A Parallel Algorithm for Enumerating Joint Weight of a Binary Linear Code in Network Coding

Shohei Ando, Fumihiko Ino, Toru Fujiwara, and Kenichi Hagihara
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka 565–0871, Japan
Email: {s-ando, ino, fujiwara, hagihara}@ist.osaka-u.ac.jp

***Abstract*—In this paper, we present a parallel algorithm for enumerating joint weight of a binary linear (n, k) code, aiming at accelerating assessment of its decoding error probability for network coding. To reduce the number of pairs of codewords to be investigated, our parallel algorithm reduces dimension k by focusing on the all-one vector included in many practical codes. We also employ a population count instruction to compute joint weight of codewords with a less number of instructions. Our algorithm is implemented on a multi-core CPU system and an NVIDIA GPU system using OpenMP and compute unified device architecture (CUDA), respectively. We apply our implementation to a subcode of a (127,22) BCH code to evaluate the impact of acceleration.**

I. INTRODUCTION

Network coding [1] is a technique for improving transmission efficiency of multicast communication. This technique allows relay nodes to apply coding arithmetic to incoming messages. Figure 1 shows an example of multicast communication using network coding over the butterfly network. In this example, the source node s transmits two messages x and y to sink nodes r_1 and r_2. On a typical network in Fig. 1(a), where relay nodes are prohibited to perform coding arithmetic, r_1 fails to receive y if v_2 transmits x rather than y. Similarly, r_2 fails to receive x if v_2 transmits y. In contrast, network coding increases multicast efficiency by allowing relay node v_2 to transmit $x \oplus y$ to the next node v_4, as shown in Fig. 1(b). The operator $\oplus$ here represents bitwise exclusive disjunction. The sink node r_1 then can extract y from its two incoming messages x and $x \oplus y$. Similarly, r_2 can receive both x and y. Li *et al.* [2] presented that the max-flow bound from the source node to each sink node can be achieved if relay nodes use a linear transformation as such coding arithmetic.

In practice, *an error-correcting code* [3] must be applied to flowing messages to achieve robust communication against noise. The performance of an error-correcting code C can be assessed by performance metrics such as the decoding error probability and error-correcting capability. An error here occurs if a transmitted message is decoded to a codeword $\mathbf{v} \in C$ that differs from the originally sent codeword $\mathbf{u} \in C$ ($\mathbf{u} \neq \mathbf{v}$). High-performance error correction can be achieved by not only maximizing the error-correcting capability but also minimizing the decoding error probability.

For a typical network, where relay nodes transmit incoming messages without applying coding arithmetic, *weight distribution* of C is useful to evaluate the decoding error probability. The weight distribution of C is denoted by an $(n + 1)$-tuple $(A_0, A_1, \ldots, A_n)$, where n is the length of C and A_i $(0 \le i \le n)$ is the number of codewords of Hamming weight i in C [4]. This performance metrics is also useful to assess the performance of codes for network coding. However, a single error occurred on a network link can affect decoding results of multiple messages. For example, the sink node r_1 in Fig. 1(b) can face with two incorrectly transmitted messages x and $x \oplus y$ if an error occurs on the link between s and v_1. In this case, errors in incoming messages are dependent. Therefore, the decoding error probability for m received codewords cannot be obtained from weight distribution of C. Instead of weight distribution, it requires *m-joint weight distribution* of C [5]. In this work, we deal with the problem of the butterfly network, so that we assume that $m = 2$ hereafter.

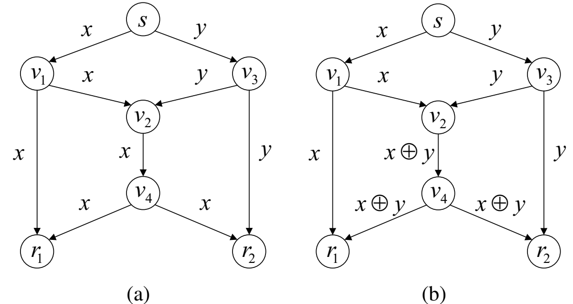

Fig. 1. An example of multicast communication using network coding over the butterfly network. (a) Node r_1 fails to receive y if node v_2 transmits x rather than y. (b) Network coding allows nodes r_1 (r_2) to receive both x and y, because y (x) can be extracted from x (y) and $x \oplus y$.

To the best of our knowledge, there is no formula that directly gives joint weight distribution of a code C except for Hamming code, simplex code, and the first order Reed-Muller code [3]. Consequently, a joint weight enumerator is needed to compute *a joint weight histogram*, which stores the occurrence of joint weight for all 2-tuples (i.e., pairs) $(\mathbf{u}, \mathbf{v})$ $(\mathbf{u}, \mathbf{v} \in C)$ of codewords in C. A histogram here is an estimate of the probability distribution of a variable, and consists of a sequence of *bins*, which store the frequency of observations over categories (i.e., intervals of a variable). Consider a binary linear (n, k) code [6] of length n and dimension k. The binary linear (n, k) code consists of 2^k

codewords. Its joint weight histogram can be computed in $\mathcal{O}(2^{2k}n)$ time, because there are 2^{2k} pairs of codewords of length n. This time complexity exponentially increases with k, so that the joint weight enumeration must be accelerated to assess codes with large dimension k.

In general, parallel-based solutions using multi-core CPUs or graphics processing units (GPUs) [7] are attractive methods for achieving acceleration for compute- and memory-intensive applications [8], [9], [10], [11]. Ando *et al.* [12] presented a parallel algorithm that enumerated joint weight on a multi-core CPU and a GPU. They exploited data parallelism in enumeration by assigning different pairs of codewords to threads. Their algorithm employed an efficient mutual exclusion mechanism, because multiple threads could simultaneously update the same bin of the histogram. Furthermore, joint weight was rapidly computed by a population count instruction. However, this parallel algorithm can be further accelerated by exploiting theoretical properties on code structure.

In this paper, we propose a parallel algorithm for enumerating joint weight of a binary linear (n, k) code. We extend the previous algorithm [12] by taking advantage of code structure to reduce the number of pairs of codewords to be investigated. We focus on the fact that many practical codes include the all-one vectors as a codeword. This assumption reduces the dimension k of the code for efficient enumeration. Similar to [12], our algorithm employ a population count instruction to rapidly compute joint weight of codewords. Our parallel algorithm currently runs on a multi-core CPU system and a compute unified device architecture (CUDA) compatible GPU system [13]. We assume that $n \leq 128$ and the target machine is equipped with an NVIDIA Kepler GPU [14].

The following paper is organized as follows. Section II introduces some related studies. Section III presents preliminaries on joint weight enumeration of a code. Section IV describes our parallel algorithm for enumerating joint weight of a binary linear (n, k) code. Section V presents experimental results. Finally, Section VI shows conclusion and future work.

II. Related Work

Ando *et al.* [12] implemented a parallel algorithm that enumerated joint weight of a binary linear (n, k) code on a multi-core CPU. Their CPU-based implementation exploited multiple CPU cores by OpenMP [15], which achieved multithreading by simply adding compiler directives to the serial code. Furthermore, single instruction multiple data (SIMD) instructions, called Streaming SIMD Extensions (SSE) [16], were used to maximize the performance per CPU core by processing a 128-bit vector data simultaneously. They also presented a GPU-based implementation that processed thousands of threads simultaneously. Their GPU-based implementation generated pairs of codewords such that threads could access different memory addresses simultaneously. In contrast to the architecture-specific optimization mentioned above, the present work focuses on code-oriented optimization that accelerate joint weight enumeration on arbitrary architectures.

Some theoretical results are useful to accelerate joint weight enumeration of a binary linear (n, k) code. The MacWilliams identity [3] is a famous theorem that relates the weight enumerator of a binary linear (n, k) code to that of its dual code, namely a binary linear $(n, n-k)$ code. According to this theorem, the time complexity of joint weight histogram computation can be reduced from $\mathcal{O}(2^{2k}n)$ to $\mathcal{O}(2^{2(n-k)}n)$ when $k > n-k$. Kido et al. [5] applied the MacWilliams identity to an m-joint weight enumerator from a theoretical point of view.

With respect to weight distribution, Desaki *et al.* [17] presented a weight enumeration algorithm that exploited code structure called trellis diagram. Although this algorithm cannot produce joint weight distribution of a code, their idea can be extended to joint weight enumeration algorithms to reduce the amount of work.

III. Preliminaries

Let $\mathbf{u} = (u_1 u_2 \cdots u_n) \in \mathbb{F}^n$ and $\mathbf{v} = (v_1 v_2 \cdots v_n) \in \mathbb{F}^n$ be vectors of length n, where $\mathbb{F}$ is a binary finite field and $u_r, v_r \in \mathbb{F}$ $(1 \leq r \leq n)$. Let $f_{pq}(\mathbf{u}, \mathbf{v})$ also be the number of r such that $u_r = p$ and $v_r = q$, where $p, q \in \mathbb{F}$. The joint weight $w(\mathbf{u}, \mathbf{v})$ of a pair $(\mathbf{u}, \mathbf{v})$ of vectors then is given by a 4-tuple

$$w(\mathbf{u}, \mathbf{v}) = (a, b, c, d), \tag{1}$$

where $a = f_{11}(\mathbf{u}, \mathbf{v})$, $b = f_{10}(\mathbf{u}, \mathbf{v})$, $c = f_{01}(\mathbf{u}, \mathbf{v})$, and $d = f_{00}(\mathbf{u}, \mathbf{v})$. For instance, we obtain $(a, b, c, d) = (2, 2, 3, 1)$ for $\mathbf{u} = (11110000)$ and $\mathbf{v} = (11001110)$. Since $d = n - a - b - c$ [3], we can omit the last element d from joint weight (a, b, c, d). Hereafter, we denote a joint weight by a 3-tuple (a, b, c) for simplicity.

Because a, b, and c are numbers enumerated from n elements, we have

$$0 \leq a, b, c \leq n, \tag{2}$$

$$0 \leq a + b + c \leq n. \tag{3}$$

Joint weight enumeration outputs a sequence of numbers, each corresponding to the frequency of a tuple (a, b, c) that satisfies Eqs. (2) and (3). Considering all combinations with repetitions, the number of possible tuples is given by $\binom{n+3}{4}$.

Let C be a binary linear code. Let $J_{a,b,c}$ be the number of pairs $(\mathbf{u}, \mathbf{v})$ $(\mathbf{u} \in C, \mathbf{v} \in C)$ of codewords that have joint weight (a, b, c). The joint weight distribution of C is then given by a $\binom{n+3}{4}$-tuple $(J_{0,0,0}, J_{0,0,1}, \ldots, J_{a,b,c}, \ldots, J_{n,0,0})$ such that a, b, and c satisfy Eqs. (2) and (3). This distribution can be stored in a joint weight histogram with $\binom{n+3}{4}$ bins.

Algorithm 1 shows a brute-force parallel algorithm that generates joint weight histogram J from the length n and dimension k of code C, and a sequence $W = (\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{2^k-1})$ of codewords in C. Parallelization can be easily achieved by assigning different pairs of codewords to threads. However, an atomic instruction [13] is needed to compute the histogram correctly, because multiple threads can simultaneously update the same bin at line 7. An alternative approach is to allow threads to have their own local histogram to prevent

Algorithm 1 Brute-force enumeration (J, n, k, W)

Input: Length n, dimension k, and sequence $W = (\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{2^k-1})$ of codewords
Output: Joint weight histogram J

1: Initialize J;
2: **for** $i \leftarrow 0$ to $2^k - 1$ **do parallel**
3: **for** $j \leftarrow 0$ to $2^k - 1$ **do parallel**
4: $a \leftarrow f_{11}(\mathbf{w}_i, \mathbf{w}_j)$;
5: $b \leftarrow f_{10}(\mathbf{w}_i, \mathbf{w}_j)$;
6: $c \leftarrow f_{01}(\mathbf{w}_i, \mathbf{w}_j)$;
7: $J_{a,b,c} \leftarrow J_{a,b,c} + 1$;
8: **end for**
9: **end for**

simultaneous access to the same memory address. Although this approach avoids atomic instructions, a post-processing stage is needed to merge local histograms into a single global histogram. More details on this hierarchical organization are presented in [12].

For a multi-core CPU system, the first loop at line 2 can be parallelized using multiple threads by adding an OpenMP directive [15] such as `#pragma omp parallel for`. On the other hand, the nested loop structure from lines 2 to 9 can be replaced with a kernel function call for GPU-based acceleration. The kernel function implements the loop body from lines 4 to 7 to enumerate joint weight in parallel.

A. Joint weight computation

Joint weight $w(\mathbf{u}, \mathbf{v}) = (a, b, c)$ of a pair $(\mathbf{u}, \mathbf{v})$ of codewords can be given by

$$a = \text{popcount}(\mathbf{u} \wedge \mathbf{v}), \quad (4)$$

$$b = \text{popcount}(\mathbf{u}) - a, \quad (5)$$

$$c = \text{popcount}(\mathbf{v}) - a, \quad (6)$$

where $\wedge$ is bitwise logical conjunction and popcount($\mathbf{u}$) is a function that counts the number of bits set to 1 in the given vector $\mathbf{u}$. Figure 2 shows a naive implementation [18] of the popcount function. This implementation processes a vector of $l = 32$ bits in $\log l$ steps, which require 20 instructions containing five additions, five shifts, and ten logical conjunctions.

Instead of this implementation, the previous algorithm [12] employed a single vector instruction to process a vector of 64 bits. For multi-core CPUs and GPUs, the POPCNT instruction of SSE 4.2 [16] and the popc instruction of CUDA [13], respectively, were used to implement the popcount function. These vector instructions require 64-bit data as their operand, so that $\lceil n/64 \rceil$ instructions are needed to compute joint weight of a pair of codeword of length n.

B. Reduced enumeration using symmetry

Let $w(\mathbf{u}, \mathbf{v}) = (a, b, c)$ be the joint weight of pair $(\mathbf{u}, \mathbf{v})$ of codewords. The joint weight of the permutated pair $(\mathbf{v}, \mathbf{u})$ then is given by $w(\mathbf{v}, \mathbf{u}) = (a, c, b)$ [3]. This symmetric relation

```
1 int popcount(unsigned int x)
2 {
3   x=(x&0x55555555)+(x>>1&0x55555555);
4   x=(x&0x33333333)+(x>>2&0x33333333);
5   x=(x&0x0f0f0f0f)+(x>>4&0x0f0f0f0f);
6   x=(x&0x00ff00ff)+(x>>8&0x00ff00ff);
7   return (x&0x0000ffff)+(x>>16&0x0000ffff);
8 }
```

Fig. 2. Basic implementation [18] of the popcount function for 32-bit data.

Algorithm 2 Symmetric enumeration (J, n, k, W)

Input: Length n, dimension k, and sequence $W = (\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{2^k-1})$ of codewords
Output: Joint weight histogram J

1: Initialize J, D, and I;
2: **for** $i \leftarrow 0$ to $2^k - 1$ **do parallel**
3: **for** $j \leftarrow i$ to $2^k - 1$ **do parallel**
4: $a \leftarrow f_{11}(\mathbf{w}_i, \mathbf{w}_j)$;
5: $b \leftarrow f_{10}(\mathbf{w}_i, \mathbf{w}_j)$;
6: $c \leftarrow f_{01}(\mathbf{w}_i, \mathbf{w}_j)$;
7: **if** $i = j$ **then**
8: $D_{a,b,c} \leftarrow D_{a,b,c} + 1$;
9: **else**
10: $I_{a,b,c} \leftarrow I_{a,b,c} + 1$;
11: **end if**
12: **end for**
13: **end for**
14: **for** $a \leftarrow 0$ to n **do**
15: **for** $b \leftarrow 0$ to $n - a$ **do**
16: **for** $c \leftarrow 0$ to $n - a - b$ **do**
17: $J_{a,b,c} \leftarrow I_{a,b,c} + I_{a,c,b} + D_{a,b,c}$;
18: **end for**
19: **end for**
20: **end for**

indicates that joint weight distribution can be obtained from approximately half of pairs of codewords.

Consider a binary linear (n, k) code C that contains 2^k codewords $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{2^k-1}$. Let $I_{a,b,c}$ be the number of pairs of codewords such that $(\mathbf{w}_i, \mathbf{w}_j)$ has joint weight of $w(\mathbf{w}_i, \mathbf{w}_j) = (a, b, c)$, where $0 \leq i < j < 2^k$. Let $D_{a,b,c}$ also be the number of pairs of codewords such that $(\mathbf{w}_i, \mathbf{w}_i)$ has joint weight of $w(\mathbf{w}_i, \mathbf{w}_i) = (a, b, c)$, where $0 \leq i < 2^k$. We then have the following relation:

$$J_{a,b,c} = I_{a,b,c} + I_{a,c,b} + D_{a,b,c}. \quad (7)$$

I and D can be obtained from $2^k(2^k - 1)/2$ and 2^k pairs of codewords, respectively. Therefore, this symmetric relation reduces the number of pairs of codewords to be investigated from 2^{2k} to $2^k(2^k + 1)/2 \approx 2^{2k-1}$.

Algorithm 2 shows a joint weight enumeration algorithm that exploits this symmetry. The nested loop from lines 2 to 13 computes a joint weight histogram for approximately half of pairs of codewords. The entire histogram is then serially

Algorithm 3 Conflict tolerant enumeration (J, n, k, W)

Input: Length n, dimension k, and sequence $W = (\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{2^k-1})$ of codewords
Output: Joint weight histogram J

```
1: Sort W in ascending order in terms of Hamming weight;
2: W' ← ∅;
3: while (W ≠ ∅) do
4:     for i ← 0 to n do
5:         if (∃u ∈ W such that popcount(u) = i) then
6:             Delete u from W;
7:             Add u to W';
8:         end if
9:     end for
10: end while
11: Call symmetric enumeration (J, n, k, W');
```

computed from lines 14 to 20 using Eq. (7).

C. Conflict tolerant enumeration on GPU

As compared with the CPU, the GPU is a highly-threaded architecture capable of running thousands of threads simultaneously. Therefore, the overhead of atomic instructions can be a performance bottleneck in GPU-based enumeration. To deal with this issue, the previous GPU-based implementation [12] reduced write conflicts by scanning pairs of codewords such that different threads updated different bins. The symmetry mentioned in Section III-B was used to realize such conflict tolerant enumeration.

Suppose that pairs $(\mathbf{u}, *)$ and $(\mathbf{u}', *)$ of codewords are assigned to threads #1 and #2, respectively, where $*$ is an arbitral codeword to be investigated. Suppose that popcount$(\mathbf{u}) = a+b$ and popcount$(\mathbf{u}') = a' + b'$. We have $a \neq a'$ or $b \neq b'$ if popcount$(\mathbf{u}) \neq$ popcount$(\mathbf{u}')$. In this case, threads #1 and #2 update a different bin, and thus, write conflicts do not occur between them. Therefore, codewords to be investigated should be classified into groups in terms of popcount value (i.e., Hamming weight). Threads #1 and #2 then are responsible for pairs $(\mathbf{u}, *)$ and $(\mathbf{u}', *)$ of codewords such that popcount$(\mathbf{u}) \neq$ popcount$(\mathbf{u}')$.

A preprocessing stage is required to realize this assignment. That is, codewords should be sorted in ascending order in terms of Hamming weight. This preprocessing stage can be processed in $\mathcal{O}(2^k n)$ time, which is much smaller than $\mathcal{O}(2^{2k} n)$, or the time complexity of joint weight distribution computation. Consequently, this sorting operation is processed on a CPU. After this preprocessing stage, joint weight distribution is computed using Algorithm 2.

Algorithm 3 shows a pseudocode of the previous algorithm for GPU-based enumeration [12]. The preprocessing stage from lines 1 to 10 produces a sequence W' of sorted codewords, which is then given as an input to Algorithm 2.

IV. Proposed Joint Weight Enumeration

Our parallel joint weight enumeration algorithm consists of four acceleration techniques: (1) dimension reduction using the all-one vector, (2) joint weight computation with a population count instruction (Section III-A), (3) reduced enumeration using symmetry (Section III-B), and (4) conflict tolerant enumeration on the GPU (Section III-C).

A. Dimension reduction using all-one vector

Many practical codes such as Bose-Chaudhuri-Hocquenghem (BCH) codes [19], [20], Hamming codes, and Reed-Muller codes include the all-one vector $\mathbf{1}$ as a codeword. Given such code C, we have

$$\mathbf{u} + \mathbf{1} = \overline{\mathbf{u}} \in C, \text{ for all } \mathbf{u} \in C. \tag{8}$$

This implies that dimension k can be reduced for enumerating joint weight efficiently. That is, $w(\mathbf{u}+\mathbf{1}, \mathbf{v})$, $w(\mathbf{u}, \mathbf{v}+\mathbf{1})$, and $w(\mathbf{u}+\mathbf{1}, \mathbf{v}+\mathbf{1})$ can be obtained from $w(\mathbf{u}, \mathbf{v}) = (a, b, c, d)$ as follows:

$$w(\mathbf{u}+\mathbf{1}, \mathbf{v}) = (c, d, a, b), \tag{9}$$
$$w(\mathbf{u}, \mathbf{v}+\mathbf{1}) = (b, a, d, c), \tag{10}$$
$$w(\mathbf{u}+\mathbf{1}, \mathbf{v}+\mathbf{1}) = (d, c, b, a). \tag{11}$$

In other words, given codewords $\mathbf{u}$ and $\mathbf{v}$, there is no need to generate codewords $\mathbf{u}+\mathbf{1}$ and $\mathbf{v}+\mathbf{1}$ to compute their joint weight. Therefore, the number of pairs of codewords can be reduced to quarter.

B. Codeword generation with dimension reduction

Before using the dimension reduction technique mentioned above, we have to confirm whether the target code C includes the all-one vector $\mathbf{1}$ or not. A straightforward scheme to perform this confirmation is to use a parity-check matrix H of C: a vector $\mathbf{x}$ is a codeword of C if $H \cdot \mathbf{x}^{\mathrm{T}} = 0$, where $\mathbf{x}^{\mathrm{T}}$ is the transpose of $\mathbf{x}$. Otherwise, the dimension reduction technique cannot be used for enumeration. Although this scheme detects the all-one vector in C, it is not clear which codeword must be generated for enumeration.

Our alternative scheme is as follows. Let G be the canonical generator matrix [3] of a binary linear (n, k) code C. G can be represented as $G = [\ I_k | P\ ]$, where I_k is the $k \times k$ identity matrix and P is a $k \times (n-k)$ matrix.

1) Replace the bottom row G_k of G with summation of rows, $\sum_{1 \leq i \leq k} G_i$, where G_i represents the i-th row of G.
2) Check the bottom row G_k to detect the all-one vector.
    a) If $G_k = \mathbf{1}^{\mathrm{T}}$, code C includes the all-one vector $\mathbf{1}$. We then eliminate G_k from G, because codewords $\mathbf{u}+\mathbf{1}$ and $\mathbf{v}+\mathbf{1}$ are not needed to compute their joint weight, as mentioned in Section IV-A. This elimination reduces the dimension of C, because the generator matrix is then given by $G = [\ I_{k-1} | P'\ ]$, where P' is a $(k-1) \times (n-k)$ matrix.
    b) Otherwise, code C does not include the all-one vector $\mathbf{1}$.
3) Use G to generate the codewords of C.

TABLE I

SPECIFICATION OF EXPERIMENTAL MACHINES.

Item	Machine #1	Machine #2
# of CPU sockets	1	2
CPU	Core i7 3770K	Xeon E5-2680v2
# of cores per socket	4	10
Frequency	3.5 GHz	2.8 GHz
Main memory capacity	16 GB	512 GB
Peak memory bandwidth	25.6 GB/s	51.2 GB/s
GPU	GTX 680	Tesla K40
# of cores	1536	2880
Core clock	1006 MHz	745 MHz
VRAM capacity	2 GB	12 GB
Peak memory bandwidth	192 GB/s	288 GB/s
PCI Express bus	3.0 ×16	3.0 ×16
OS	Ubuntu 12.04 64-bit	Ubuntu 13.10 64-bit
C++ compiler	GCC 4.6.3	GCC 4.8.1
Graphics driver	331.62	
CUDA	6.0	
Compile option	`-arch sm_30 -O3`	`-arch sm_35 -O3`

Similar to the sorting operation presented in Section III-C, we decided to serially process this preprocessing stage on a CPU.

V. Experimental Results

To evaluate the performance of our parallel algorithm, we compared our implementation with the previous implementation [12] in terms of execution time. Table I shows the specification of our experimental machines. Our machines had a 4-core Core i7 CPU and two 10-core Xeon E5 CPUs, respectively. In addition, these machines were equipped with a GTX 680 GPU and a Tesla K40 GPU, respectively. Both GPUs were based on the Kepler architecture [14]. The error check and correct (ECC) capability of the K40 card was turned off during measurement. The algorithms were implementated using OpenMP to realize multithreaded enumeration. We created two threads per CPU core to take advantage of hyperthreading technology.

We used a subcode of the (127,22) binary BCH code [19], [20] of dimension t $(\leq k)$, where $11 \leq t \leq 22$. Consequently, a joint weight histogram was computed for 2^t codewords. The bin size of the global joint weight histogram was set to 8 bytes, because the maximum value of a bin could reach 2^{2t}, where $11 \leq t \leq 22$. On the other hand, the bin size of local joint histograms was set to 2 bytes and 4 bytes for the CPU- and GPU-based implementations, respectively [12].

A. Performance Comparison

Figure 3 shows the execution times for $n = 127$ and $t = 22$. The execution times of GPU-based implementations include the transfer time needed to copy data between the CPU and GPU. On all CPUs and GPUs, our algorithm reduced the execution time to quarter. These timing results are the same as what expected in Section IV-A. The highest performance was obtained on the Xeon processors, which reduced the execution time from 1,111 to 275 seconds. The speedup over the previous algorithm reached a factor of 4.03, which was slightly higher than a factor of 4.

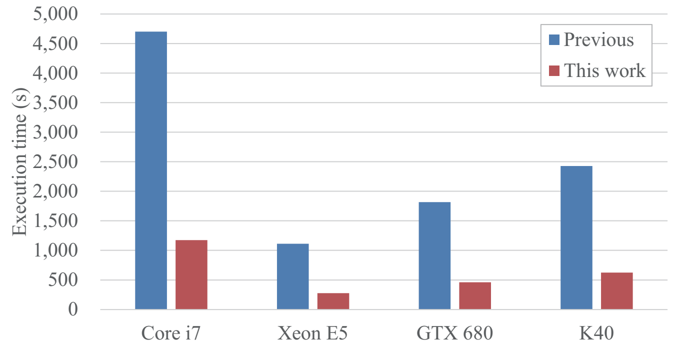

Fig. 3. Execution times of the proposed algorithm and the previous algorithm [12].

Because the K40 card had a higher memory bandwidth than the GTX 680 card, we expected that the former would outperform the latter for this memory-intensive application. However, we found that the GTX 680 card was 1.36 times faster than the K40 card for this application. Because this speedup ratio is close to the clock speed ratio $(1006/745 = 1.35)$, we think that this performance behavior is due to atomic instructions. The throughput of atomic instructions on the Kepler architecture varies according to the memory address to be accessed. For example, the worst throughput of 1 instruction per clock is obtained when threads access the same address simultaneously [21]. In contrast, this throughput increases to 8 instructions per streaming multiprocessor [13] per clock when threads in the same warp access different addresses in the same cache line. Consequently, the worst throughput rather than the best throughput determines the performance of our GPU-based implementation, though write conflicts are reduced by sorting codewords.

Our algorithm requires a preprocessing stage on the CPU. Both the CPU- and GPU-based implementations generate codewords before computing the joint weight histogram. The GPU-based implementation then sorts codewords and transfers them to the GPU. We found that the preprocessing overhead was negligible against the entire execution time. For example, when $t = 22$, the preprocessing time and the transfer time were approximately 0.009% and 0.002% of the execution time on the GTX 680 card, respectively. The performance of our algorithm is dominated by joint weight distribution computation.

B. Efficiency Analysis

We next analyzed the efficiency of the implementations in terms of memory throughput, because access to histogram bins determined their performance. The effective memory throughput is given by $B = AM/T$, where A, M, and T are the number of pairs of codewords, the amount of memory reads/writes per pair, and the execution time, respectively. Note that the execution time T here includes the preprocessing time and the transfer time mentioned above. For each pair of codewords of length n, our implementation updates an 8-byte bin. Therefore, we have $M = 2\lceil n/8 \rceil + 8$ in byte. Considering

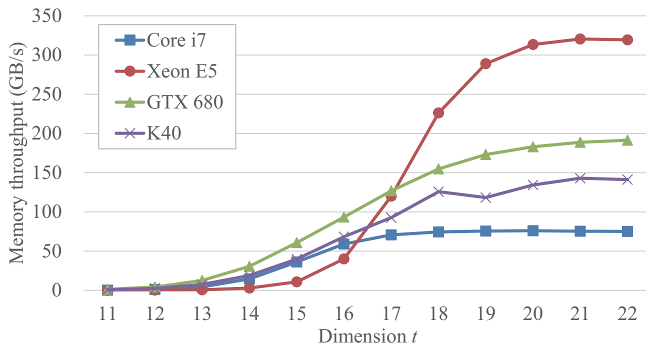

Fig. 4. Effective memory throughputs measured during parallel joint weight enumeration.

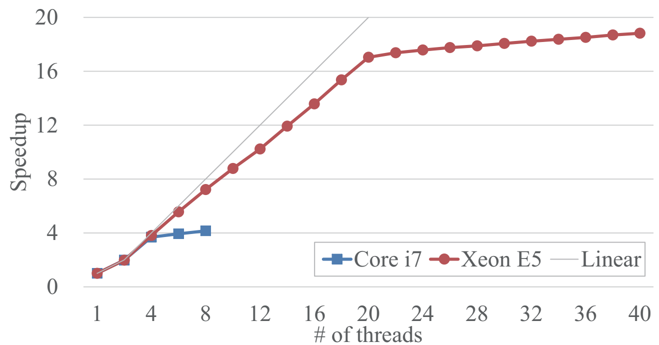

Fig. 5. Speedups over a single-core version on Core and Xeon processors.

combinations of two codewords to be selected from the total 2^k codewords, we have $A = \binom{2^k}{2}$.

Figure 4 shows the effective memory throughputs of the implementations. When $t = 22$, The effective memory throughputs of the GTX 680 and K40 reached 191 GB/s and 141 GB/s, respectively. These results were equivalent to 99.6% and 49.0% of the peak memory bandwidth, respectively. The former is close to 100%, demonstrating the effectiveness of hierarchical histogram organization mentioned in Section III. That is, local histograms are small enough to fit into the GPU cache called shared memory [13]. Therefore, the memory bandwidth was efficiently saved using the shared memory, increasing the effective memory throughput close to the peak memory bandwidth.

With respect to CPU-based results, we found that the effective memory throughputs were higher than the peak memory bandwidth. Similar to GPU-based results, this behavior can be explained by cache hits. Actually, a local joint weight histogram for $n \leq 128$ can be stored in a memory region of approximately 17 KB [12], which is smaller than the capacity of L1 cache (32 KB). Consequently, the instruction issue rate determines the performance of our CPU-based implementation. According to this analysis, the two Xeon processors are 4 times faster than the single Core processor, because the formers have five times more physical cores but with 20% slower clock rate than the latter. Actually, the Xeon processors achieved approximately 4.3 times higher memory throughput than the Core processor when $t = 22$.

When $t \leq 16$, the Xeon processors failed to outperform the Core processor. This is due to the memory allocation overhead. For example, memory allocation on this big memory machine took 0.4 seconds, whereas the execution time at $t = 16$ was 0.76 seconds.

Finally, Fig. 5 shows the speedup over a single-core version on the Core and Xeon processors, which have 4 and 20 physical cores, respectively. The performance of our implementation linearly increased with the number of threads. This performance behavior also explains why the memory throughputs are higher than the peak memory bandwidth. Because local histograms were entirely stored in the L1 cache, the performance was mainly dominated by the instruction issue rate. Therefore, the performance increased with the number of physical cores to be exploited for enumeration. Owing to the hyperthreading technology, the performance slightly increased after assigning the second threads to CPU cores.

VI. Conclusion

In this paper, we presented a parallel algorithm for enumerating joint weight of a binary linear (n, k) code. Our algorithm reduces the number of pairs of codewords to be investigated. To realize this, we reduce the dimension k of the code by focusing on the all-one vector, which is included in typical error-correcting codes. Our algorithm also employ a population count instruction to reduce the number of instructions needed to compute joint weight. We also sort codewords in terms of Hamming weight to realize conflict tolerant enumeration.

Our experimental results showed that the dimension reduction reduced the execution time to quarter on multi-core CPUs and a GPU. We also found that the performance of our GPU-based implementation was dominated by the core clock speed of the GPU. Similarly, our CPU-based implementation had a performance bottleneck in the instruction issue rate.

Future work includes further exploitation of code structure such as trellis diagram [17]. The MacWilliams identity is also useful to accelerate enumeration for codes of large dimension.

Acknowledgment

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Number 24560458 and the Japan Science and Technology Agency CREST program, "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems."

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Information Theory*, vol. 46, no. 4, pp. 1204–1216, Jul. 2000.

[2] S.-Y. R. Li, R. W. Yeung, and N. Cai, "Linear network coding," *IEEE Trans. Information Theory*, vol. 49, no. 2, pp. 371–381, Feb. 2003.

[3] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[4] M. Mohri, Y. Honda, and M. Morii, "A method for computing the local distance profile of binary cyclic codes," *IEICE Trans. Fundamentals (Japanese Edition)*, vol. J86-A, no. 1, pp. 60–74, Jan. 2003.

[5] Y. Kido and T. Fujiwara, "MacWilliams identity for joint weight enumerator to evaluate decoding error probability of linear block code in network coding," in *Proc. 34th Symp. Information Theory and its Applications (SITA'11), 3.4.1*, Nov. 2011, pp. 184–189, (In Japanese).
[6] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*, 2nd ed. Prentice Hall, 2004.
[7] D. Luebke and G. Humphreys, "How GPUs work," *Computer*, vol. 40, no. 2, pp. 96–100, Feb. 2007.
[8] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008.
[9] K. Ikeda, F. Ino, and K. Hagihara, "Efficient acceleration of mutual information computation for nonrigid registration using CUDA," *IEEE J. Biomedical and Health Informatics*, vol. 18, no. 3, pp. 956–968, May 2014.
[10] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 4, pp. 751–759, Apr. 2012.
[11] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," *Parallel Computing*, vol. 36, no. 2/3, pp. 129–141, Feb. 2010.
[12] S. Ando, F. Ino, T. Fujiwara, and K. Hagihara, "A parallel method for accelerating joint weight distribution computation," *IEICE Trans. Information and Systems (Japanese Edition)*, vol. J97-D, no. 9, pp. xx–xx, Sep. 2014, (In Japanese).
[13] NVIDIA Corporation, "CUDA C Programming Guide Version 6.0," Feb. 2014, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[14] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," May 2012, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
[15] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA: Morgan Kaufmann, Oct. 2000.
[16] Intel Corporation, "Intel SSE4 Programming Reference," Jul. 2007, http://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf.
[17] Y. Desaki, T. Fujiwara, and T. Kasami, "A method for computing the weight distribution of a block code by using its trellis diagram," *IEICE Trans. Fundamentals*, vol. E77-A, no. 8, pp. 1230–1237, Aug. 1994.
[18] H. S. Warren, *Hacker's Delight*, 2nd ed. Addision-Wesley Professional, 2012.
[19] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, Mar. 1960.
[20] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres*, vol. 29, no. 2, pp. 147–156, Apr. 1959, (In French).
[21] L. Nyland and S. Johns, "Understanding and using atomic memory operations," in *4th GPU Technology Conf. (GTC'13)*, Mar. 2013, http://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf.