J Inf Process Syst, Vol.16, No.6, pp.1359~1371, December 2020
https://doi.org/10.3745/JIPS.01.0060

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

# GPU-Based ECC Decode Unit for Efficient Massive Data Reception Acceleration

Jisu Kwon[*], Moon Gi Seok[**], and Daejin Park[*,***]

**Abstract**
In transmitting and receiving such a large amount of data, reliable data communication is crucial for normal operation of a device and to prevent abnormal operations caused by errors. Therefore, in this paper, it is assumed that an error correction code (ECC) that can detect and correct errors by itself is used in an environment where massive data is sequentially received. Because an embedded system has limited resources, such as a low-performance processor or a small memory, it requires efficient operation of applications. In this paper, we propose using an accelerated ECC-decoding technique with a graphics processing unit (GPU) built into the embedded system when receiving a large amount of data. In the matrix–vector multiplication that forms the Hamming code used as a function of the ECC operation, the matrix is expressed in compressed sparse row (CSR) format, and a sparse matrix–vector product is used. The multiplication operation is performed in the kernel of the GPU, and we also accelerate the Hamming code computation so that the ECC operation can be performed in parallel. The proposed technique is implemented with CUDA on a GPU-embedded target board, NVIDIA Jetson TX2, and compared with execution time of the CPU.

**Keywords**
Embedded System, Error Correction Code, GPU-Based Acceleration, Hamming Code, Sparse Matrix–Vector Multiplication

## 1. Introduction

Recently, with the expansion of domains where neural networks are used [1–4], the data used for learning networks has become much larger in size because of visual data, such as an image or a video [5,6]. However, in the case of an Internet of Things (IoT) network, which is composed of a multiple-edge device and a cloud with edges that are connected, an embedded system needs a more efficient technique to handle massive data that communicates with the edges and the server with high speed, because an embedded system has limited resources and constraints in comparison to a cloud, which possesses enough computation capacitance [7,8].

When an edge device receives massive data from the cloud, if the device is in an environment with high error probability, such as wireless communication, data becomes very vulnerable to external noise [9–14]. Therefore, a system of receiving data needs the ability to detect and correct errors from the received data by itself.

Manuscript received March 19, 2020; first revision June 2, 2020; accepted June 9, 2020.
**Corresponding Author:** Daejin Park (boltanut@knu.ac.kr)
* School of Electronic and Electrical Engineering, Kyungpook National University, Korea (kjisu96@knu.ac.kr, boltanut@knu.ac.kr)
** School of Computer Science and Engineering, Nanyang Technological University, Singapore (moongi.seok@ntu.edu.sg)
*** School of Electronics Engineering, Kyungpook National University, Korea (boltanut@knu.ac.kr)

In this paper, we assumed that applying the error correction code (ECC)-decoding operation to the received data is necessary due to the high probability of error occurrence where massive data is sequentially transmitted. To use limited resources efficiently under a graphics processing unit (GPU)-based embedded environment, we accelerate the ECC-decoding operation using GPU and decrease the execution time to compare with the conventional method. Among the various codes used for ECC functions, we decided the Hamming code was a target of the experiment. Because the decoding operation of the Hamming code consists of matrix-vector multiplications, it is easy to deploy to cores of the GPU. We expressed the matrix–vector multiplication during the Hamming code decoding process as a compressed sparse row (CSR) format, one of the data structures representing a sparse matrix. Through the CSR format matrix multiplying directly with a codeword vector, we were able to derive a syndrome vector and execute the ECC-decoding operation much more efficiently. Further, we made the multiplication of the parity-check matrix and the codeword vector to derive a syndrome vector from the Hamming code decoding process run in parallel on the GPU's kernel.

The proposed technique was implemented on the NVIDIA Jetson TX2 module-based target board under a GPU-embedded system environment, and we analyzed the time cost for the Hamming code decoding operation. As a result, we found an optimal point between the amount of data that is an object of the ECC operation and execution time.

This paper is organized as follows. Section 2 introduces related works and the contribution of this paper, and Section 3 introduces the proposed whole system. In Section 4, we run the proposed technique on the actual target board and analyze it from the perspective of execution time, and we conclude in Section 5.

## 2. Related Works

There are several ECC operations to protect data from errors, such as low density parity check (LDPC) code [15], Bose–Chaudhuri–Hocquenghem (BCH) code [16], and Turbo code [17]. In the case of complex ECC methods, such as LDPC code and BCH code, research about computation acceleration using GPUs was conducted [18–24]. However, in the case of a Hamming code, which is a relatively simple ECC algorithm, acceleration using a GPU has not been attempted. Also, an ECC is used to detect and correct errors from data stored in memory, such as DRAM, at the on-chip level [25–30]. These studies focused on handling the ECC operation with hardware for a small amount of data. Unlike in previous studies, we approached the ECC operation from the viewpoint of massive data and a simple ECC algorithm. When data encoded by a Hamming code is transmitted, the data receiving module executes the decoding process by multiplication with a pre-stored parity-check matrix and a codeword vector. Compared to the sequential decoding process of data streaming using a single CPU, the proposed method executes the decoding computation of data stored on GPU memory in parallel.

## 3. Architecture

### 3.1 Proposed Technique

In this paper, we aim to accelerate a process for detecting and correcting errors using a Hamming code in the environment of massive sequentially transmitted data. Massive data, which is assumed in this paper, is collected on a buffer of the receiver system before going through the ECC-decoding process, and then all the data undergoes a decoding process at once when the collection is finished. The decoding process of the Hamming code finds a syndrome vector by multiplication of the parity-check matrix and the transmitted codeword vector and corrects errors according to the value of the syndrome vector.

Fig. 1 shows the process whereby the ECC decoding is operated in parallel. When a large amount of data is transmitted in a transmit module (TX), an error may occur in a part of the data due to an environment in which an external error is high. Therefore, it is necessary to receive codeword data from a receive module (RX) and undergo an ECC-decoding operation. However, in the conventional method, codewords sequentially received are immediately input into the ECC-decoding unit. The proposed technique waits until all codewords are received and collects them in the buffer. When all data is transferred, the data is copied to the built-in GPU. An ECC-decoding operation is performed in the GPU. At this time, decoding operations are arranged for each core embedded in the GPU to be performed in parallel. The details of the operations performed in the core are the multiplications between the parity-check matrix and codeword vector, and in the proposed technique, the parity-check matrix is regarded as a sparse matrix and converted to a multiplication between the vector and the vector in CSR format. After the ECC-decoding operation, syndrome vector data is copied back to the CPU and used to make reliable data.

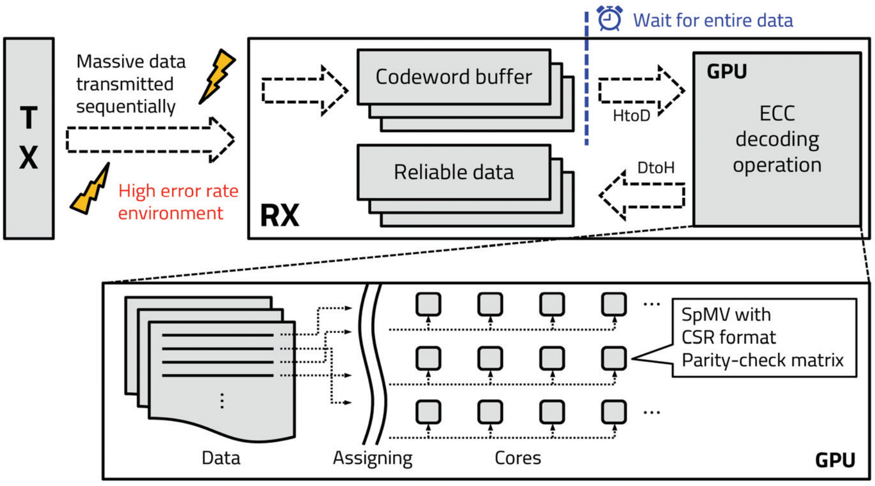


**Fig. 1.** Proposed GPU-based ECC-decoding process architecture.

## 3.2 Hamming Code

We chose the Hamming code as a function used in the ECC. The Hamming code makes error detection and correction available by inserting several parity bits into the data. When the number of data bits is $\boldsymbol{k}$ and the additional parity bit is $\boldsymbol{p}$, the number of generated codeword $\boldsymbol{n}$ can be found, as in Eq. (1).

$$2^p - 1 \geq k + p = n \tag{1}$$

The Hamming code encoding process can be regarded as generating parity bits. A parity bit can be obtained through an XOR operation between each data bit. The locations of the parity bits are marked in Fig. 2, which is composed of a power of 2.

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded data bits | | **p1** | **p2** | d1 | p4 | d2 | d3 | d4 | **p8** | d5 | d6 | d7 | d8 | d9 | d10 | d11 | **p16** | d12 |
| Parity Bit coverage | **p1** | O | | O | | O | | O | | O | | O | | O | | O | | O |
| | **p2** | | O | O | | | O | O | | | O | O | | | O | O | | |
| | **p4** | | | | O | O | O | O | | | | | O | O | O | O | | |
| | **p8** | | | | | | | | O | O | O | O | O | O | O | O | | |
| | **p16** | | | | | | | | | | | | | | | | O | O |

**Fig. 2.** Locations of data bits for generating parity bits.

We first need to find the syndrome vector to decode the Hamming code. The syndrome vector $\vec{\boldsymbol{S}}$ can be derived as below Eq. (2), which is a *modulo* (%) 2 of multiplication of the parity-check matrix $\boldsymbol{H}$ and the codeword vector $\vec{\boldsymbol{c}}$.

$$\vec{S} = (H \cdot \vec{c})\ \%\ 2 \tag{2}$$

If the number of a codeword bits and a data bits is the same, the parity-check matrix gets fixed elements. Also, if there is no error during the transmission, all the elements of the syndrome vector become zero. In contrast, if error $\vec{\boldsymbol{e}}$ is added to the codeword vector, as in Eq. (3), the elements of the syndrome vector cannot be zero, and we can confirm the presence or absence of error.

$$\vec{S} = \big(H \cdot (\vec{c} + \vec{e})\big)\ \%\ 2 \tag{3}$$

Likewise, according to the maximum error correction capacity by the minimum Hamming distance, if the syndrome vector value has a one-to-one correspondence with a specific error location, we can find the position of the error occurrence. As the location of the parity bits is a power of 2, the value converted from the binary string of the syndrome vector to the decimal is the location of the error.

This paper focused on the decoding process of a Hamming code. During the syndrome vector calculation, if the speed of the parity-check matrix and codeword vector multiplication was to be higher, the whole ECC-decoding operation could also be done at high speed.

## 3.3 Sparse Matrix–Vector Multiplication

In this paper, we applied a CSR-formatted sparse matrix–vector multiplication (SpMV) to the Hamming code decoding operation. Sparse matrix is an expression that indicates when a matrix value is mostly zero. Eq. (4) is the parity-check matrix $\boldsymbol{H}(15, 11)$ of the Hamming code.

$$H = \begin{bmatrix} 111000111100100 \\ 100110101011100 \\ 010101100110110 \\ 001011001110101 \end{bmatrix} \tag{4}$$

The format of the matrix above is close to the sparse matrix from the ratio occupied by zero-value elements. Fig. 3 shows the process that represents these sparse matrices conversion to the CSR format.

If a matrix is represented as a CSR format, one matrix transforms into three vectors. The components that make up the sparse matrix are vector $\boldsymbol{val}$, which has a value of each element; vector $\boldsymbol{col}$, which has a column value of each element; and vector $\boldsymbol{ptr}$, which has the number of elements in each row. If the size of the sparse matrix is $\boldsymbol{M}$ by $\boldsymbol{N}$, the sizes of the vectors are as indicated in Eq. (5).

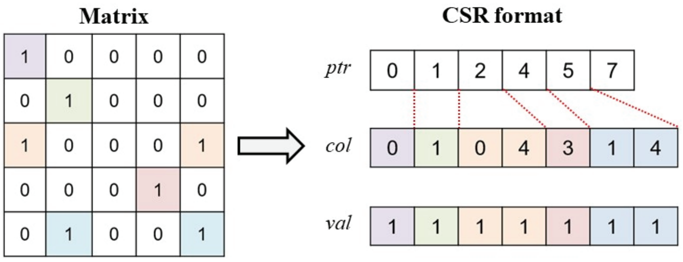


**Fig. 3.** Overview of matrix conversion to a CSR format.

$$\begin{cases} ptr = M + 1 \\ col = NNZ \\ val = NNZ \end{cases} \tag{5}$$

where, NNZ is number of non-zero elements.

The first element of the $\boldsymbol{ptr}$ vector is zero. From the second element of the $\boldsymbol{ptr}$ vector, the number of non-zero elements of each row is stored in order as a cumulative sum. As the CSR is a format that rearranges horizontally, in order of rows, the information is separated in units of rows from the vector. The SpMV in the CSR format has an advantage in that it only needs vectors to calculate it and does not need to change to form the matrix again. Algorithm 1 shows a conventional matrix–vector multiplication method.

**Algorithm 1.** Matrix–vector multiplication

```
Input:  $(\boldsymbol{M} \times \boldsymbol{N})$ matrix $\boldsymbol{A}$
        $(\boldsymbol{1} \times \boldsymbol{N})$ vector $\boldsymbol{x}$
Output: $(\boldsymbol{1} \times \boldsymbol{M})$ vector $\boldsymbol{y}$

for each row $\boldsymbol{i}$ from $\boldsymbol{1}$ to $\boldsymbol{M}$
    $\boldsymbol{sum = 0}$
    for each column $\boldsymbol{j}$ from $\boldsymbol{1}$ to $\boldsymbol{N}$
        $\boldsymbol{sum = sum + A_{ij} * x_j}$
    end for
    $\boldsymbol{y_i = sum}$
    $\boldsymbol{i = i + 1}$
end for
```

Eq. (6) gets vector $\vec{\boldsymbol{y}}$ by multiplying matrix $\boldsymbol{A}$, size $\boldsymbol{M}$ by $\boldsymbol{N}$, and vector $\vec{\boldsymbol{x}}$, size 1 by $\boldsymbol{M}$.

$$\vec{y} = A\vec{x} \tag{6}$$

The elements from $\boldsymbol{1}$ to $\boldsymbol{N}$ of the $\boldsymbol{i^{th}}$ row of matrix $\boldsymbol{A}$ are multiplied by one-to-one correspondence with elements from $\boldsymbol{1}$ to $\boldsymbol{N}$ of vector $\vec{\boldsymbol{x}}$ and are added cumulatively to the $\boldsymbol{sum}$. When the cumulative addition of the $\boldsymbol{i^{th}}$ row is completed, the calculation of one row is completed by storing the value in the $\boldsymbol{i^{th}}$ element of the output vector. If one is calculating vector $\boldsymbol{y}$ in the same way as stated above, multiplication and addition are executed $\boldsymbol{MN}$ times and $(\boldsymbol{M-1})\boldsymbol{N}$ times, respectively.

On the other hand, Algorithm 2 shows the SpMV using a CSR format.

**Algorithm 2.** CSR-formatted sparse matrix–vector multiplication

```
Input:  $(\mathbf{1} \times (\boldsymbol{M} + \mathbf{1})$ vector $\boldsymbol{ptr}$
        $(\mathbf{1} \times \boldsymbol{NNZ})$ vector $\boldsymbol{col}$
        $(\mathbf{1} \times \boldsymbol{NNZ})$ vector $\boldsymbol{val}$
        $(\mathbf{1} \times \boldsymbol{M})$ vector $\boldsymbol{x}$
Output: $(\mathbf{1} \times \boldsymbol{M})$ vector $\boldsymbol{y}$

for each row $\boldsymbol{i}$ from $\mathbf{1}$ to $\boldsymbol{M}$
    $\boldsymbol{sum} = \mathbf{0}$
    for each $\boldsymbol{k}$ from $\boldsymbol{ptr_i}$ to $\boldsymbol{ptr_{i+1}}$
        $\boldsymbol{sum} = \boldsymbol{sum} + \boldsymbol{val_k} * \boldsymbol{x_{col_k}}$
    end for
    $\boldsymbol{y_i} = \boldsymbol{sum}$
    $\boldsymbol{i} = \boldsymbol{i} + \mathbf{1}$
end for
```

Matrix $\boldsymbol{A}$ is converted to three vectors $\boldsymbol{ptr}$, $\boldsymbol{col}$, and $\boldsymbol{var}$ and their sizes are the same with Eq. (5). When input vector $\boldsymbol{x}$ and output vector $\boldsymbol{y}$ are the same as in Algorithm 1, the $\boldsymbol{y_i}$ value is the cumulative sum of the multiplications of $\boldsymbol{val_k}$ and $\boldsymbol{x_{col_k}}$ according to $\boldsymbol{k}$. $\boldsymbol{x_{col_k}}$ finds an element location of vector $\boldsymbol{x}$ that will be multiplied with the existing matrix value $\boldsymbol{val}$ by denoting vector $\boldsymbol{col}$ once.

## 3.4 Computation Parallelizing using GPU

When performing a matrix–vector multiplication operation on massive amounts of data at a time using a sparse-matrix format CSR, this paper proposed accelerating the execution time by parallelizing the use of the GPU, which has many cores. Massive data goes through the parallelized ECC operation on a GPU's core.

We used CUDA [31] to accelerate the Hamming code decoding process on the GPU. In this paper, we assumed that a transmission module will send data sequentially when a receiver needs massive data. The conventional method performed the matrix–vector multiplication operation during the Hamming code decoding process using single instruction, multiple data (SIMD), which is supported by the CPU for received data. However, this paper does not immediately apply the Hamming code decoding operation to receive data, collect data of sufficient size on the buffer first, and perform massive data computation in parallel with the GPU at once. As a drawback of using a GPU, memory overhead can occur from data interchanging between a host (CPU) and a device (GPU). In the case of this paper, we again regarded the following tasks as overhead: copying codeword data from the host (CPU) to the device (GPU) and copying the ECC operation's finished syndrome vector data from the device (GPU) to the host (CPU). We stacked data in parallel to reduce execution time during the ECC operation to improve overhead.

When implementing CSR-formatted SpMV using a CUDA kernel, we distributed computation in a scalar way. Fig. 4 shows a computation arrangement according to GPUs' blocks and thread configurations. The kernel function required to use CUDA on the GPU is a parallel flow running on the device and is represented by each thread in Fig. 4. The batch of these threads is called a block. Inside a block, there are a certain number of threads per device. Threads in the same block may sync using memory that can be shared with each other or memory that only one thread can use. Since every thread has its own unique ID, when writing a CUDA program, we can decide which program will run in the thread. Fig. 5 shows a memory hierarchy GPU device. There are several levels of the GPU device's memory hierarchy. There are several threads in each block that perform operations in the GPU. Local memory is the first memory that can be accessed by threads. It is a private memory that only one thread can access. In addition, there is shared memory among threads that existing in one block. To share memory with threads in different blocks, we can use global memory at the highest level.

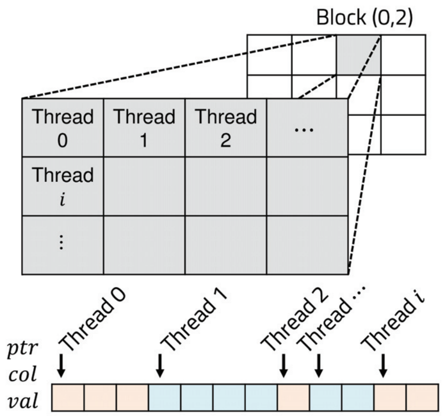


**Fig. 4.** Operation assignment based on GPU thread configuration.

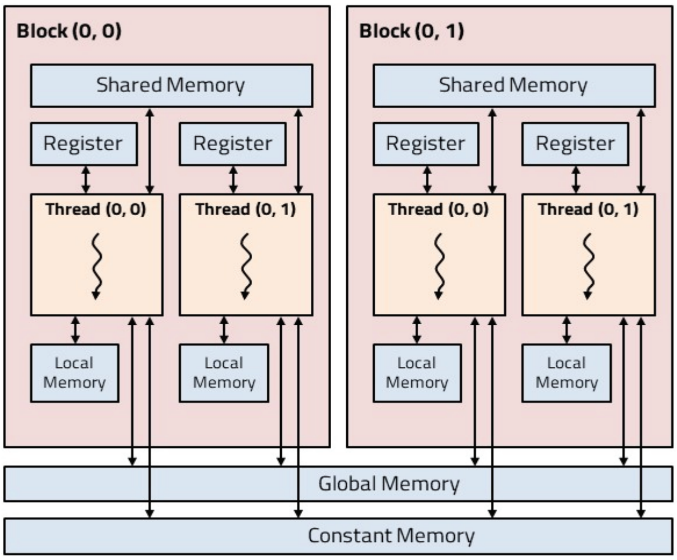


**Fig. 5.** CUDA memory hierarchy.

The kernel of the GPU is configured in such a way that several threads form a block and the blocks form a grid. As shown in Fig. 3, the CSR format has data stored in a vector composed of a unit of rows. Therefore, each computation corresponding to a row is assigned to one thread, and when it is expanded to large data, threads are assigned according to the number of rows.

## 4. Experimental Results

To implement the proposed GPU-based massive data in the ECC-decoding unit, we used NVIDIA Jetson TX2, which has specifications in Table 1, as a target board. Table 1 shows the specifications of the GPU, the CPU, and the target board memory. Considering that most embedded systems or IoT devices are constructed on the basis of the ARM architecture, a target board using an ARM CPU was selected. According to the specification of the target board, the Pascal architecture GPU was embedded. Based on these specifications, the target board could be regarded as an embedded system with a built-in GPU. Also, the target board operates on Ubuntu 16.04 Linux OS, runs the proposed algorithm, and analyzes the algorithm from the execution time viewpoint. Fig. 6 shows the NVIDIA Jetson TX2 that we used as the target board in the actual experiment.

**Table 1.** Specifications of NVIDIA Jetson TX2

| | Specifications |
|---|---|
| GPU | 256-core Pascal |
| CPU | Quad-core ARM Cortex-A57<br>Dual-core NVIDIA Denver 2 |
| Memory | 8GB 128-bit LPDDR4<br>59.7GB/s |

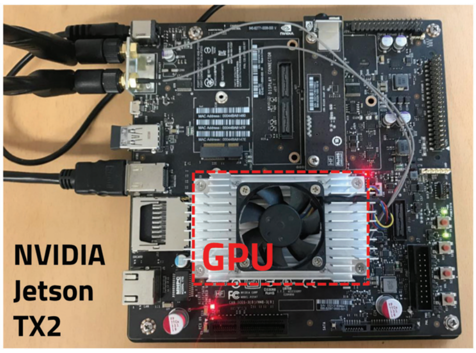


**Fig. 6.** Target board (NVIDIA Jetson TX2).

We conducted the experiment with a comparison of the ECC-decoding execution time between using a CPU and using a GPU. In the case of using a CPU, we measured two cases. In the first case, we measured the execution time of ECC-decoding operations performed sequentially at the software level in C language. In the second case, we implemented parallelism for the ECC-decoding operation at the CPU level using SIMD instructions of NEON [32] vector functions by ARM, which is the core of the Jetson TX2 board. Next, in the case of using a GPU, in the kernel function implemented by CUDA, we arranged the operations required for the ECC-decoding operation process in each core, and they could be performed in parallel. Fig. 7 shows a comparison of the measured execution time of each ECC decoding operation when we used the CPU and the GPU while changing the data size. In the measurement, it is assumed that the sizes of data targeted for the ECC-decoding operation are 1 MB, 5 MB, 10 MB, and 100 MB. Program execution time was measured using the *gettimeofday* function provided by *sys/time.h*. Table 2 shows the execution time of the CPU and GPU according to the change in data size.

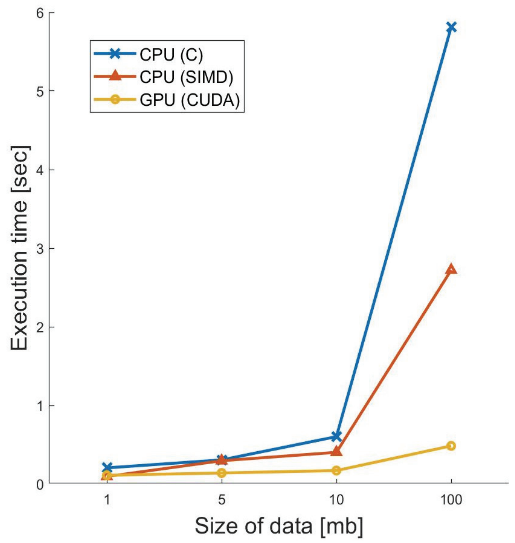


**Fig. 7.** Execution time of ECC-decoding operation according to size of data.

**Table 2.** Hamming code decoding execution time measurement results according to data size

| | **Execution time (s)** | | | |
|---|---|---|---|---|
| | **1 MB** | **5 MB** | **10 MB** | **100 MB** |
| CPU (C) | 0.205 | 0.3064 | 0.6025 | 5.8128 |
| CPU (SIMD) | 0.0951 | 0.296 | 0.4029 | 2.7215 |
| GPU (CUDA) | 0.1114 | 0.1401 | 0.1708 | 0.4849 |

When the size of the data is small, the execution time on the GPU takes longer than that on the CPU due to the overhead of copying data to the device (GPU) and copying back to the host (CPU). However, as the size of the data increases, performing decode operations on the GPU is more beneficial in terms of execution time. When the size of the data targeted for the ECC-decoding operation is larger, the difference in execution time measurement is larger when the CPU is used than when the GPU is used.

For example, the data was set at 100 MB, and we measured the execution time as 0.48 seconds when using the GPU (CUDA), 2.72 seconds when using the CPU (with SIMD), and 5.81 seconds when using the CPU (with C). As a result, in the case of using the GPU, the execution time decreased by 82.35% and 91.74% compared to in the other two cases.

As the size of the data becomes larger, the required time for copying data between the host (CPU) and the device (GPU) also becomes larger. Across the entire execution time, Fig. 8 shows the proportion of time spent on copying data to the device (GPU), copying data to the host (CPU), and activating the CUDA kernel function. As the amount of data increases, we can see a graph in which the proportion of time spent copying data to a device or host increases. In contrast, the execution time ratio for the CUDA kernel function is growing smaller. From these measurement results, we confirmed that the part acting as a bottleneck is a process of copying data from the host (CPU) to the device (GPU).

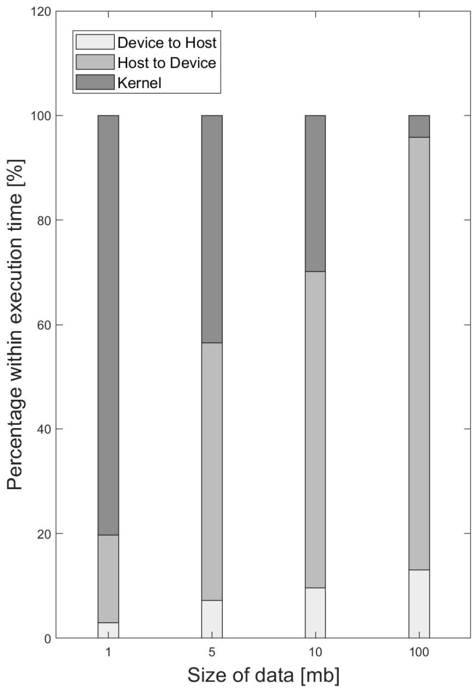


**Fig. 8.** Percentage of execution time configuration according to size of data.

## 5. Conclusion

This paper achieved accelerated computation of ECC Hamming code decoding under a GPU-based embedded system environment by parallelizing SpMV for massive data using a GPU. By performing SpMV on a row-wise basis in the kernel function of the GPU implemented through CUDA, we obtained the result of time reduction taken in the overall process despite the memory exchange overhead between the host and the device. We validated the proposed technique on the target board, NVIDIA Jetson TX2, and the result shows that execution time is much smaller when the Hamming code decoding operation was performed with the GPU rather than with the CPU. From the results, we confirmed that the proposed technique is a structure that can be used to quickly detect and correct errors from massive data in GPU-based embedded systems. The status of this evaluation is still lacking on providing power consumption analysis. In the future, we will expand this study to the viewpoint of power consumption to compare with the existing method.

## Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. NRF-2019R1A2C2005099), and Ministry of Education (No. NRF-2018R1A6A1A03025109), and BK21 Four project funded by the Ministry of Education, Korea (September 2020).

## References

[1] S. Maity, M. Abdel-Mottaleb, and S. S. Asfour, “Multimodal biometrics recognition from facial video with missing modalities using deep learning,” *Journal of Information Processing Systems*, vol. 16, no. 1, pp. 6-29, 2020.

[2] A. Gorodilov, D. Gavrilov, and D. Schelkunov, “Neural networks for image and video compression,” in *Proceedings of 2018 International Conference on Artificial Intelligence Applications and Innovations (IC-AIAI)*, Nicosia, Cyprus, 2018m pp. 37-41.

[3] H. Zeng, Q. Wang, C. Li, and W. Song, “Learning-based multiple pooling fusion in multi-view convolutional neural network for 3D model classification and retrieval,” *Journal of Information Processing Systems*, vol. 15, no. 5, pp. 1179-1191, 2019.

[4] M. J. J. Ghrabat, G. Ma, I. Y. Maolood, S. S. Alresheedi, and Z. A. Abduljabbar, “An effective image retrieval based on optimized genetic algorithm utilized a novel SVM-based convolutional neural network classifier,” *Human-centric Computing and Information Sciences*, vol. 9, article no. 31, 2019.

[5] V. S. Chua, J. Z. Esquivel, A. S. Paul, T. Techathamnukool, C. F. Fajardo, N. Jain, O. Tickoo, and R. Iyer, “Visual IoT: ultra-low-power processing architectures and implications,” *IEEE Micro*, vol. 37, no. 6, pp. 52-61, 2017.

[6] E. Khan, S. Lehmann, H. Gunji, and M. Ghanbari, “Iterative error detection and correction of H.263 coded video for wireless networks,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 12, pp. 1294-1307, 2004.

[7] J. Kwon, M. G. Seok and D. Park, “User Insensible Sliding Firmware Update Technique for Flash-Area/Time-Cost Reduction toward Low-Power Embedded Software Replacement,” *2020 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, Kokubunji, Japan, 2020, pp. 1-3, doi: 10.1109/COOLCHIPS49199.2020.9097638.

[8] J. Kwon and D. Park, “Implementation of Computation-Efficient Sensor Network for Kalman Filter-based Intelligent Position-Aware Application,” *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, Fukuoka, Japan, 2020, pp. 565-568, doi: 10.1109/ICAIIC48513.2020.9065098.

[9] M. R. Mesbahi, A. M. Rahmani, and M. Hosseinzadeh, “Reliability and high availability in cloud computing environments: a reference roadmap,” *Human-centric Computing and Information Sciences*, vol. 8, article no. 20, 2018.

[10] S. Unterschütz and V. Turau, “Fail-safe over-the-air programming and error recovery in wireless networks,” in *Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems*, Klagenfurt, Austria, 2012, pp. 27-32.

[11] Q. Zhang, G. Wang, Z. Xiong, J. Zhou, and W. Zhu, “Error robust scalable audio streaming over wireless IP networks,” *IEEE Transactions on Multimedia*, vol. 6, no. 6, pp. 897-909, 2004.

[12] M. Manohara, R. Mudumbai, J. Gibson, and U. Madhow, “Error correction scheme for uncompressed HD video over wireless,” in *Proceedings of 2009 IEEE International Conference on Multimedia and Expo*, New York, NY, 2009, pp. 802-805.

[13] P. Kukieattikool and N. Goertz, “Staircase codes for high-rate wireless transmission on burst-error channels,” *IEEE Wireless Communications Letters*, vol. 5, no. 2, pp. 128-131, 2016.

[14] Y. Qassim and M. E. Magana, “Error-tolerant non-binary error correction code for low power wireless sensor networks,” in *Proceedings of the International Conference on Information Networking (ICOIN)*, Phuket, Thailand, 2014, pp. 23-27.

[15] R. Motwani, Z. Kwok, and S. Nelson, “Low density parity check (LDPC) codes and the need for stronger ECC,” in *Proceedings of the 6th Annual Flash Memory Summit*, Santa Clara, CA, 2011, pp. 41-50.

[16] W. Liu, J. Rho, and W. Sung, “Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories,” in *Proceedings of 2006 IEEE Workshop on Signal Processing Systems Design and Implementation*, Banff, Canada, 2006, pp. 303-308.

[17] K. Sripimanwat, *Turbo Code Applications*. Dordrecht: Springer, 2005.

[18] S. Keskin and T. Kocak, “GPU accelerated gigabit level BCH and LDPC concatenated coding system,” in *Proceedings of 2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, 2017, pp. 1-4.

[19] A. K. Subbiah and T. Ogunfunmi, “Memory-efficient Error Correction Scheme for Flash Memories using GPU,” in *Proceedings of 2018 IEEE International Workshop on Signal Processing Systems (SiPS*), Cape Town, South Africa, 2018, pp. 118-122..

[20] A. K. Subbiah and T. Ogunfunmi, “Three-bit fast error corrector for BCH codes on GPUs,” in *Proceedings of 2019 IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, 2019, pp. 1-4.

[21] A. K. Subbiah and T. Ogunfunmi, “Fast BCH syndrome generator using parallel polynomial division algorithm for GPGPUs,” in *Proceedings of 2019 IEEE 6th International Conference on Industrial Engineering and Applications (ICIEA)*, Tokyo, Japan, 2019, pp. 458-462.

[22] S. Keskin and T. Kocak, “GPU-based gigabit LDPC decoder,” *IEEE Communications Letters*, vol. 21, no. 8, pp. 1703-1706, 2017.

[23] C. H. Chan and F. C. Lau, “Parallel decoding of LDPC convolutional codes using OpenMP and GPU,” in *Proceedings of 2012 IEEE Symposium on Computers and Communications (ISCC)*, Cappadocia, Turkey, 2012, pp. 225-227.

[24] H. Ahn, Y. Jin, S. Han, S. Choi, and S. Ahn, “Design and implementation of GPU-based turbo decoder with a minimal latency,” in *Proceedings of the 18th IEEE International Symposium on Consumer Electronics (ISCE)*, JeJu, South Korea, 2014, pp. 1-2.

[25] H. L. Kalter, C. H. Stapper, J. E. Barth, J. DiLorenzo, C. E. Drake, J. A. Fifield, G. A. Kelly, S. C. Lewis, W. B. van der Hoeven, and J. A. Yankosky, “A 50-ns 16-Mb DRAM with a 10-ns data rate and on-chip ECC,” *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1118-1128, 1990.

[26] T. Tanzawa, T. Tanaka, K. Takeuchi, R. Shirota, S. Aritome, H. Watanabe, et al., “A compact on-chip ECC for low cost flash memories,” *IEEE Journal of Solid-State Circuits*, vol. 32, no. 5, pp. 662-669, 1997.

[27] K. Dang and X. T. Tran, “Parity-based ECC and mechanism for detecting and correcting soft errors in on-chip communication,” in *Proceedings of 2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Hanoi, Vietnam, 2018, pp. 154-161.

[28] F. Alzahrani and T. Chen, “On-chip TEC-QED ECC for ultra-large, single-chip memory systems,” in *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, MA, 1994, pp. 132-137.

[29] S. H. Kim, W. O. Lee, J. H. Kim, S. S. Lee, S. T. Hwang, C. I. Kim, et al., “A low power and highly reliable 400Mbps mobile DDR SDRAM with on-chip distributed ECC,” in *Proceedings of 2007 IEEE Asian Solid-State Circuits Conference*, Jeju, South Korea, 2007, pp. 34-37.

[30] J. A. Fifield and C. H. Stapper, “High-speed on-chip ECC for synergistic fault-tolerance memory chips,” *IEEE Journal of Solid-State Circuits*, vol. 26, no. 10, pp. 1449-1452, 1991.

[31] NVIDIA, “CUDA Toolkit Documentation,” [Online]. Available: https://docs.nvidia.com/cuda/.

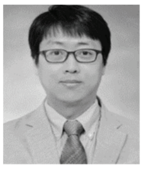
[32] ARM Developer, “Neon ISA Description,” [Online]. Available: https://developer.arm.com/architectures/ instruction-sets/ simd-isas/neon.

**Jisu Kwon** **https://orcid.org/0000-0002-0433-9533**

He received his B.S. degree in Electronics Engineering at Kyungpook National University, Daegu, Korea in 2019. He is currently an integrated PhD student in School of Electronic and Electrical Engineering at Kyungpook National University, Daegu, Korea. He is now pursuing toward his PhD degree in AI-Embedded System-on-Chip lab. His research interests include the behavior changeable neuromorphic deep learning processor based on partial software partial replacement and hardware reconfiguration architecture.

**Moon Gi Seok** **https://orcid.org/0000-0002-8159-9910**

He received the B.S. degree in electronics engineering from Korea University, Seoul, Korea in 2009, and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2011 and 2017, respectively. He was a post-doctoral researcher at KAIST and Arizona State University, in 2018 and 2019. He is currently a research fellow at Nanyang Technological University, Singapore. His research interest includes multi-level modeling, simulation, and verification of system-on-chip (SoC) designs, and high-performance simulation methodology in various domains.



**Daejin Park** **https://orcid.org/0000-0002-5560-873X**

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2001, the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2003, and 2014, respectively. He was a Research Engineer in SK Hynix Semiconductor, Samsung Electronics over 12 years from 2003 to 2014, respectively and have worked on designing low-power embedded processors architecture and implementing fully AI-integrated system-on-chip with intelligent embedded software on the custom-designed hardware accelerator, especially for hardware/software tightly-coupled applications, such as smart mobile devices, industrial electronics. He was nominated as one of Presidential Research Fellows 21, Republic of Korea in 2014. Prof. Park is now with School of Electronic and Electrical Engineering and School of Electronics Engineering as full-time assistant processor in Kyungpook National University, Daegu, Korea, since 2014. He has published over 170 technical papers and 37 patents.