Vol.2012-HPC-133 No.32
2012/3/27

GPGPUによる多元LDPC符号を用いた可変レート伝送のシミュレーション

藤坂祐介[†1]　金子晴彦[†1]

近年，コンピュータや高機能な携帯端末などの普及により，より信頼性がありかつ高速な無線通信技術が求められている．多元 LDPC 符号は誤り訂正能力の高い誤り訂正符号として知られているが，復号時の計算量が多く応用技術に活かされていないのが現状である．本稿では，可変レート多元 LDPC 符号がより高速かつ効率的に符号化，および復号を行うためのシミュレーション環境を作ることを目的として，GPGPU による符号化・復号ソフトウェアを実装した．本研究の結果，符号化については CPU によるプログラムと比べ最大で約 160 倍高速となり，また復号については最大で約 950 倍高速に動作することがわかった．

The Simulation of Variable-Rate Transmission using Non-Binary LDPC Codes with GPGPU

Yusuke Fujisaka[†1] and Haruhiko Kaneko[†1]

The spread of computer and high-spec mobile phones demands faster and more reliable wireless communication. Non-binary LDPC codes are known as strong error-correcting codes, but no application exists so far, because the complexity of the decoding algorithm is prohibitive. In this study, we implemented encoding and decoding software for Non-binary LDPC codes on GPGPU, with the aim of creating a simulation environment to design faster and more reliable rate-compatible non-binary LDPC codes. In our results, we show that our GPGPU based encoding program worked more than 160 times faster than the CPU based program, and the decoding program worked more than 950 times faster.

†1 東京工業大学
Tokyo Institute of Technology

1. はじめに

近年，コンピュータや高機能な携帯端末が普及し，また動画や音声など情報量の多いメディアが配信されるようになったため，より高速で効率の良い無線通信が求められている．一般に無線通信において，ノイズや通信の寸断に対してデータを修正する方法として，誤り訂正符号を用いる．多元 Low-Density Parity-Check(LDPC) 符号は，誤り訂正符号の一つであり，Gallager[1] によって提案された二元 LDPC 符号について，符号語の要素である各ビットをガロア体の要素に置き換えたものであり，1998 年に Davey ら[2] によって提案された．多元 LDPC 符号は，誤り訂正能力が非常に高いことが知られている．しかし，LDPC 符号は他の線形符号と異なり，復号時に Belief Propagation(BP) 法と呼ばれる，符号語の各シンボルごとの事後確率を他のシンボルに伝播させ，誤り訂正を行う方法を用いる．多元 LDPC 符号においては，符号語の各シンボルについて，並列的に復号を行う方法がいくつか[4)6)7)8)] 提案されてはいるものの，符号語の各シンボルについて，ガロア体の要素の値それぞれについて確率計算が必要なため，二元 LDPC 符号と比べ復号時の計算量が飛躍的に増大する．そのため，通信などにおける応用技術には活かされていない．

本研究は，多元 LDPC 符号の符号化と復号がより高速かつ効率的に行えるよう検討するため，高速に動作するシミュレーション環境を整備することを目的として，並列プロセッサである GPU を汎用計算に用い，様々な符号化率をとることができる，可変レート多元 LDPC 符号の符号化，および復号をシミュレートするプログラムを実装した．GPGPU によるプログラムと，CPU による同様の処理を行うプログラムと動作スループットを比較した．

以下，2 章では多元 LDPC 符号について述べる．3 章では GPGPU による可変レート多元 LDPC 符号の符号化について，4 章では同じく GPGPU による可変レート多元 LDPC 符号の復号法について提案する．5 章では符号化，および復号のスループットを CPU による処理と比較して示す．

2. 可変レート多元 Low-Density Parity-Check(LDPC) 符号

2.1 多元 LDPC 符号

多元 LDPC 符号は，線形符号の一種であり，符号語 $\mathbf{c}$ とパリティ検査行列 H の関係が

$$H\mathbf{c}^T = \mathbf{0} \tag{1}$$

を満たす符号である．LDPC 符号の特徴として，H が非常に疎であることが挙げられる．

一般に，符号長 n に対し，非零の要素数は $O(n)$ である．

2.2 可変レート LDPC 符号

符号語のうち，情報語が占める割合を符号化率という．符号化率を変更できる誤り訂正符号を可変レート符号という．LDPC 符号においては，以下 2 通りのレート変更の方法がある．

- 符号長をそのままに，情報シンボル数を増やす: パリティ検査行列の行数を減らす
- 情報シンボル数はそのままに，符号長を短くする: パリティ検査行列の列数と行数を減らす

上記のように，パリティ検査行列の列数，および行数を変更することで可変レートを実現する[11]．

3. GPGPU による可変レート多元 LDPC 符号の符号化

GPGPU に適した多元 LDPC 符号の符号化法として，式 (1) を基に，符号語をパリティ検査行列 H から直接求める方法[5] を用いる．GF(2^m) 上の長さ k の情報語から，長さ n の符号語を生成することを考える．$(n-k) \times n$ の大きさを持つパリティ検査行列 H について，2 種類の形式を持つ行列について符号化を行う．

形式 1　パリティ検査行列 H が，$(n-k) \times k$-行列 A と，$(n-k) \times (n-k)$-下三角行列 T の連結 $H = (A\ T)$ であるとき，符号語 $\mathbf{c}$ を情報語 $\mathbf{u}$ とパリティ $\mathbf{p}$ の連結 $(\mathbf{u}\ \mathbf{p})$ として生成する．式 (1) より，パリティ列は

$$\mathbf{p} = -T^{-1}A\mathbf{u} \tag{2}$$

として求められる．

形式 2　[5] パリティ検査行列 H が，以下の式 (3) で与えられるとする．

$$H = \begin{pmatrix} A & B & T \\ C & D & E \end{pmatrix} \tag{3}$$

ただし，各小行列 A, B, C, D, E, T は以下の通り $(1 \le g \le (n-k-1))$ である．

- A: $k \times g$-行列
- B: $(n-k-g) \times g$-行列
- C: $k \times (n-k-g)$-行列
- D: $(n-k-g) \times (n-k-g)$-行列
- E: $g \times (n-k-g)$-行列

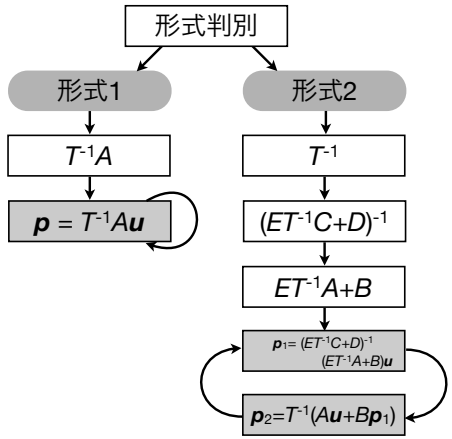

図 1　符号化処理

- T: $g \times g$-正則下三角行列

符号語 $\mathbf{c} = (\mathbf{u}\ \mathbf{p}_1\ \mathbf{p}_2)$ のパリティ $\mathbf{p}_1, \mathbf{p}_2$ は，同様に式 (1) を用いて，以下のように計算できる．

$$\mathbf{p}_1 = (-ET^{-1}C + D)^{-1}(-ET^{-1}A + B)\mathbf{u}$$
$$\mathbf{p}_2 = T^{-1}(A\mathbf{u} + B\mathbf{p}_1)$$

ここで，パリティが計算できる条件として $(-ET^{-1}C + D)$ が正則であることが必要である．

GPGPU 上のプログラムは図 1 の順に処理を行う．ベクトル計算においては，図 2 のように GPU の処理単位である，スレッドブロックおよびスレッドごとの処理を調整する．1 つのスレッドごとに行列の 1 行を計算し，1 シンボルのパリティを出力する．複数 (最大 256 スレッド) のスレッドをまとめてスレッドブロックを構成し，スレッドブロック単位で共有して利用できる Shared メモリに行数分の行列要素を読み込む．

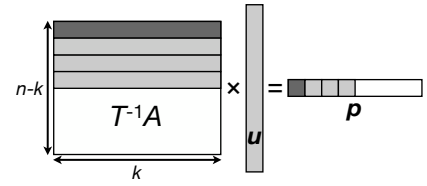

図 2　ベクトル計算

Vol.2012-HPC-133 No.32
2012/3/27

4. GPGPU による可変レート多元 LDPC 符号の復号

パリティ検査行列は，図 3 のように，各列および各行を端点とし，非零の要素がある行および列を辺とする二部グラフを用いて表す方法がある．この二部グラフは Tanner グラフと呼ばれ，各列を表す端点を変数ノード，各行を表す端点をチェックノードという．各端点の間を，通信路を通した入力 s を元にした事後確率 $P_i(c_i = a|s)$，もしくは事後確率の比をメッセージとして伝播させることにより，復号処理を行う．本研究においては，多元 LDPC 符号の BP 法のうち，ガロア体でのフーリエ変換を用いた Sum-Product 法 (FFT Sum-Product 法)[6]，および Min-Max 法[8] について実装している．FFT Sum-Product 法はより厳密に事後確率を求める方法であり，誤り訂正能力により優れる．一方の Min-Max 法は Sum-Product 法の近似である．以下，次のような記号を用いて復号法を示す．

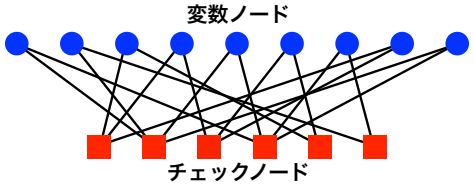

図 3 Tanner グラフの例: 9×6-行列

表 1 本章で用いる記号

名称	記号
ガロア体の要素の値	a
復号ループ回数	κ
H の非零要素	H_{ij}
入力される事後確率	$P_i(c_i = a\|s)$
変数ノード i からチェックノード j へのメッセージ値	$m_v(i \to j, a, \kappa)$
i 番目の変数ノードに接続するチェックノードの集合	$\mathcal{V}_i$
チェックノード j から変数ノード i へのメッセージ値	$m_c(j \to i, a, \kappa)$
j 番目のチェックノードに接続する変数ノードの集合	$\mathcal{C}_j$

FFT Sum-Product 法の計算処理は図 4 に示す．計算は以下の通りである．

(1) メッセージ入力:

$$m_v(i \to j, a, 0) = P(c_i = a|s)$$

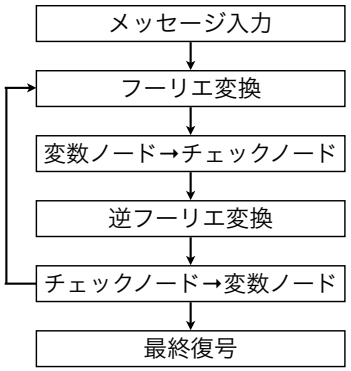

図 4 FFT Sum-Product 法の処理

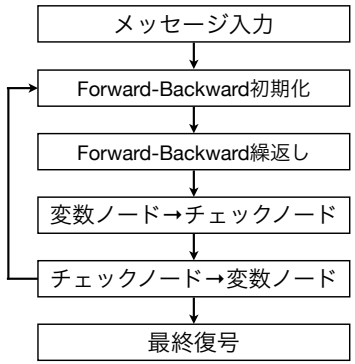

図 5 Min-Max 法の処理

(2) フーリエ変換: フーリエ変換後のメッセージの値を $\mathcal{M}_v$ とする．

$$\mathcal{M}_v(x \to j, \alpha, \kappa) = \sum_{a \in \mathrm{GF}(2^q)} m_v(x \to j, a, \kappa) \cdot (-1)^{\alpha \cdot a} \tag{4}$$

(3) 変数ノード→チェックノード: フーリエ領域におけるチェックノードのメッセージの値 $\mathcal{M}_c$ を求める．

$$\mathcal{M}_c(j \to i, a, \kappa + 1) = \prod_{x \in \mathcal{C}_j \backslash i} \mathcal{M}_v(x \to j, (H_{xj})^{-1} a, \kappa)$$

(4) 逆フーリエ変換:

$$m_c(j \to i, a, \kappa + 1) = \frac{1}{2^q} \sum_{\alpha \in \mathrm{GF}(2^q)} \mathcal{M}_c(j \to i, \alpha, \kappa + 1) \cdot (-1)^{(a \cdot \alpha)}$$

(5) チェックノード→変数ノード:

$$m_v(i \to j, a, \kappa) = m_v(i \to j, a, 0) \cdot \prod_{x \in \mathcal{V}_i \backslash j} m_c(x \to i, a, \kappa) \tag{5}$$

(6) 最終復号:

$$M_v(i, a) = m_v(i \to j, a, 0) \cdot \prod_{x \in \mathcal{V}_i} m_c(x \to i, a, \kappa)$$

$$c_i = \underset{\alpha \in \mathrm{GF}(q)}{\mathrm{argmax}} M_v(i, \alpha)$$

また，Min-Max 法の計算処理は図 5 に示す．計算は以下の通りである．

(1) メッセージ入力:

$$m_v(i \to j, a, 0) = \ln\left(\frac{P(c_i = a_{i_{\max}} \mid s)}{P(c_i = a \mid s)}\right)$$
$$a_{i_{\max}} = \underset{a \in GF(q)}{\arg\max}\ P(c_i = a \mid s)$$

(2) Forward-Backward 初期化: 各変数ノードごとに Forward 値 F と Backward 値 B を用意する.

$$\tilde{m}_c(j \to \delta_i, a) = m_v(\delta_i \to j, H_{\delta_i j}^{-1} a, \kappa)$$
$$F_1(a) = \tilde{m}_c(j \to \delta_1, a)$$
$$B_d(a) = \tilde{m}_c(j \to \delta_d, a)$$

(3) Forward-Backward 繰返し:

$$F_i(a) = \min_{\substack{a', a'' \in \mathrm{GF}(2^q) \\ a' \oplus a'' = a}} \left(\max(F_{i-1}(a'), \tilde{m}_c(j \to \delta_i, a''))\right)$$
$$B_i(a) = \min_{\substack{a', a'' \in \mathrm{GF}(2^q) \\ a' \oplus a'' = a}} \left(\max(B_{i+1}(a'), \tilde{m}_c(j \to \delta_i, a''))\right)$$

(4) 変数ノード→チェックノード:

$$m_c(j \to \delta_i, a, \kappa + 1) = \begin{cases} B_1(a) & \text{if } i = 1 \\ F_d(a) & \text{if } i = d \\ \min\limits_{\substack{a', a'' \in \mathrm{GF}(2^q) \\ a' \oplus a'' = -H_{\delta_i j} a}} \left(\max(F_{i+1}(a'), B_{i-1}(a''))\right) & \text{otherwise} \end{cases}$$

(5) チェックノード→変数ノード:

$$m_v(i \to j, a, \kappa) = m_v(i \to j, a, 0) + \left(\sum_{x \in \mathcal{C}_i \setminus j} m_c(x \to i, a, \kappa)\right) \tag{6}$$

(6) 最終復号:

$$M_v(i, a) = m_v(i \to j, a, 0) + \left(\sum_{x \in \mathcal{C}_i} m_c(x \to i, a, \kappa)\right)$$
$$c_i = \underset{a \in GF(q)}{\operatorname{argmin}} M_v(i, a)$$

2 つの復号法は，それぞれ図 4，図 5 の矢印で示す範囲を繰り返してメッセージを伝播させる．プログラムにおいては，各処理 (各図のブロックで示した単位) を GPU 上で 1 つのグリッドとして処理を順次行う．

GPGPU でのメモリアクセス例を図 6 および図 7 に示す．図 6 は，FFT Sum-Product 法および Min-Max 法における，チェックノードのメッセージを集約して変数ノードのメッセージを計算する処理 (式 (5)，(6)) の流れを表している．1 つのスレッドブロックごとに，Tanner グラフにおける 1 つの辺についての処理を行い，また 1 つのスレッドごとにガロア体の 1 つの要素の値について計算を行う．図における処理の順序は以下の通りである．

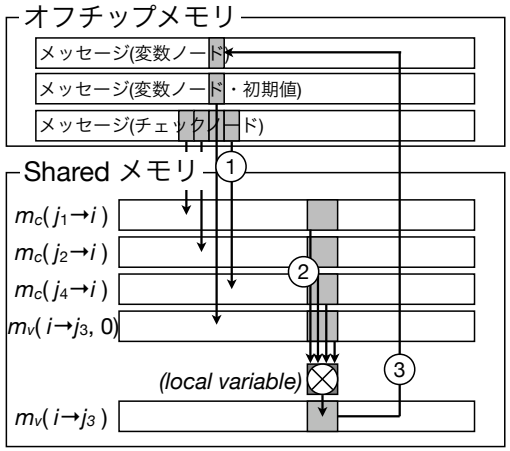

図 6 GPGPU による処理例 1: チェックノードから変数ノードへの伝播

(1) Global メモリから，チェックノードの番号が $\mathcal{V}_i \backslash j$ に属する，チェックノードのメッセージの値と，変数ノードのメッセージ初期値を 1 つのスレッドブロック分だけ Shared メモリに読み出す．Global メモリには，チェックノードのメッセージが連続して読み出せるように値を配置する．

(2) スレッドごとに，チェックノードのメッセージと変数ノードの初期値を計算 (FFT Sum-Product 法では乗算，Min-Max 法では加算) し，ローカル変数に保存する．

(3) 計算した新たな変数ノードのメッセージの値を Shared メモリにコピーし，1 つのスレッドブロックごとに Global メモリにコピーする．

このような流れで処理を行うことにより，オフチップメモリとキャッシュのメモリコピーを比較的大きなバイト数で行うことによりトランザクション数を減らすことができる coalesced memory access[16] が利用できる．また，有効にキャッシュを利用することができるため，計算を行う場合の読み出し時間を減少させることができるため，高速化が実現できる．

図 7 は FFT Sum-Product 法におけるガロア体でのフーリエ変換処理 (式 (4)) を示したものである．1 つのスレッドブロックに対して，1 つのガロア体の要素の値に対するメッセー

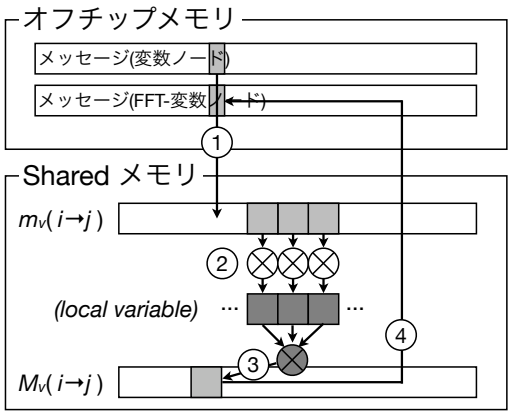

図 7 GPGPU による処理例 2: FFT

ジの値を計算することとし，処理の順序は以下の通りである．

(1) フーリエ変換を行う前のメッセージを，1 要素分だけ Global メモリから Shared メモリに読み込む．

(2) それぞれ，ガロア体の値ごとに正負を反転させ，スレッドごとに保存する．

(3) 1 要素分の正負を反転させた値を乗算し，他のスレッドで求めた値と足しあわせる．

(4) 求めた値を Global メモリにコピーする．

このように 1 つのスレッドブロックを細かく分けることにより，高速化を実現させた．

5. シミュレーション結果

符号化および復号を GPU 上で行ったシミュレーション結果は以下の通りである．シミュレーション条件は表 2 の通りである．

図 8 は符号化処理のスループットを示したものである．ベクトル計算により符号化を行うため，ガロア体の要素数に応じてスループットが伸び，最大で約 15Mbps のスループットを実現した．また，図 9 は，符号化処理を CPU によるプログラムと比較したものである．最大で CPU によるプログラムの 166 倍高速に動作している．

次に，復号を行った際のシミュレーションについて，図 10 は FFT Sum-Product 法のスループットを，また図 11 は Min-Max 法のスループットを示している．スループットは FFT Sum-Product 法が高く，最大で GF(32) において約 15Mbps のスループットを実現

表 2 シミュレーション条件

条件: GPU	
GPU プロセッサ	NVIDIA Tesla C1070
プロセッサコア数	512
プロセッサ動作速度	1.30GHz
条件: 符号 1	
ガロア体	$GF(2^m)$, $1 \leq m \leq 8$
符号長	16 - 8192 シンボル
符号化率	0.5
列重み	2
非零要素の決定法	ランダム

条件: CPU	
CPU プロセッサ	Intel Core i7 920
プロセッサ動作速度	3.20GHz
条件: 符号 2	
ガロア体	$GF(2^m)$, $1 \leq m \leq 7$
符号長	100 - 8100 シンボル
符号化率	0.05 - 0.95
列重み	2
非零要素の決定法	形式 2 に従う

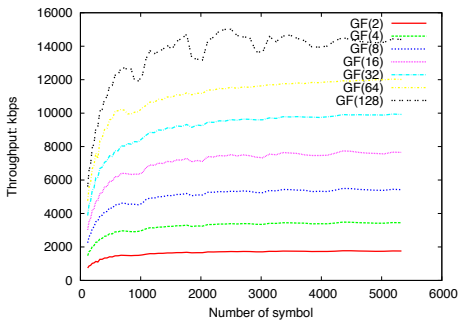

図 8 符号化スループット (符号 2，符号化率 0.5)

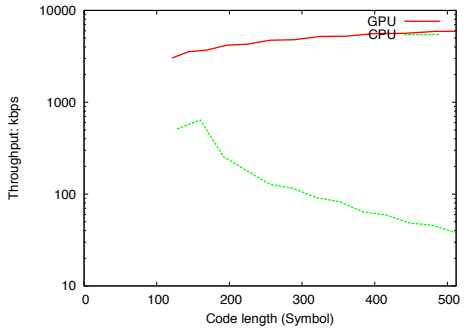

図 9 符号化スループット: CPU との比較 (符号 2，GF(16)，符号化率 0.5)

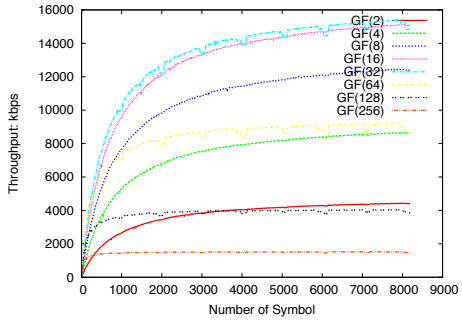

図 10 復号スループット: FFT Sum-Product 法 (符号 1)

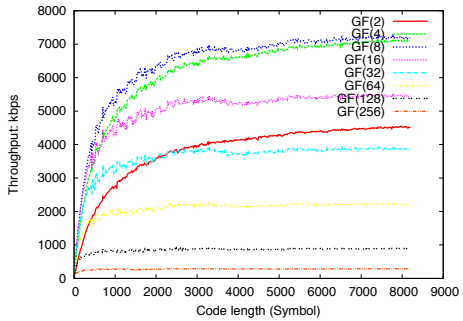

図 11 復号スループット: Min-Max 法 (符号 2)

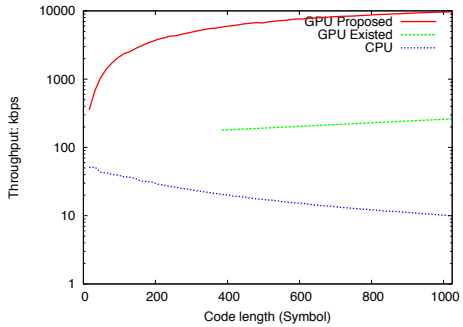

図 12 復号スループット: CPU との比較 (符号 1，GF(16))

している．一方，Min-Max 法は最大でも約 7Mbps のスループットにとどまった．また，図 12 は，本研究におけるプログラムと CPU によるプログラム，また既存の GPU によるプログラム[14] との比較を示したものである．CPU によるプログラムと比べ最大で 972 倍，また既存の GPU によるプログラムと比べ最大で 36 倍高速に動作している．

6. ま と め

本研究では，多元 LDPC 符号の高速なシミュレーション環境を整備するため，GPU による符号化，および復号プログラムを実装し，CPU によるプログラムと比較を行った．符号化プログラムは GPGPU に有利に働くベクトル計算により実装したほか，復号プログラムは既存のアルゴリズムから並列計算に優れる処理を検討した．その結果，CPU のプログラムと比べ，符号化は最大で約 160 倍，復号は最大で約 950 倍高速に動作することがわかった．また，符号化と復号について，どちらも最大で約 15Mbps の速度で動作させることが可能となった．

参 考 文 献

1) Gallager, R.: ”Low Density Parity Check Codes”, Monograph, M.I.T. Press (1963).
2) Davey, M. C. and MacKay, D. J. C.: ”Low Density Parity Check Codes over GF(q)”, ITW 1998, Killarney, Ireland, pp. 70–71 (1998).
3) Yazdani, M. R. and Banihashemi, A. H.: ”On Construction of Rate-Compatible Low-Density Parity-Check Codes”, IEEE Communications Letters, Ver. 8, No. 3, pp. 159–161 (2008).
4) Declercq, D. and Fossorier, M. : ”Decoding Algorithms for Nonbinary LDPC Codes Over GF(q)”, IEEE Transactions on Communication, Vol. 55, No.4, pp. 633–643 (2007).
5) Richardson, T. J.: ”Efficient Encoding of Low-Density Parity-Check Codes”, IEEE Transactions on Information Theory, Vol.47, No.2, pp. 638–656 (2001).
6) Kasai, K. and Sakaniwa, K.: ”Fourier Domain Decoding Algorithm of Non-Binary LDPC codes for Parallel Implementation”, IEICE Trans. Fundamentals, Vol.E93-A, No.11, (2010).
7) Chen, J., Tanner, R. M. , Jones, C. and Li, Y.: ”Improved Min-Sum Decoding Algorithms for Irregular LDPC Codes, ” International Symposium on Information Theory, 2005., pp. 449–453 (2005).
8) Satin, V.: ”Min-Max decoding for non binary LDPC codes, ” International Symposium on Information Theory, 2008., pp. 960–964 (2008).
9) Moreira, J. C. and Farrell, P. G.: ”Essentials of Error-Control Coding”, Wiley (2006).
10) Richardson, T. and Urbanke, R.: ”Modern Coding Theory”, Cambridge University Press, Preliminary version (2007).
11) Baldi, M. , Cancellieri, G. and Chiaraluce, F.: ”Variable Rate LDPC Codes for Wireless Applications”, International Conference on Software in Telecommunications and Computer Networks, 2006., pp. 301–305 (2006).
12) Falcao, G. , V. Silva and Sousa, L.: ”How GPUs can outperform ASICs for fast LDPC decoding”, ICS ’09 Proceedings of the 23rd international conference on Supercomputing, pp. 390–399 (2009).
13) Falcao, G., Sousa, L. and Silva, V.: ”Massive Parallel LDPC Decoding on GPU”, PPoPP ’08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 83–90 (2008).
14) 藤坂祐介, ”並列計算を利用した多元 LDPC 符号の高速復号法”, 学士論文, 東京工業大学工学部 (2010).
15) ”OpenCL - The open standard for parallel programming of heterogeneous systems”, available from ⟨http://www.khronos.org/opencl/⟩ (accessed 2012-02-27).
16) NVidia, inc. ”NVIDIA CUDA(TM) NVIDIA CUDA C Programming Guide”, Version 4.0 (2011).