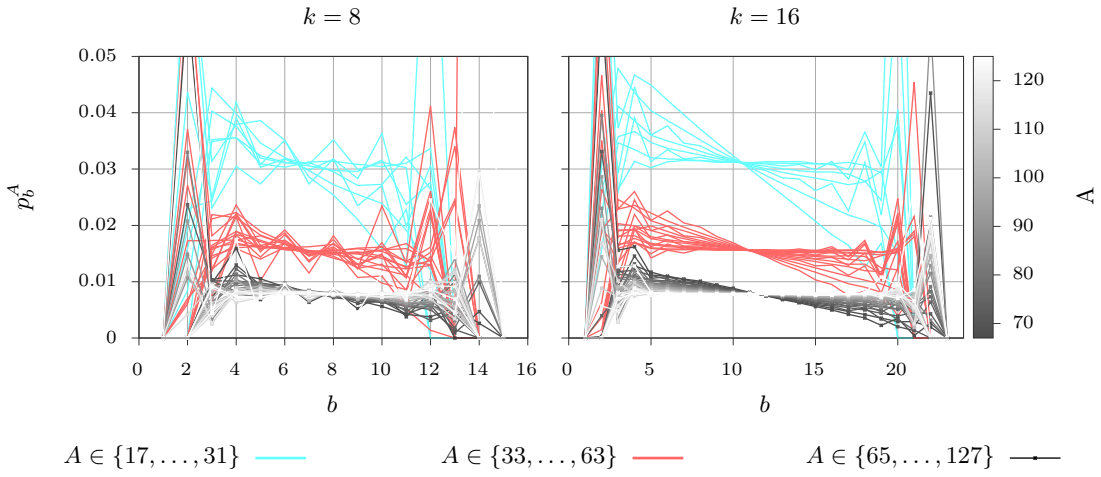

Figure 5: How value A controls level of probability for $k = 8$ and $k = 16$

Figure 4 compares the impact of value M on the maximal relative error due to the approximation in greater detail. For random numbers the error seems uniformly distributed, while grid points indicate minimal values of relative error for odd Ms.

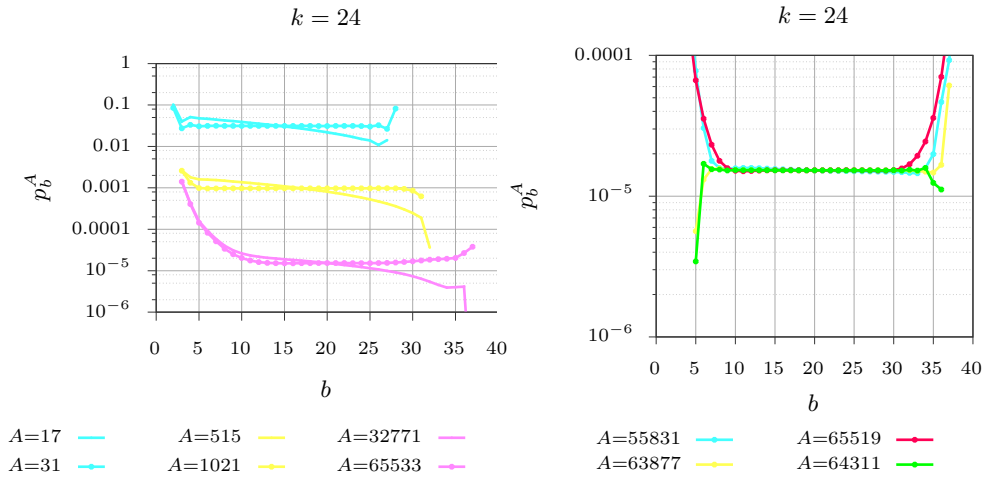

Figure 6: How value A controls level of probability for $k = 24$

Figure 5 and Figure 6 compare the probability functions depending on the generator A and data word width k. Within $2^{h-1} < A < 2^h$ low values of A give low probabilities on the right side of the probability function p_b^A. Conversely, values of A close to 2^h mostly yield lower probabilities on the left side of p_b^A. It looks like a seesaw being pushed down on the left. In contrast, at the borders (i.e. very small/large b) the SDC probabilities vary largely with no obvious pattern, which confirms previous findings [8]. The flat region follows 2^{-h}, $h = \lceil \log_2 A \rceil$ and grows with n. Large values of A are potentially more able to keep SDC probabilities at zero on the left side. However, Figure 6 with $A = \{55831, 63877\}$ and prime number 65519 demonstrates that, even in the upper range, choosing the wrong A can lead to very high probability of SDC for low values of bit flips b. $A = 64311$ is the super A found for $k = 24$ and $h = 16$.

Table 2 lists super As for different widths of A ($h \in \{3, \ldots, 16\}$) and different data widths ($k \in \{8, 16, 24, 32\}$), which adhere to our definition by Equation 5. Values k^* refer to the approximation algorithm using 1D lattice points with $M = 1001$ for $k = 24$ and $M = 3$ for $k = 32$. First of all, it shows that different As are optimal for the same h and varying k. Secondly, it shows the relation between the minimum Hamming distance of each code and h and k: for increasing k we obviously need larger h to keep the same minimum Hamming distance. It should be noted that not all super As are prime numbers and are always located close to 2^h.

Table 2: Super As and minimum Hamming distances for different data widths. *=approx., **bold**=prime.

h	$k=8$	$k=16$	$k^*=24$	$k^*=32$	h	$k=8$	$k=16$	$k^*=24$	$k^*=32$
3	**7** [2]	**7** [2]	**7** [2]	**7** [2]	10	**857** [4]	**947** [4]	981 [4]	**881** [4]
4	**13** [2]	**13** [2]	**13** [2]	15 [2]	11	1939 [5]	1939 [4]	1939 [4]	**2029** [4]
5	**29** [3]	**29** [2]	**29** [2]	21 [2]	12	3813 [5]	3349 [4]	3829 [4]	3565 [4]
6	**59** [3]	**61** [3]	**61** [3]	55 [2]	13	7463 [5]	7785 [5]	**6311** [4]	7947 [4]
7	115 [3]	119 [3]	111 [3]	125 [3]	14	**13963** [6]	14781 [5]	15993 [5]	16041 [5]
8	**233** [4]	**233** [3]	237 [3]	225 [3]	15	27247 [6]	**28183** [5]	29675 [5]	28691 [5]
9	**487** [4]	**463** [4]	423 [3]	445 [3]	16	55831 [7]	63877 [6]	64311 [5]	64311 [5]

5 Conclusion

In this paper the distance distributions and therewith SDC probabilities of AN codes are computed on GPUs. Since AN codes are non-systematic and non-linear, the complexity of $\mathcal{O}(4^k)$ is tackled with sampling methods. With an odd number of iterations M the uniform grid points show better runtimes and convergence than pseudo- or quasi-random number sampling.

The multi-GPU implementation accelerates computation of the distance distribution: e.g. for a single A and k=24 the 4×K80 GPUs compute the exact solution in 27 min, which is almost 16× faster than the 24 Haswell CPU cores. The 1D grid sampling algorithm returned the solution with a relative error of 10^{-3} after just 354 ms, despite the very small number of grid points of $M = 1001$ compared to 2^{24} code words in the exact algorithm. These techniques allow to approximate SDC probabilities for data widths of $k > 24$ in reasonable time. There is space left for optimizations. A recent implementation of a shared memory solution showed a speedup of about 1.5× on K80 and can be found on our github repository. Currently, the algorithms only can run for data word widths up to 32 bits due to overflow issues, but it can be extended to 64 bits using a global array variable with 128 bit integer in the CUDA kernel.

Different values of A are examined up to k=32 and the corresponding super As are listed in Table 2. For large k the probability function p^A becomes continuous-like, except irregularities on the borders, which confirms previous findings [8]. The value within the flat regions is computed by 2^{-h} and for the corresponding distances it holds $2^{k-h}\binom{n}{b}$. As a rule of thumb, larger As offer more potential to increase the code's minimum Hamming distance. Furthermore, not all super As are prime numbers.

Acknowledgment

This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (Resilience Path) and by Nvidia through the GPU Center of Excellence (GCOE) at the Center for Information Services and High Performance Computing (ZIH), TU Dresden, where the K80 GPU cluster Taurus was used for all the computations.

References

[1] T. Kolditz, D. Habich, D. Kuvaiskii, *et al.*, "Needles in the haystack—tackling bit flips in lightweight compressed data," in *Data Management Technologies and Applications*, 2015.

[2] R. H. Morelos-Zaragoza, *The art of error correcting coding*. John Wiley & Sons, 2006.

[3] F. J. MacWilliams and N. J. A. Sloane, *The theory of error correcting codes*. 1977, vol. 16.

[4] P. Raab, S. Krämer, and J. Mottok, "Reliability of data processing and fault compensation in unreliable arithmetic processors," *Microprocessors and Microsystems*, vol. 40, pp. 102–112, 2016, ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2015.07.014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933115001131.

[5] P. Forin, "Vital coded microprocessor: Principles and application for various transit systems," *Proc. IFACGCCT*, 2014.

[6] F. P. Preparata, "A class of optimum nonlinear double-error-correcting codes," *Information and Control*, vol. 13, no. 4, pp. 378–400, 1968.

[7] U. Schiffel, "Hardware error detection using AN codes," PhD thesis, TU Dresden, 2011.

[8] M. Hoffmann, P. Ulbrich, C. Dietrich, *et al.*, "A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN Codes," in *HASE*, 2014.

[9] E. Guerrini, E. Orsini, and M. Sala, "Computing the distance distribution of systematic non-linear codes," *Journal of Algebra and Its Applications*, vol. 09, no. 02, pp. 241–256, 2010.

[10] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, 1950.

[11] D. Fiala, F. Mueller, C. Engelmann, *et al.*, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC12*.

[12] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of accelerated dram soft error rates measured at component and system level," in *IRPS*, 2008.

[13] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, 2012.

[14] B. Schroeder and G. A. Gibson, "A Large-scale Study of Failures in High performance-computing Systems," *Dependable and Secure Computing*, vol. 7, no. 4, 2010.

[15] C. Lemieux, "Monte Carlo and Quasi-Monte Carlo Sampling," in, Springer, Ed. 2009, ISBN: 978-1441926760.