ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 4, No. 2, 2019

DATA CORRECTION USING HAMMING CODING AND HASH FUNCTION AND ITS CUDA IMPLEMENTATION

Anatoliy Melnyk, Nazar Kozak

Lviv Polytechnic National University, 12, S. Bandery Str., Lviv, 79013, Ukraine
Authors' e-mail: aomelnyk@lp.edu.ua, nazar.kozak@mail.com

Submitted on 28.11.2019

Abstract: This article deals with the use of block code for the entire amount of data. A hash function is used to increase the number of errors that can be detected. The automatic parallelization of this code by using special means is considered.

Index Terms: GPGPU, Hamming code, hash function, automatic parallelization.

I. INTRODUCTION

The peculiarity of the block error-correction codes is that they are applied separately to the data blocks. Using graphical accelerator block code can be applied to the entire amount of data as a single unit. It will reduce the number of additional bits to error correcting.

II. MODIFYING THE HAMMING CODE USING A HASH FUNCTION

If we apply the Hamming code [1] to all the data at once, one bit can be corrected with just a few dozen of additional bits. Equality 1 shows how many maximum i-bits of information can be used when applying additional k-bits:

$$i_{\max} = 2^k - k - 1 \tag{1}$$

For example, 48 additional bits is enough to correct a single error of 281 trillion bits (32 Tbytes).

With the application of (Table 1). It will allow to detect almost any accidental change of data.

Table 1

The Hamming code and the hash function

	hash function	Hamming code
Changes in one bit of data	detection	detection and correction
Changes in a few bits of data	detection	–
Change in a large piece of data	detection*	–

* Accidentally changing data to obtain the same hash value is unlikely.

Given the presence of bitwise instructions in modern processors, it is possible to effectively apply the Hamming code to independent bit positions (Fig. 1). If 1 byte is used as an elementary part of data, then can be corrected up to 8 erroneous bits in different bit positions.

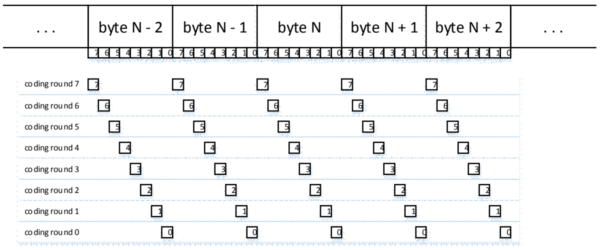


Fig. 1. Multiple Hamming code

When there are errors in the same bits of different bytes of data, the Hamming code for such bits will not work correctly and may indicate an error in a byte that does not actually have the errors. If we can detect the largest number of errors in a given byte, we can conditionally assume that the Hamming code worked correctly for these bits. This byte can now be considered as a central one and the value of adjacent bytes is matched to the hash function convergence with the expected value. If the assumption was false, it would not be possible to find a value for the hash function convergence, which would indicate an error that cannot be corrected (Fig. 2).

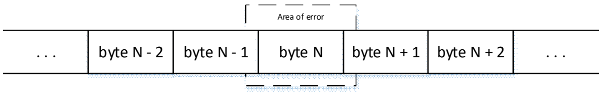


Fig. 2. Central error byte position

If the error bits that do not belong to the central byte are symmetrically relative to it, then it will be possible to correct a fragment of one and a half bytes (12 bits) according to the ratio:

$$N_c = \frac{3N_H}{2}, \tag{2}$$

where N_H – the number of Hamming encodings used. In any other case, even if these bits are located on one side of the central byte, it will be possible to correct one bit less (11 bits):

$$N_{c_{\min}} = \frac{3N_H}{2} - 1 \tag{3}$$

When applying more than 1-byte base fragment, for example 2-, 4-, or 8-byte fragment, the number of errors that can be corrected increases. Such encodings can also be implemented effectively on the 32-bit and 64-bit instruction set architecture of modern processors.

When evaluating the effectiveness of a error-correcting code, the indicator such as code rate is often used, which is the ratio of bits that directly encode data into the total number of code bits. This ratio for effective coding should be close to 1. Fig. 3 shows that this ratio is already approaching 1 for 8 kilobytes of information.

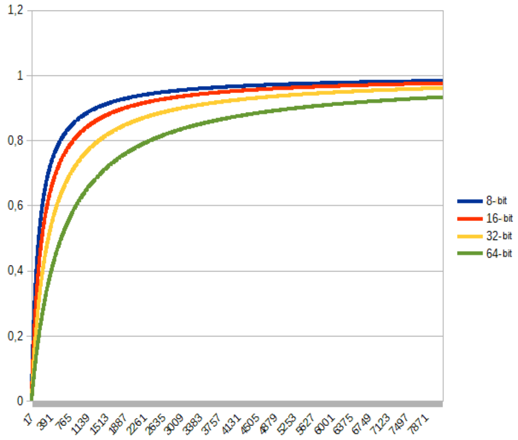
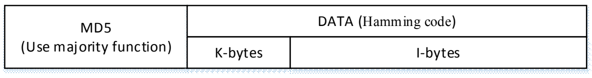


Fig. 3. Code efficiency

So, the proposed code is based on the following principles:

- all bits in a byte (or 2, 4 or 8 bytes) are encoded independently by the Hamming code;
- all messages are signed by the hash function;
- hash function is repeatedly written and read using the majority function;
- in the case of unsuccessful decoding, when the value of the hash function does not match the message received, an attempt is made to find the central fragment of the error and to select for it and the neighboring fragments of values that would allow to form the correct value of the hash function.

Then the proposed coding (Fig. 4) will contain the hash function and the data represented by the Hamming code. Since the hash value is much smaller, it can be represented by majority coding.



MD5 (Use majority function)	DATA (Hamming code)	
	K-bytes	I-bytes

Fig. 4. Data fields

The total amount of data will be equal to the size of the hash function 5 * 16 bytes = 80 bytes. The number of additional bits of data for the Hemming code is estimated at 64 bytes. Incremental error codes are used when using codes (Table 2).

In general, there are 5 levels of work with the proposed code (Table 3).

Also, code can be applied in partial modes, and full mode may not look for collisions. In addition, data bits can be used without any functions for error detection and correction, since these bits are presented unchanged. This may be relevant in cases where such code is generated by a stationary computer system and a low-performance embedded computer system acts as the recipient of the data.

Table 2

The number of Hamming code encodings in the proposed code

	The proposed code			
	The number of Hamming code encodings in the proposed code			
	8	16	32	64
MD5	80	80	80	80
K-bits of the Hamming code set (number of bytes)	64	128	256	512
The total amount of additional data	144	208	336	592
Detection and correction	11	23	47	95
Detection and possible correction	12	24	48	96
Detection	Detect any error			

Table 3

5 levels of work

Level	Usage
1	Error analysis is not applicable. The K and M bytes are not analyzed
2	Only Hamming code is used. M-bytes are not parsed
3	Hamming code correction and hash validation
4	Hamming code correction and hash validation. If the value of the hash does not match, it is assumed that the error occurred in the byte with the highest number of erroneous bits. This byte is matched by the value of the hash drop
5	Hamming code correction and hash validation. If the hash value does not match, all the bytes in which the wrong bits were detected using the Hamming code are erroneous. All erroneous bytes detected are matched to the hash function

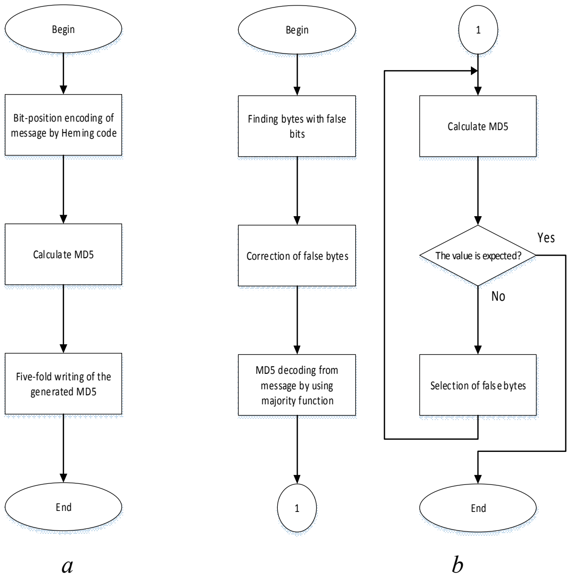

Begin
Bit-position encoding of message by Heming code
Calculate MD5
Five-fold writing of the generated MD5
End
a
Begin
Finding bytes with false bits
Correction of false bytes
MD5 decoding from message by using majority function
1
1
Calculate MD5
The value is expected?
Yes
No
Selection of false bytes
End
b

Fig. 5. Encoding and error-correction:
a – coding; b – correction

Fig. 5 shows a simple algorithm for encoding and error correction when applying the proposed code composition.

III. PARALLELIZATION

Some parts of the proposed algorithm can be parallelized. Firstly, the calculation of the control K-bits of the Hamming code can be parallelized (Fig. 6). Also, hash values (MD5) can be calculated in parallel when selecting values in bytes for various attempts to correct erroneous data.

$$\begin{aligned}
k_1 &= i_3 \oplus i_5 \oplus i_7 \oplus i_9 \oplus i_{11} \oplus i_{13} \oplus i_{15};\\
k_2 &= i_3 \oplus i_6 \oplus i_7 \oplus i_{10} \oplus i_{11} \oplus i_{14} \oplus i_{15};\\
k_4 &= i_5 \oplus i_6 \oplus i_7 \oplus i_{12} \oplus i_{13} \oplus i_{14} \oplus i_{15};\\
k_8 &= i_9 \oplus i_{10} \oplus i_{11} \oplus i_{12} \oplus i_{13} \oplus i_{14} \oplus i_{15};
\end{aligned}$$

$$\begin{array}{l|l|c|l|}
k_1 = & i_3 \oplus i_5 \oplus i_7 & \oplus & i_9 \oplus i_{11} \oplus i_{13} \oplus i_{15};\\
k_2 = & i_3 \oplus i_6 \oplus i_7 & \oplus & i_{10} \oplus i_{11} \oplus i_{14} \oplus i_{15};\\
k_4 = & i_5 \oplus i_6 \oplus i_7 & \oplus & i_{12} \oplus i_{13} \oplus i_{14} \oplus i_{15};\\
k_8 = & i_9 \oplus i_{10} \oplus i_{11} & \oplus & i_{12} \oplus i_{13} \oplus i_{14} \oplus i_{15};
\end{array}$$

Fig. 6. Parallel calculation of control bits for the Hamming code

IV. IMPLEMENTATION ON CUDA

From the very beginning of the development of GPGPU technology [3], CUDA technology [2] is mainly used to perform mathematical calculations [4, 5, 6, 7, 8]. Therefore, this technology is well-suited to error-correcting code.

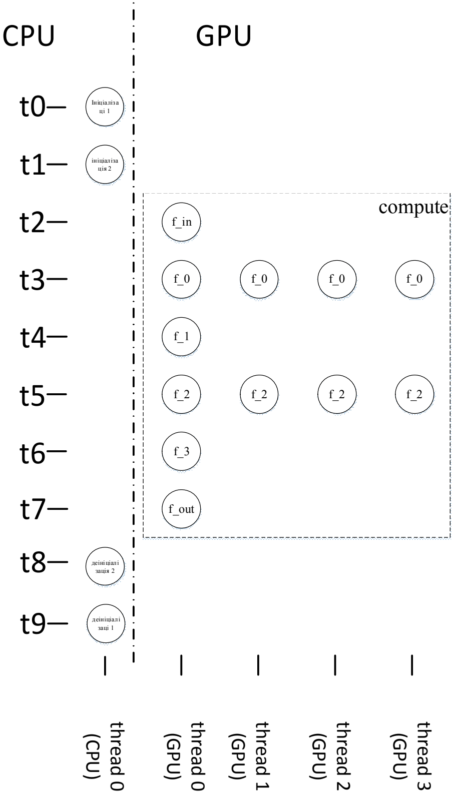


Fig. 7. The general model of execution

For effective use of CUDA technology, compute process must consist of many execution kernels [9]. Significant restrictions are imposed on the execution of the code of each such kernel [10, 11].

The process of decoding will have the structure which is shown in Fig. 7.

Synchronization points are provided for part of the CUDA code to execute directly on the GPU (Fig. 8).

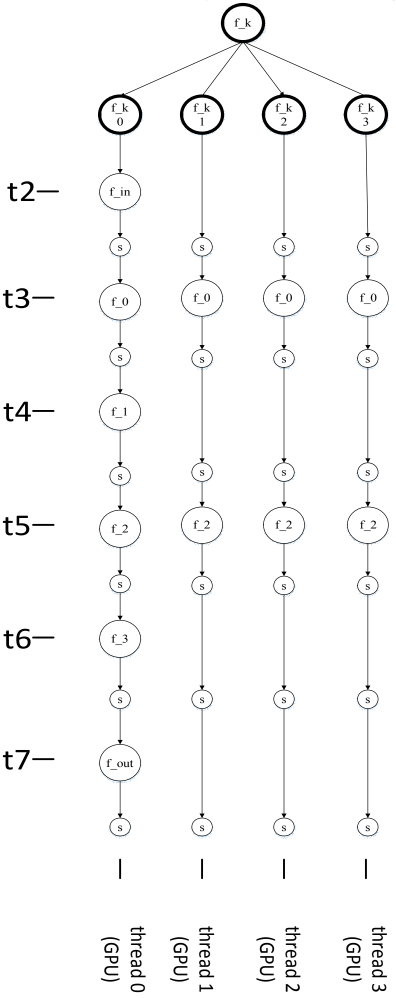


Fig. 8. The model of performance on the graphic accelerator

Each such stream of execution has access to several types of memory (Fig. 9).

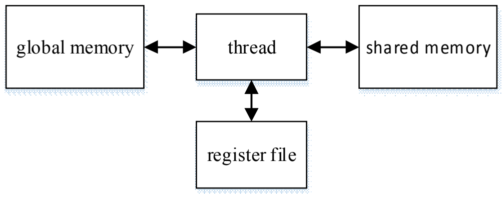


Fig. 9. The local context of the workflow for the system

The basic code execution constraints on the graphical accelerator will be imposed when working with shared memory [12]. The following macro was created for efficient operation.

Listing 1

```
#define GET_SHARED_INDEX(BASE, BASE_INDEX, PART_INDEX) \
 (BASE ? BASE - PART_INDEX * BLOCK_ALIGNED_COUNT - \
  BASE_INDEX / SHARED_CHANEL_COUNT * SHARED_CHANEL_COUNT + \
  BASE_INDEX % SHARED_CHANEL_COUNT : \
  PART_INDEX * BLOCK_ALIGNED_COUNT + BASE_INDEX)
```

A prototype of the CUDA execution kernels is listed in Listing 2.

Listing 2

```
DF_SPECIFY
DF_RETURN_TYPE f_out(
unsigned int * local,
unsigned int * shared,
unsigned int * secondShared,
unsigned int * data,
unsigned int sharedArgsStartIndex,
unsigned int sharedResultsStartIndex,
unsigned int stageIndex = 0,
unsigned int baseIndex = 0);
```

The initialization of the structure of a parallel program is formed as the following function:

Listing 3

```
RDF_SPECIFY
RDF_RETURN_TYPE
reInitParallelComputeModel2(DF_RETURN_TYPE(*(*parallelComputeModel)
[BLOCK_COUNT + 1])(unsigned int *, unsigned int *, unsigned int *,
unsigned int *, unsigned int, unsigned int, unsigned int, unsigned int)){
            parallelComputeModel[0][0] = f_in;
            parallelComputeModel[0][1] = NULL;
            unsigned int index = 0;
            for (; index < BLOCK_COUNT; ++index){
                        parallelComputeModel[1][index] = f0;
            }
            parallelComputeModel[1][index] = NULL;
            parallelComputeModel[2][0] = f1;
            parallelComputeModel[2][1] = NULL;
            index = 0;
            for (; index < BLOCK_COUNT; ++index){
                        parallelComputeModel[3][index] = f2;
            }
            parallelComputeModel[3][index] = NULL;
            parallelComputeModel[4][0] = f3;
            parallelComputeModel[4][1] = NULL;
            parallelComputeModel[5][0] = f_out;
            parallelComputeModel[5][1] = NULL;
            parallelComputeModel[6][0] = NULL;

            RDF_RETURN
}
```

The code for executing such kernels is given in Listing 4.

Listing 4

```
void cudaRunParallelComputeModel(
            DF_RETURN_TYPE(*(*parallelComputeModel)[BLOCK_COUNT +
1])(unsigned int *, unsigned int *, unsigned int *, unsigned int
*, unsigned int, unsigned int, unsigned int, unsigned int),
unsigned int * host_data) {

            unsigned int * data;
            cudaMalloc((void**)&data, MAJORITY_RANK * MD5_SIZE +
K_MAX_SIZE + DATA_SIZE * sizeof(unsigned char));

            begin(data);
            begin_device(host_data, data);

            for (unsigned int startStageIndex = 0; startStageIndex
< STAGE_COUNT + 1; ++startStageIndex){
                        for (unsigned int startIndex = 0; startIndex
< BLOCK_COUNT + 1; startIndex += CUDA_ITERATION_BLOCK_COUNT *
CUDA_BLOCK_SIZE){
                                    cudaRunParallelComputeModel__core
<<< CUDA_ITERATION_BLOCK_COUNT, CUDA_BLOCK_SIZE >>>
                                                (parallelComputeModel,
data, startStageIndex, startStageIndex + 1, startIndex,
startIndex + CUDA_ITERATION_BLOCK_COUNT * CUDA_BLOCK_SIZE);
                                    cudaDeviceSynchronize();
                        }
            }

            end_device(host_data, data); // TODO: move cudaFree
            end(host_data);
            cudaFree(data);
}
```

In non-hash mode, the CUDA acceleration results are shown in Table 4.

Table 4

Run time for 4MB data

	Run time for 4MB data (Hamming code only)
Sequential execution	≈2.192 s
Run on CUDA	≈0.296 s

In full mode, the CUDA acceleration results are shown in Table 5.

Table 5

Run time for data size 16kB

	Run time for data size 16kB
Sequential execution	≈19.832 s
Run on CUDA	≈1.211 s

V. CONCLUSION

The proposed data correction based on Hamming coding and hash function allows:

• to use a small number of additional bits;

• to detect any error;

• to correct er• to attempt to correct multiple errors in adjacent bytes;

rors in different bit positions;

• to use 5 different modes for systems with different performance.

The proposed data correction also shows:

• possibility of parallel execution;

• increased performance when using CUDA techology.

REFERENCES

[1] History of Hamming Codes. Archived from the original on 2007-10-25. Retrieved 2008-04-03.

[2] NVIDIA CUDA Programming Guide, Version 2.1, 2008

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In Proc. International Conference on Supercomputing, 2008.

[4] N. Fujimoto. Fast Matrix-Vector Multiplication on GeForce 8800 GTX. In Proc. IEEE International Parallel & Distributed Processing Symposium, 2008

[5] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In Proc. Supercomputing, 2008.

[6] G. Ruetsch and P. Micikevicius. Optimize matrix transpose in CUDA. NVIDIA, 2009.

[7] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-lite: Reducing GPU programming Complexity, In Proc. Workshops on Languages and Compilers for Parallel Computing, 2008

[8] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proc. Supercomputing, 2008.

[9] J. A. Stratton, S. S. Stone, and W. W. Hwu. MCUDA:An efficient implementation of CUDA kernels on multicores. IMPACT Technical Report IMPACT-08-01, UIUC, Feb. 2008.

[10] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Optimization space pruning for a multithreaded GPU. In Proc. International Symposium on Code Generation and Optimization, 2008.

[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.

[12] S. Hong and H. Kim. An analytical model for GPU architecture with memory-level and thread-level parallelism awareness. In Proc. International Symposium on Computer Architecture, 2009.

Anatoliy Melnyk has been a Head of Computer Engineering Department at Lviv Polytechnic National University since 1994. He graduated from Lviv Polytechnic Institute with the Engineer Degree in Computer Engineering in 1978. In 1985 he obtained his Ph.D in Computer Systems at Moscow Power Engineering Institute. In 1992, he received his D.Sc. degree at the Institute of Modelling Problems in Power Engineering of the National Academy of Science of Ukraine. He was recognized for his outstanding contributions into high-performance computer systems design as a Fellow Scientific Researcher in 1988. He became a Professor of Computer Engineering in 1996. From 1982 to 1994 he was a Head of Department of Signal Processing Systems at Lviv Radio Engineering Research Institute. From 1994 to 2008 he was a Scientific Director of the Institute of Measurement and Computer Technique at Lviv Polytechnic National University. From 1999 to 2009 he was a Dean of the Department of Computer and Information Technologies at the Institute of Business and Perspective Technologies, Lviv, Ukraine. Since 2000 he has served as a President and CEO of Intron ltd. He has also been a professor at Kielce University of Technology, University of Information Technology and Management, Rzeszow, University of Bielsko-Biala, John Paul II Catholic University of Lublin.

Nazar Kozak was born in 1985 in Ukraine. He received the B.S. and the M.S. degrees in computer engineering at Lviv Polytechnic National University in 2007 and 2008. He has been doing scientific and research work since 2008. His work resulted in 13 publications. Currently, he is an assistant proessor at the Computer Engieering Deparment, Lviv Polytechnic National University.