International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 2, pages 290–303, July 2015

Enumerating Joint Weight of a Binary Linear Code Using Parallel Architectures: multi-core CPUs and GPUs[1]

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

Received: February 15, 2015
Revised: May 2, 2015
Accepted: June 1, 2015
Communicated by Susumu Matsumae

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. Our algorithm is implemented on a multi-core CPU system and an NVIDIA graphics processing unit (GPU) system using OpenMP and compute unified device architecture (CUDA), respectively. 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. Furthermore, an efficient atomic vote and reduce scheme is deployed in our GPU-based implementation. We apply our CPU- and GPU-based implementations to a subcode of a (127,22) BCH code to evaluate the impact of acceleration.

Keywords: Joint weight, joint weight histogram, acceleration, atomics, GPU

1 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.* [14] 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* [17] 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 $\mathbf{u} \in C$ $(\mathbf{u} \neq \mathbf{v})$.

[1]A preliminary version [2] was presented at the CANDAR 2014 conference.

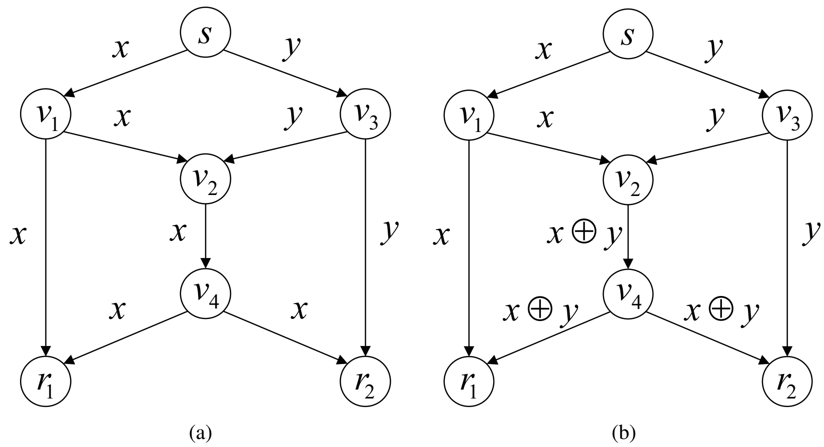

Figure 1: An example of multicast communication using network coding over the butterfly network. (a) Node r_1 fails to receive y if node v_2 transmits x rather than y. (b) Network coding allows nodes r_1 (r_2) to receive both x and y, because y (x) can be extracted from x (y) and $x \oplus y$.

High-performance error correction can be achieved by not only maximizing the error-correcting capability but also minimizing the decoding error probability. Notice here that the meaning of the term "performance" is different from that usually used in the high-performance computing community, where the performance is usually associated with timing aspects such as execution time. Thus, rapid computation of the decoding error probability is useful to design an error-correcting code for practical networks.

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 $(0 \leq i \leq n)$ is the number of codewords of Hamming weight i in C [19]. 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 [13]. In this work, we deal with the problem of the butterfly network, so that we assume that $m = 2$ hereafter.

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 [17]. 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 [15] 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) [16] are attractive methods for achieving acceleration for compute- and memory-intensive applications [8, 10, 11, 25]. Ando *et al.* [3] 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 [3] 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 [3], 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 [23]. We assume that $n \leq 128$ and the target machine is equipped with an NVIDIA Kepler GPU [22]. We extend our preliminary results [2] with an efficient atomic vote and reduce scheme [7].

The following paper is organized as follows. Section 2 introduces some related studies. Section 3 presents preliminaries on joint weight enumeration of a code. Section 4 describes our parallel algorithm for enumerating joint weight of a binary linear (n, k) code. Section 5 presents experimental results. Finally, Section 6 shows conclusion and future work.

2 Related Work

Ando *et al.* [3] 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 [5], which achieved multithreading by simply adding compiler directives to the serial code. Furthermore, single instruction multiple data (SIMD) instructions, called Streaming SIMD Extensions (SSE) [12], 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 [17] 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. [13] applied the MacWilliams identity to an m-joint weight enumerator from a theoretical point of view.

With respect to weight distribution, Desaki *et al.* [6] 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.

There are many studies that accelerated histogram computation on a GPU. Podlozhnyuk [26] accelerated 256-bin histogram computation for image processing applications. This GPU-based implementation exploited on-chip shared memory [23] to realize efficient histogram computation. In contrast to this 256-bin histogram, our target problem requires a joint weight histogram that requires $\mathrm{O}(n^3)$ space. This relatively large data cannot be stored entirely in shared memory, whose capacity is in the order of KB. Due to the same reason, similar approaches [18, 20, 27] that exploited shared memory for histogram computation cannot be applied directly to our target problem.

Ikeda *et al.* [10] accelerated joint histogram computation for nonrigid registration of medical images. Their GPU-based implementation reduces the number of histogram bins by taking advantage of typical distribution of X-ray intensities. This reduction enables joint histogram data to be stored in on-chip shared memory. With respect to joint weight of codewords, such a typical distribution is not known. However, the space complexity of a joint weight histogram can be reduced from $\mathrm{O}(n^3)$ space to $\mathrm{O}(n^2)$ space if one of a codeword pair is given [3]. This reduction allows our algorithm to exploit on-chip memory with higher locality in joint weight histograms. This localization was effective on CPU-based implementations and GPU-based implementation running on the Fermi architecture [21], but it was not effective on the Kepler architecture [22], which has a higher atomic instruction throughput than the Fermi.

3 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

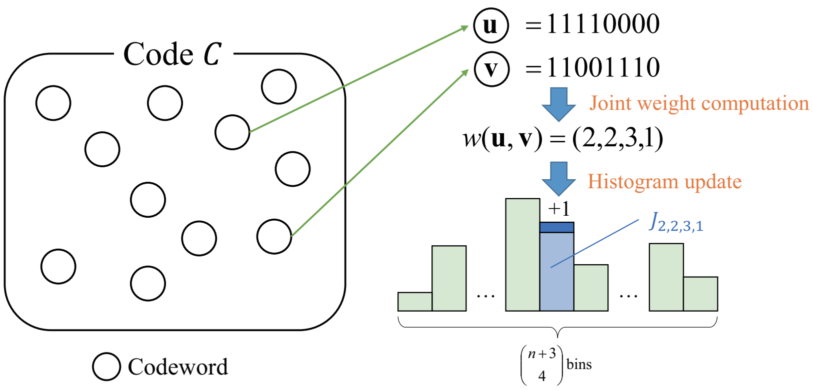

Figure 2: An overview of joint weight enumeration of a code.

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**

$p, q \in \mathbb{F}$. 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 $d = n - a - b - c$ [17], 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

$$0 \leq a, b, c \leq n, \tag{2}$$
$$0 \leq a + b + c \leq n. \tag{3}$$

Figure 2 shows an overview of joint weight enumeration of a code. 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 [23] 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 simultaneous access to the same memory address. Although this approach avoids atomic instructions,

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

Figure 3: Basic implementation [28] of the popcount function for 32-bit data.

a post-processing stage is needed to merge local histograms into a single global histogram. More details on this hierarchical organization are presented in [3].

For a multi-core CPU system, the first loop at line 2 can be parallelized using multiple threads by adding an OpenMP directive [5] 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.

3.1 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

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

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

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

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 3 shows a naive implementation [28] 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 [3] 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 [12] and the popc instruction of CUDA [23], 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.

3.2 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 $w(\mathbf{v}, \mathbf{u}) = (a, c, b)$ [17]. This symmetric relation 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 $0 \le i < j < 2^k$. 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 $0 \le i < 2^k$. We then have the following relation:

$$J_{a,b,c} = I_{a,b,c} + I_{a,c,b} + D_{a,b,c}. \tag{7}$$

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

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

Initialize J, D, and I;
for $i \leftarrow 0$ to $2^k - 1$ **do parallel**
  for $j \leftarrow i$ to $2^k - 1$ **do parallel**
    $a \leftarrow f_{11}(\mathbf{w}_i, \mathbf{w}_j)$;
    $b \leftarrow f_{10}(\mathbf{w}_i, \mathbf{w}_j)$;
    $c \leftarrow f_{01}(\mathbf{w}_i, \mathbf{w}_j)$;
    if $i = j$ **then**
      $D_{a,b,c} \leftarrow D_{a,b,c} + 1$;
    else
      $I_{a,b,c} \leftarrow I_{a,b,c} + 1$;
    end if
  end for
end for
for $a \leftarrow 0$ to n **do**
  for $b \leftarrow 0$ to $n - a$ **do**
    for $c \leftarrow 0$ to $n - a - b$ **do**
      $J_{a,b,c} \leftarrow I_{a,b,c} + I_{a,c,b} + D_{a,b,c}$;
    end for
  end for
end for

3.3 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 [3] reduced write conflicts by scanning pairs of codewords such that different threads updated different bins. The symmetry mentioned in Section 3.2 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 arbitrary 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 [3]. 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.

4 Proposed Joint Weight Enumeration

Our parallel joint weight enumeration algorithm consists of five acceleration techniques: (1) dimension reduction using the all-one vector, (2) efficient atomics on the GPU, (3) joint weight computation with a population count instruction (Section 3.1), (4) reduced enumeration using symmetry (Section 3.2), and (5) conflict tolerant enumeration on the GPU (Section 3.3).

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

```
Sort W in ascending order in terms of Hamming weight;
W' ← ∅;
while (W ≠ ∅) do
    for i ← 0 to n do
        if (∃u ∈ W such that popcount(u) = i) then
            Delete u from W;
            Add u to W';
        end if
    end for
end while
Execute symmetric enumeration (J, n, k, W');
```

4.1 Dimension reduction using all-one vector

Many practical codes such as Bose-Chaudhuri-Hocquenghem (BCH) codes [4,9], 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.

4.2 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 [17] 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, $\sum_{1 \le i \le k} G_i$, 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 4.1. 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.

Similar to the sorting operation presented in Section 3.3, we decided to serially process this preprocessing stage on a CPU.

4.3 Efficient atomics on GPU

Our conflict tolerant enumeration in Section 3.3 intends to reduce the number of conflicts caused by atomic instructions. Although the Kepler architecture [22] achieves 9 times higher atomic instruction throughput than the previous Fermi architecture [21], it is significantly lower than the peak memory bandwidth available on the GPU. To deal with this problem, we integrate an efficient atomic scheme [7] into our GPU-based implementation. This scheme reduces the number of conflicts among threads in a warp [23] in which threads execute the same instruction at the same time. In other words, threads in a warp can cause many conflicts owing to their simultaneous behavior, but at the same time, this simultaneous behavior is useful to detect and eliminate such intra-warp conflicts before accessing data atomically.

There are three variations in this atomic scheme: atomic vote and reduce (atomicVR), atomic scan and reduce (atomicSR) and atomic set scatter (atomicSS) algorithms. Among these three variations, we use the atomicVR algorithm, which is efficient for non-clustered conflicts frequently appeared in histogram computation and graph traversal [7]. Non-clustered conflicts here correspond to situations in which the conflicting threads appear randomly in the warp. Firstly, the atomicVR algorithm obtains the memory address of a randomly selected thread in a warp. It then executes the `__ballot()` function to compute the number of threads that access the same memory address. If this number is smaller than a threshold α, the algorithm performs the naive atomic instruction. Otherwise, the algorithm locally reduces the values of the corresponding threads in parallel. Because this parallel reduction is carried out within a warp, where threads are implicitly synchronized each other, barrier synchronization is not necessary for this reduction. A representative thread then writes the reduced value to the memory address atomically. We experimentally determined to use $\alpha = 16$ in our implementation (see Section 5).

Note that the atomicSR algorithm is efficient for clustered conflicts such as those observed in sparse matrix vector multiplication. In this case, the conflicting threads have consecutive thread indexes. Consequently, local parallel reduction is useful to eliminate conflicts within a warp. Finally, the atomicSS algorithm classifies threads in a warp in terms of the memory addresses they access, and then iteratively generates a group of threads that access different memory addresses. Owing to this classification and iteration, the atomicSS algorithm incurs a relatively large overhead compared to the atomicVR and atomicSR algorithms.

5 Experimental Results

To evaluate the performance of our parallel algorithm, we compared our implementation with the previous implementation [3] in terms of execution time. Table 1 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 [22]. The error check and correct (ECC) capability of the K40 card was turned off during measurement. The algorithms were implemented 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 [4, 9] 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 [3].

As mentioned above, our GPU-based implementation uses $\alpha = 16$ as a threshold for switching the atomic implementation. We measured the execution time of our joint weight enumeration program with varying α from 1 to 32. We then selected the best value $\alpha = 16$, which minimized the execution time for the largest dimension $t = 22$. The worst result was obtained with $\alpha = 1$, which was roughly 30% slower than $\alpha = 16$ for all $11 \leq t \leq 22$.

Table 1: 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 frequency	1006 MHz	745 MHz
VRAM capacity	2 GB	12 GB
Peak memory bandwidth	192 GB/s	288 GB/s
PCI Express bus	3.0 $\times$16	3.0 $\times$16
OS	Ubuntu 12.04 64-bit	Ubuntu 13.10 64-bit
C++ compiler	GCC 4.6.3	GCC 4.8.2
Graphics driver	340.29	
CUDA	6.5	
Compile option	`-arch sm_30 -O3`	`-arch sm_35 -O3`

5.1 Performance Comparison

We first present results with the largest dimension $t = 22$ to facilitate understanding of typical results. Figure 4 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 except for the K40 card, our dimension reduction technique reduced the execution time to quarter. These timing results are the same as what expected in Section 4.1. On the K40 card, our dimension reduction technique reduced the execution time from 36.3 to 10.5 seconds, achieving a speedup of 3.5$\times$. This implies that our dimension reduction technique slightly reduced its effectiveness on the K40 card. Because our dimension reduction technique reduced the number of kernel invocations into quarter, the K40 card slightly degraded the efficiency of kernel execution. Unfortunately, we could not identify the reason for this internal behavior. The highest performance was obtained on the Xeon processors, which reduced the execution time from 18.5 to 4.6 minutes (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. With respect to GPU-based implementations, the atomicVR algorithm further reduced execution time by 5% and 3% on the GTX 680 card and the K40 card, respectively.

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 1.34 ($= 10.2/7.6$) times faster than the K40 card for this application. Because this speedup ratio is close to the clock frequency ratio ($1006/745 = 1.35$), we think that the core clock frequency rather than the memory bandwidth determines the performance of our GPU-based implementation. In fact, the worst throughput of atomic instructions on the Kepler architecture is 1 instruction per clock, which is obtained when threads access the same address simultaneously [24]. Consequently, the atomic instruction is still the performance bottleneck of our algorithm, though write conflicts are reduced by sorting codewords and using the atomicVR algorithm. Actually, when $t = 22$, only 564 histogram bins had non-zero values, which ranged from 1 to 773,930,601,234. Because the atomicVR algorithm avoids intra-warp conflicts, inter-warp conflicts must be avoided to achieve further performance improvement.

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.

Figure 5 shows the execution times for $n = 127$ and $11 \le t \le 22$. Notice that a logarithmic scale

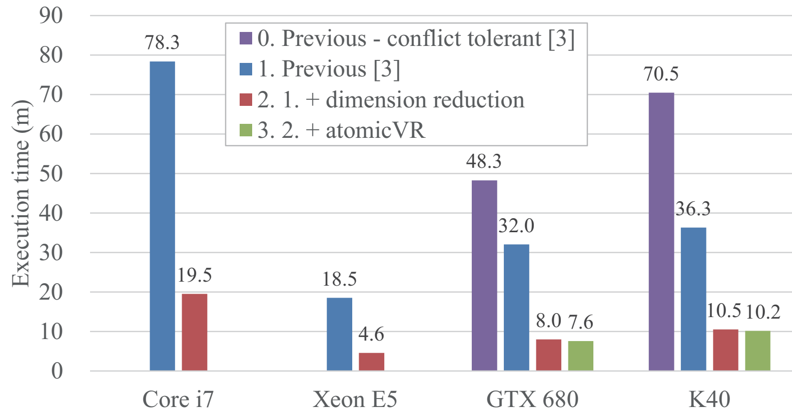

Figure 4: Execution times of the proposed algorithm and the previous algorithm [3] ($t = 22$).

is used for the vertical axis. As compared with the previous algorithm, our dimension reduction technique successfully reduced the execution time for all t. The atomic scheme further reduced the execution time for all t, but its impact was limited: the improvement ratio over the previous algorithm remained constant (approximately 3–5%) for all $11 \leq t \leq 22$. However, the speedup was lower than a factor of 3 for small $t < 17$. Such small problems can be completed within 1 second, and thus, reducing the kernel execution time was not so effective.

When $t \leq 16$, the Xeon processors failed to outperform the Core processor, and its execution time did not increase with t. 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.

5.2 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 combinations of two codewords to be selected from the total 2^t codewords, we have $A = \binom{2^t}{2}$ for algorithms without dimension reduction. With dimension reduction, we have $A = \binom{2^{t-1}}{2}$.

Figure 6 shows the effective memory throughputs of the implementations. When $t = 22$, the effective memory throughputs of the GTX 680 and K40 reached 193.3 GB/s and 144.0 GB/s, respectively. These results were equivalent to 100.6% and 50.0% of the peak memory bandwidth, respectively. The former slightly exceeds 100%, demonstrating the effectiveness of hierarchical histogram organization mentioned in Section 3. That is, local histograms are small enough to fit into the GPU cache called shared memory [23]. Therefore, the memory bandwidth was efficiently saved using the shared memory, increasing the effective memory throughput close to the peak memory bandwidth.

Without the atomicVR algorithm, the efficiency reduced to 95.1% and 48.4% (182.7 GB/s and 139.3 GB/s) on the GTX 680 and K40, respectively. Without our conflict tolerant enumeration, the efficiency further reduced to 63.2% and 28.9% (121.5 GB/s and 83.2 GB/s) on the GTX 680 and K40, respectively. Thus, write conflicts were mainly eliminated by conflict tolerant enumeration rather than the atomicVR algorithm.

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 [3], 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

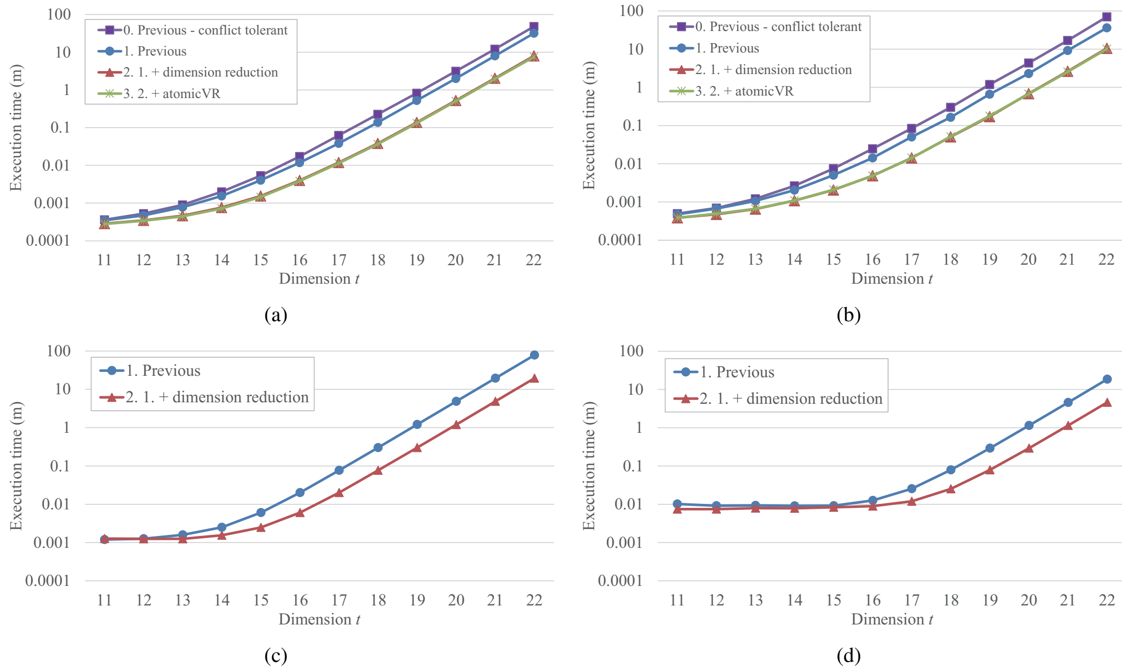

Figure 5: Execution times of the proposed algorithm and the previous algorithm [3] ($11 \leq t \leq 22$). Results on (a) GTX 680, (b) K40, (c) Core i7 and (d) Xeon E5.

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 $t = 22$.

Finally, Fig. 7 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.

6 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. In addition, we sort codewords in terms of Hamming weight to realize conflict tolerant enumeration. An efficient atomic scheme [7] is integrated into our GPU-based implementation to avoid conflicts within a warp.

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 [6]. The MacWilliams identity is also useful to accelerate enumeration for codes of large dimension. We also plan to evaluate our algorithm with other practical codes.

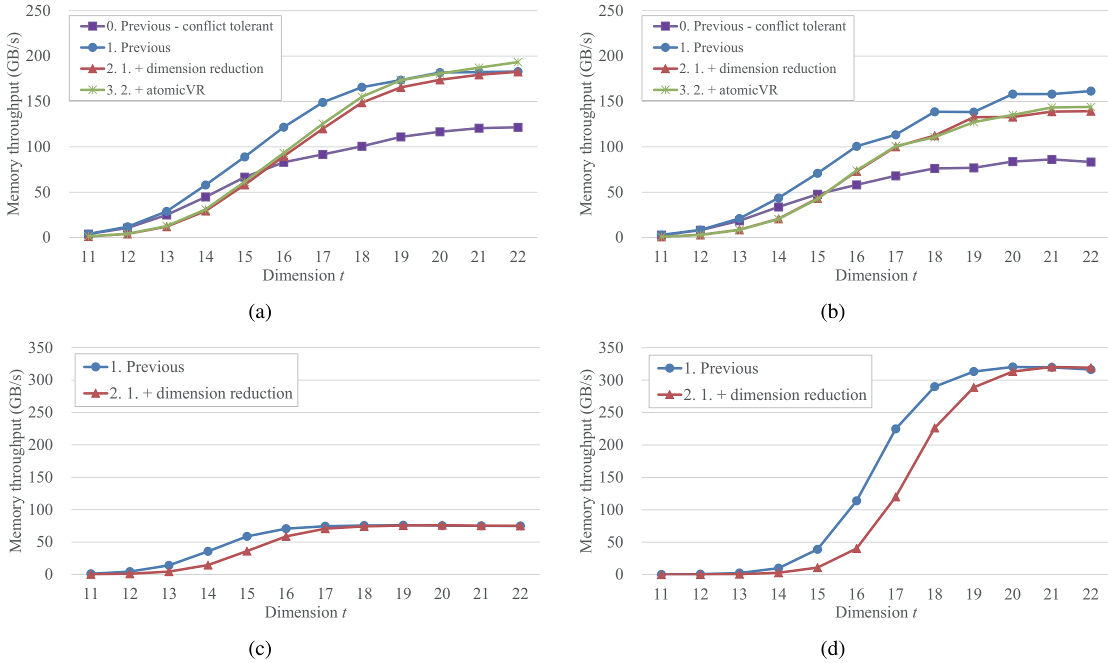

Figure 6: Effective memory throughputs measured during parallel joint weight enumeration. Results on (a) GTX 680, (b) K40, (c) Core i7 and (d) Xeon E5.

Acknowledgment

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 24560458, 15K12008 and 15H01687, and the Japan Science and Technology Agency CREST program, "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE Trans. Information Theory*, 46(4):1204–1216, July 2000.

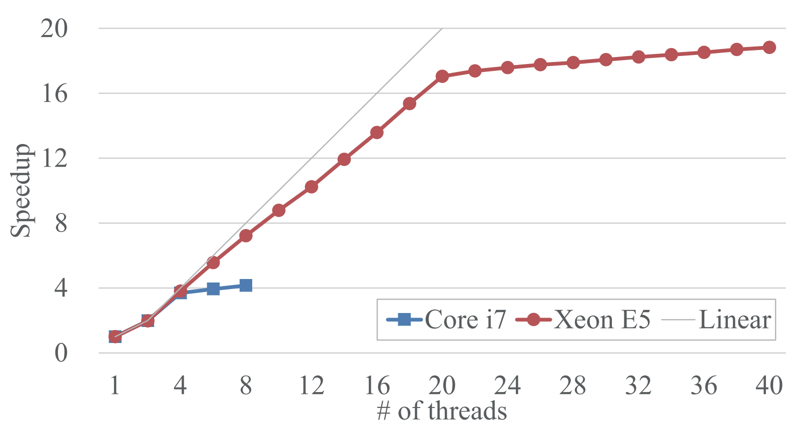

Figure 7: Speedups over a single-core version on Core and Xeon processors.

[2] Shohei Ando, Fumihiko Ino, Toru Fujiwara, and Kenichi Hagihara. A parallel algorithm for enumerating joint weight of a binary linear code in network coding. In *Proc. 2nd Int'l Symp. Computing and Networking (CANDAR'14)*, pages 137–143, December 2014.

[3] Shohei Ando, Fumihiko Ino, Toru Fujiwara, and Kenichi Hagihara. A parallel method for accelerating joint weight distribution computation. *IEICE Trans. Information and Systems (Japanese Edition)*, J97-D(9):1471–1480, September 2014. (In Japanese).

[4] R. C. Bose and D. K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, March 1960.

[5] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Mateo, CA, October 2000.

[6] Yoshihisa Desaki, Toru Fujiwara, and Tadao Kasami. A method for computing the weight distribution of a block code by using its trellis diagram. *IEICE Trans. Fundamentals*, E77-A(8):1230–1237, August 1994.

[7] Ian Egielski, Jesse Huang, and Eddy Z. Zhang. Massive atomics for massive parallelism on GPUs. In *Proc. 4th Int'l Symp. Memory Management (ISMM'14)*, pages 93–103, June 2014.

[8] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008.

[9] A. Hocquenghem. Codes correcteurs d'erreurs. *Chiffres*, 29(2):147–156, April 1959. (In French).

[10] Kei Ikeda, Fumihiko Ino, and Kenichi Hagihara. Efficient acceleration of mutual information computation for nonrigid registration using CUDA. *IEEE J. Biomedical and Health Informatics*, in print.

[11] Fumihiko Ino, Yuma Munekawa, and Kenichi Hagihara. Sequence homology search using fine grained cycle sharing of idle GPUs. *IEEE Trans. Parallel and Distributed Systems*, 23(4):751–759, April 2012.

[12] Intel Corporation. Intel SSE4 Programming Reference, July 2007. `http://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf`.

[13] Yoshiro Kido and Toru 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*, pages 184–189, November 2011. (In Japanese).

[14] Shuo-Yen Robert Li, Raymond W. Yeung, and Ning Cai. Linear network coding. *IEEE Trans. Information Theory*, 49(2):371–381, February 2003.

[15] Shu Lin and Daniel J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, second edition, 2004.

[16] David Luebke and Greg Humphreys. How GPUs work. *Computer*, 40(2):96–100, February 2007.

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

[18] Ugljesa Milic, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Milo Tomasevic. Parallelizing general histogram application for CUDA architectures. In *Proc. 13rd Int'l Conf. Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS'13)*, July 2013.

[19] Masami Mohri, Yukihiro Honda, and Masakatu Morii. A method for computing the local distance profile of binary cyclic codes. *IEICE Trans. Fundamentals (Japanese Edition)*, J86-A(1):60–74, January 2003.

[20] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs. In *Proc. 4th Workshop General Purpose Processing on Graphics Processing Units (GPGPU'11)*, March 2011.

[21] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, November 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[22] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, May 2012. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[23] NVIDIA Corporation. CUDA C Programming Guide Version 6.5, August 2014. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[24] Lars Nyland and Stephen Johns. Understanding and using atomic memory operations. In *4th GPU Technology Conf. (GTC'13)*, March 2013. `http://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf`.

[25] Yusuke Okitsu, Fumihiko Ino, and Kenichi Hagihara. High-performance cone beam reconstruction using CUDA compatible GPUs. *Parallel Computing*, 36(2/3):129–141, February 2010.

[26] Victor Podlozhnyuk. Histogram calculation in CUDA, July 2012. `http://docs.nvidia.com/cuda/samples/3_Imaging/histogram/doc/histogram.pdf`.

[27] Ramtin Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int'l Conf. Signal Processing and Communications Systems (ICSPCS'07)*, pages 418–422, December 2007.

[28] Henry S. Warren. *Hacker's Delight*. Addision-Wesley Professional, second edition, 2012.