Vol.2015-HPC-148 No.10
2015/3/2

離散ウェーブレット変換のGPU実装における入力の上書きによるメモリ使用量削減

生澤 拓也[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本論文では，大規模データに対する高速な離散ウェーブレット変換の実現を目的として，GPU（Graphics Processing Unit）上で少ないメモリ使用量および高いメモリ参照効率を両立する並列手法を提案する．提案手法は，入力領域を出力領域と統合し，さらにデータサイズ n よりも小さな作業用領域のみを用い，メモリ使用量を節約する．この統合は，読み込む前のデータに対する上書きを誘発し，そのままでは並列処理できない．そこで，複数の要素からなる小さなチャンクごとにデータをあらかじめ並べ替えたうえで，変換を並列処理する．この並べ替えにより，データ依存に関して独立な計算を構成できるだけでなく，連続領域への同時参照のみで変換を実現できる．さらに，前処理となる並べ替えにおいて，互いにデータ依存のある一連の代入を巡回置換とみなせ，それらの複雑な移動先番地を，n よりも短い長さの数列から計算する GPU コードを記述できる．

キーワード：離散ウェーブレット変換，リフティング方式，CUDA，GPU

Reducing Memory Usage of Discrete Wavelet Transform by Overwriting the Input on a GPU

Takuya Ikuzawa[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, to realize fast discrete wavelet transformation for large-scale data, we present a parallel method capable of achieving both a small amount of memory usage and highly efficient memory accesses on a graphics processing unit (GPU). Our method saves the amount of memory usage by not only unifying input region with output region but also using a working region smaller than data size n.This unification induces overwrites to unfetched data, avoiding naive parallelization of the transformation. To deal with this issue, our method rearranges each small chunk of data elements, and then processes the transformation in parallel. Our data rearrangement scheme makes it possible to not only produce data-independent tasks in the transformation but also realize the transformation with only simultaneous access to contiguous memory region. In addition, a sequence of data-dependent assignments needed for rearrangement can be regarded as a cyclic permutation. This interpretation allows us to write GPU code that computes a sequence of complicated memory addresses from a sequence of numbers shorter than n.

Keywords: Discrete wavelet transform, lifting scheme, CUDA, GPU

1. はじめに

離散ウェーブレット変換（DWT：Discrete Wavelet Transform）とは，ウェーブレットと呼ばれる有限長波形を基底とする周波数解析手法であり，入力を低域成分および高域成分に分離して出力する．入力が n 個の要素からなるとき，同様の解析手法であるフーリエ変換の時間計算量 $\mathcal{O}(n \log n)$ と比較して，DWT の計算量は $\mathcal{O}(n)$ で済む．DWT は JPEG 2000 などのデータ圧縮 [1]，地球観測衛星データの復号 [2] および地理データのクラスタリング解析 [3] などに応用されている．これらで多用される多重解

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

像度解析は，低域成分および高域成分に対して DWT を再帰的に適用する．したがって，時間制約のある応用を実時間処理するために，DWT の高速化が求められている．

DWT の高速化手段は，アルゴリズムの改良およびアクセラレータの活用によるものに分類できる．前者としてリフティング方式 [4] があり，その計算量は従来のフィルタバンク方式 [5] と比べておよそ半分である．さらに，リフティング方式は in-place 処理が可能であり，フィルタバンク方式と比べてメモリ使用量を半減できる．ここで，in-place 処理とは，入力の他に必要とするメモリ領域のサイズが $\mathcal{O}(1)$ であるものを指す．

一方，後者としては FPGA（Field Programmable Gate Array）[6] や GPU（Graphics Processing Unit）[7] による実装がある．これらのうち，GPU は高いピークメモリ帯域幅を持つ市販品であり，メモリ参照が性能ボトルネックとなる DWT の高速化を期待できる．

Laan ら [8] は，リフティング方式を GPU 向けの統合開発環境 CUDA（Compute Unified Device Architecture）[9] 上に実装している．この既存手法は，ビデオメモリの参照効率を高めるために，低域成分（および高域成分）を連続領域に格納する．結果，GPU の得意とするメモリ参照パターン，すなわち連続領域への同時参照により，GPU の実効メモリ帯域幅を最大化できる．ただし，既存手法は入力領域を出力領域と区別しているため，リフティング方式の特長である in-place 処理を損ない，メモリ使用量が多い．したがって，大規模な入力に対しては，その一部を CPU へ退避させる必要があり，CPU・GPU 間のデータ転送が全体性能を低下させる．なお，低域成分および高域成分を交互に格納すれば，in-place 処理を実現できる．ただし，多重解像度解析において再帰が深くなるたびにメモリ参照時のストライド幅が 2 倍に増大し，不連続領域への参照が実行効率を低下させる．

そこで本論文では，大規模データに対する高速な DWT の実現を目的として，GPU 上で少ないメモリ使用量および高い参照効率を両立する並列手法を提案する．提案手法は，入力領域を出力領域と統合し，さらに入力サイズ n よりも小さな作業用領域のみを用い，メモリ使用量を節約する．この統合は，読み込み前の要素に対する上書きを誘発し，そのままでは並列処理できない．そこで，複数の要素からなる小さなチャンクごとに入力をあらかじめ並べ替えたうえで，変換を並列処理する．この並べ替えにより，データ依存に関して独立な計算を構成できる．なお，前処理となる並べ替えにおいて，データ依存のあるものはグループ化し，グループ内の一連の移動を逐次処理する．したがって，同時に移動できるチャンクはグループの数だけ存在し，それらのチャンクが含む要素の数だけの並列性がある．

さらに，提案手法は連続領域への同時参照のみで変換を実現できる．つまり，GPU の得意な参照パターンを用いてチャンクを生成しさえすれば，以降の移動も連続領域への同時参照で済む．ここで解決すべき課題は，GPU 上の各スレッドが n よりも小さな情報から一連の複雑な移動先番地を計算できる仕組みを確立することである．提案手法は，グループ内の一連の移動が巡回置換として表現できることに着目し，それらの移動先番地を計算するための種となる数列をあらかじめ CPU 上で計算する．この数列の長さは n よりも短いため，入出力領域を区別するときと比較してメモリ領域を節約できる．得られた数列を GPU 側へ転送し，それらを基に GPU 上の並列スレッドは一連の移動を実現する．

以降では，まず 2 章で関連研究を紹介し，3 章でリフティング方式についてまとめる．次に，4 章で提案手法を説明し，5 章で評価実験の結果を示す．最後に 6 章で本論文をまとめる．

2. 関連研究

Wong ら [10] は，グラフィクス API の 1 つである OpenGL[11] を用い，フィルタバンク方式を GeForce 7800 GTX 上に実装した．グラフィクス処理に特化した OpenGL は，汎用処理の観点からはプログラミングに関する制約が強いため，リフティング方式の実装は見送られている．OpenGL を用いたリフティング方式の実装は，Tenllado ら [12] が示したが，複雑なデータ依存が原因で，フィルタバンク方式の方が高速であった．

Franco ら [13] は，柔軟なプログラミングを可能とする CUDA を用い，フィルタバンク方式を実装した．この実装は，Tesla C870 上で 8192×8192 画素からなる画像を 33 ミリ秒程度で処理し，このときの変換スループットは 2.0 Gpixel/s に達する．しかし，162 ミリ秒を要する CPU・GPU 間のデータ転送が全体性能を律速していて，データ転送を含めた変換スループットは 0.3 Gpixel/s に留まる．また，入出力領域を区別している．

Laan ら [8] は, CUDA によるリフティング方式を GeForce 8800 GTX 上に実装し，4096×4096 画素の画像に対して 1.1 Gpixel/s の変換スループットを達成した．この性能は，Tenllado らの OpenGL 版（0.4 Gpixel/s）よりも高い．また，CUDA では，フィルタバンク方式よりもリフティング方式の方が 2 倍高速であることを示した．Kucis ら [14] は，Laan らの手法を基に，同期の削減により GeForce GTX 580 上で 30%の性能向上を果たしている．このときの GPU 上の変換スループットは 8.3 Gpixel/s に達する．しかし，CPU・GPU 間のデータ転送時間を含めた場合，変換スループットは 0.4 Gpixel/s に低下し，モバイル向け CPU（Core 2 Quad Q9000）が得た 0.7 Gpixel/s よりも低速である．

Sharma ら [15] は，OpenCL[16] を用いて GeForce GTX 285 上にリフティング方式を実装した．OpenCL は標準仕様であるため，彼らの実装は CUDA 互換でない GPU やマ

ルチコア CPU 上で動作する．この実装は 4096×4096 画素からなる画像を，およそ 0.8 Gpixel/s の変換スループットで処理できたが，CUDA 版の 1.1 Gpixel/s と比べて低速であった．

3. リフティング方式による離散ウェーブレット変換

DWT の入力となる信号を $x_0, x_1, \ldots, x_{n-1}$ とする．ここで，n は信号の標本数とし，2 のべき乗である．また，出力となる信号のうち，低域成分を $c_0, c_1, \ldots, c_{n/2-1}$ とし，高域成分を $d_0, d_1, \ldots, d_{n/2-1}$ とする．このとき，低域成分 c_t および高域成分 d_t $(0 \le t < n/2)$ はそれぞれ式 (1) および (2) で与えられる．

$$c_t = x_{2t} + \sum_{i=-k_0}^{k_0-1} u_i d_t \tag{1}$$

$$d_t = x_{2t+1} - \sum_{j=-k_1+1}^{k_1} p_j x_{2(t+j)} \tag{2}$$

ここで，定数 u_i $(-k_0 \le i < k_0)$ および p_j $(-k_1 < j \le k_1)$ は，基底となるウェーブレットに依存する係数である．

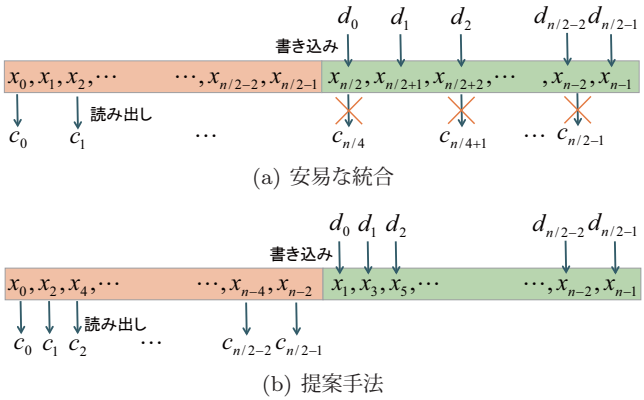
式 (1) および (2) を in-place 処理するためには，d_t を x_{2t+1} に出力したのちに，c_t を x_{2t} に出力すればよい．つまり，$c_0, d_0, c_1, d_1, \ldots, c_{n/2-1}, d_{n/2-1}$ のように，低域成分 c_t および高域成分 d_t を交互に出力するサイクリック方式を採用すればよい．さらに，高域成分を計算するカーネル関数の完了後に低域成分を計算するカーネル関数を呼び出せば，バリア同期を持たない GPU 上で並列処理できる．しかし，サイクリック方式は実行効率の観点で，再帰的な変換を必要とする多重解像度解析との相性が悪い．低域成分に対する DWT は，再帰のたびにメモリ参照時のストライド幅を 2 倍に増大させてしまい，GPU の高いピークメモリ帯域幅を生かせない．

3.1 Laan らの既存手法

Laan ら [8] は，サイクリック方式の課題を解決した．具体的には，$c_0, c_1, \ldots, c_{n/2-1}, d_0, d_1, \ldots, d_{n/2-1}$ のように，低域成分および高域成分のそれぞれを連続領域に出力するブロック方式を用い，メモリの参照効率を高めている．

しかし，この既存手法は入力領域を出力領域と区別しているため，メモリ使用量が多く，リフティング方式の特長である in-place 処理を損なう．入出力領域を区別する理由は，式 (1) および (2) に存在するデータ依存にある．仮に，入出力領域を安易に統合したとすると，ブロック方式は d_t $(0 \le t < n/2)$ を $x_{t+n/2}$ に出力したのちに（**図 1**(a)），c_t を x_t に出力する．この場合，d_t を出力した時点で，c_t の計算が必要とする偶数番目の要素の一部，すなわち式 (2) の $x_{2(t+j)}$ が消失してしまう．

この消失を防ぐためには，これらデータ依存のある計算



(a) 安易な統合

(b) 提案手法

図 1 入出力領域を統合した場合のメモリ参照の様子

を逐次処理する，あるいはすべての要素 $x_0, x_1, \ldots, x_{n-1}$ を他のメモリ領域へ読み込んだのちに計算結果を書き出すしかない．前者は，結果的に式 (2) の逐次処理になってしまい，後者は入出力領域を区別することに他ならない．

4. 提案手法

提案手法は，GPU の持つ高いピークメモリ帯域幅を活用するために，Laan ら [8] と同様にブロック方式を採用する．さらに，単一の入出力領域に対して変換を並列処理するために，要素 $x_0, x_1, \ldots, x_n$ を，あらかじめ $x_0, x_2, \ldots, x_{n-2}, x_1, x_3, \ldots, x_{n-1}$ のように並べ替える（図 1(b)）．この際，一連の要素を B バイトごとのチャンクとみなし，チャンク単位で並べ替える．この並べ替えは，複雑に絡み合った，チャンク間のデータ依存を整理でき，単一の入出領域に対する並列処理を可能とする．並べ替え後，式 (1) および (2) の計算を処理する．

なお，チャンクサイズ B は 128 の倍数とする．この理由は，GPU におけるメモリトランザクションが 128 バイト単位で処理されるためである [9]．結果，提案手法の並べ替えは連続領域に対する参照のみで実現でき，高いメモリ参照効率を実現できる．しかし，既存手法 [8] と比べて並べ替えのためにメモリ参照量が増えてしまう．また，データとは別に，十分に小さいが n に依存する長さの数列を必要とするため，in-place 処理は実現できていない．

4.1 並べ替えによるデータ依存の整理

提案手法の並べ替えは，以下に示す手順からなる．

(1) チャンクの生成：入力 $x_0, x_1, \ldots, x_{n-1}$ に対し，チャンク 2 個分（$2B$ バイトごと）の区間を考え，区間内の要素の並べ替えにより，偶数番目の要素のみからなる偶数チャンクおよび奇数番目の要素のみからなる奇数チャンクを生成する（**図 2**）．この段階では，偶数チャンクおよび奇数チャンクは交互に格納されている．以降では，チャンク 1 つあたりの要素数を b とし，偶数チャンク（あるいは奇数チャンク）の数を m $(= \lceil n/b \rceil)$

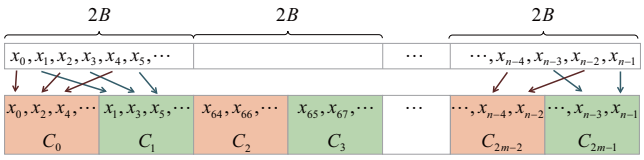


図 2 チャンクの生成

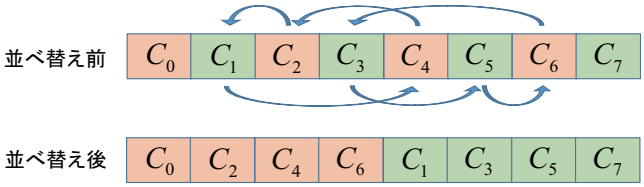


図 3 チャンクの並べ替え（$m = 4$ の場合）

Algorithm 1 チャンク生成

Input: n 個の要素 $x_0, x_1, \ldots, x_{n-1}$
Output: チャンク単位で並べ替え済の要素 $x_0, x_2, \ldots, x_{n-1}$

```
1: i ← スレッド ID
2: j ← スレッドブロック ID
3: w ← x_{2bj+i}                    ▷ 担当要素をレジスタにコピー
4: if i is even then                ▷ 偶数チャンク生成
5:     x_{2bj+i/2} ← w
6: else                             ▷ 奇数チャンク生成
7:     x_{2bj+b+⌊i/2⌋} ← w
8: end if
```

とする．このとき，チャンクの数は $2m$ となる．

(2) チャンクの並べ替え：$2m$ 個のチャンクを並べ替え，m 個の偶数チャンクのあとに m 個の奇数チャンクを格納する（**図 3**）．つまり，$2i$ $(0 \le i < m)$ 番目および $2i + 1$ 番目にあるチャンクは，並べ替え後にそれぞれ i 番目および $i + m - 1$ 番目のチャンクに移動する．

まず，チャンク生成のアルゴリズムを Algorithm 1 に示す．このアルゴリズムは，GPU 上の各スレッドが処理すべき内容を記述したものであり，$2b$ 個のスレッドからなるスレッドブロックがチャンク 2 個分の区間を担当する．各スレッドは，自身が担当する要素をグローバルメモリからレジスタ w にコピーし（3 行目），自身のスレッド ID を基に，グローバルメモリ上の適切なメモリ番地へ書き戻す（4〜8 行目）．このとき，偶数のスレッド ID を持つスレッドが x_{2bj}〜$x_{2bj+b-1}$ を，奇数のスレッドが x_{2bj+b}〜$x_{2bj+2b-1}$ を参照するため，連続領域への参照が実現する．

次に，チャンクの並べ替えは l 個の巡回置換の積 $\sigma_0 \circ \sigma_1 \circ \ldots \circ \sigma_{l-1}$ で表現できる．例えば，i 番目のチャンクと j 番目のチャンクの交換を互換 $(i\ j)$ で表せば，$m = 4$ に対する並べ替えは巡回置換の積 $(1\ 4\ 2) \circ (3\ 5\ 6)$ で表せる（図 3）．このとき，データ依存を持つ一連のチャンクの移動が 1 つの巡回置換を構成していて，それらは並列処理できない．しかし，先の例での $(1\ 4\ 2)$ および $(3\ 5\ 6)$ のように，異なる巡回置換は独立に処理できる．したがって，提案手法は各巡回置換に対して 1 つのスレッドブロックを

Algorithm 2 巡回置換に基づくチャンクの並べ替え

Input: 並べ替え前のチャンク $C_0, C_1, \ldots, C_{2m-1}$，チャンクの数 $2m$ および巡回置換の代表元 $p_0, p_1, \ldots, p_{l-1}$
Output: 並べ替え後のチャンク $C_0, C_1, \ldots, C_{2m-1}$

```
1: j ← スレッドブロック ID
2: W ← C_{p_j}                      ▷ 並列に移動
3: i ← p_j
4: while 2i mod (2m − 1) ≠ p_j do
5:     C_i ← C_{2i mod (2m−1)}      ▷ 並列に移動
6:     i ← 2i mod (2m − 1)
7: end while
8: C_i ← W                          ▷ 並列に移動
```

割り当てる．スレッドブロックは，割り当てられた巡回置換を構成する一連の移動を逐次処理する．ただし，チャンクそのものは複数の要素を含んでいるため，それらの要素はスレッドブロック内の b 個のスレッドを用いて並列に移動できる．

次に解決すべき課題は，n に対して一意に定まる各巡回置換を特定することである．そのための定理を以下に示す．

定理 1. 手順 (2) におけるチャンクの並べ替えを表現する任意の巡回置換 σ_j $(0 \le j < l)$ に対し，その大きさを s としたとき，$s \le \log 2m$ が成り立つ．

証明. 任意の巡回置換を $(a_0\ a_1\ a_2\ \ldots\ a_{s-1})$ とする．ここで，任意の $0 \le k < s$ に対して $0 \le a_k < 2m$ である．このとき，式 (3) が成り立つ．

$$a_k = \begin{cases} a_{k+1}/2, & \text{if } a_{k+1} \text{ is even,} \\ \lfloor a_{k+1}/2 \rfloor + m, & \text{otherwise.} \end{cases} \tag{3}$$

ここで，a_k および a_{k+1} を $\log 2m$ ビットの 2 進数で表すと，式 (3) より a_{k+1} を 1 ビット右循環シフトしたものが a_k となる．したがって，a_k の値に関係なく，$\log 2m$ ビット右循環シフトすれば，a_k に戻ることから s が存在する．また，$a_0 = a_{\log 2m}$ が成り立つことから $s \le \log 2m$ である． □

式 (3) より，式 (4) が成り立ち，巡回置換の代表元となる a_0 を特定できれば各巡回置換を計算できる．すなわち，各チャンクの移動先を決定できる．

$$a_{k+1} = 2a_k \bmod (2m - 1) \tag{4}$$

Algorithm 2 に，手順 (2) にしたがいチャンクを並べ替えるアルゴリズムを示す．このアルゴリズムは，巡回置換の数 l ならびに各巡回置換 σ_j $(0 \le j < l)$ に対する代表元 p_j が入力として与えられることを前提とし，l 個のスレッドブロックで処理する．まず，スレッドブロック j は，代表元に対応するチャンクをレジスタ上の作業領域にコピーする（2 行目）．次に，巡回置換 σ_j にしたがい，さきほど退避させたメモリ領域へチャンクへ移動させる（5 行目）．以降，チャンクの移動先が代表元に戻るまで，チャンクの

Algorithm 3 巡回置換ごとの代表元の事前計算

Input: チャンク数 $2m$
Output: 巡回置換の数 l および巡回置換の代表元 $p_0, p_1, \ldots, p_{l-1}$

```
1: 大きさ 2m の配列 X を 0 で初期化
2: q ← 0
3: for i ← 0 to 2m − 1 do                    ▷ p_q の算出
4:     if X_i = 0 then                        ▷ 未登録
5:         p_q ← i
6:         X_i ← 1
7:         j ← 2i mod (2m − 1)  ▷ i と同じ巡回置換に含まれる値をチェック
8:         while j ≠ i do
9:             X_j ← 1
10:            j ← 2j mod (2m − 1)
11:        end while
12:        q ← q + 1
13:    end if
14: end for
```

移動を繰り返す（4〜7 行目）．これらの並べ替えが必要とする作業領域の大きさは，巡回置換 1 つあたりチャンク 1 つ分の大きさ B であり，入出力領域を区別する場合と比べてメモリ使用量を節約できる．また，チャンク 1 つあたりに必要なメモリ参照は読み書き 1 回ずつであり，並び替えに要するメモリ参照量は最適である．

4.2 移動先番地の計算

与えられた n に対して，巡回置換の数 l ならびに巡回置換 σ_j に対する代表元 p_j $(0 \le j < l)$ を，GPU 上で計算できれば，並べ替え処理を含めて変換の全体を in-place 処理できる．しかし，並列に動作するスレッドがすべての代表元を独立に計算する方法は明らかでない．

そこで，提案手法では並べ替えの前処理として，CPU 上で $p_0, p_1, \ldots, p_{l-1}$ を計算する．Algorithm 3 に，一連の代表元を計算するアルゴリズムを示す．このアルゴリズムは，巡回置換 σ_j を構成する元を順番に計算し，それらの最小値を p_j として登録していく．巡回置換ごとに元を計算しているため，過去に出現した元を記憶し，重複を避けるための配列 X を用意している．

なお，一連の代表元 $p_0, p_1, \ldots, p_{l-1}$ は，チャンクを生成する前にビデオメモリ上の定数メモリへ転送しておく．現在の GPU アーキテクチャでは，定数メモリの容量は 64 KB に過ぎないが，ビデオメモリが枯渇しない程度の n に対しては十分な大きさである（5.1 節で後述）．

5. 評価実験

提案手法を評価するために，メモリ使用量および実行時間に関して提案手法を Laan らの既存手法 [8] と比較した．実験には，$N \times N$ 画素の 2 次元画像を用い，そのサイズ N は 2K〜32K とした．各画素は 4 バイトの値を持つ．また，ウェーブレットとして Deslauriers-Dubuc (13,7)[17] を用いた．**表 1** に，実験環境を示す．

表 1 実験環境

項目	仕様
CPU	Intel Core i7-3770K
主記憶容量	16 GB
OS	Windows 7 Professional 64bit
GPU	NVIDIA GeForce GTX 680
ビデオメモリ容量	2 GB
CUDA バージョン	6.5
ドライババージョン	347.25

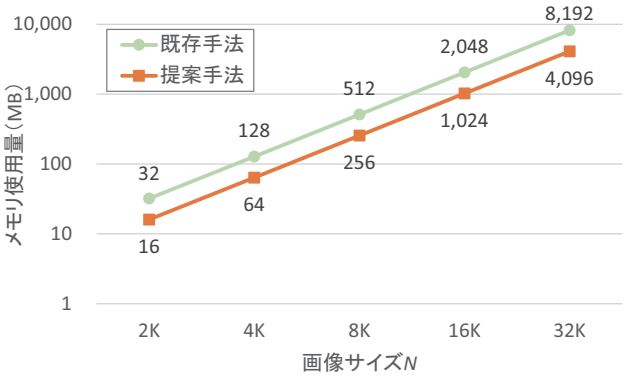


図 4 画像サイズ N ごとのメモリ使用量の比較

5.1 メモリ使用量

図 4 に，提案手法および既存手法のメモリ使用量を示す．提案手法は入出力領域の統合により，既存手法に対してメモリ使用量を半減できている．これにより，$N = 16$K までの画像を，分割することなく変換できる．一方，入出力領域を区別する既存手法は $N = 8$K に留まる．なお，ビデオメモリは画面表示のためのフレームバッファを保持しているため，その容量のすべてを変換対象の画像に割り当てることはできないことに注意されたい．

次に，一連の代表元を格納するために必要な作業領域のサイズ α を調べた．**図 5** に示すように，α は N に対して十分に小さく，データサイズが 4 GB に達する大きな画像（$N = 32$K）に対しても 3 KB 程度である．このように，提案手法は in-place 処理を実現できてはいないが，現実的な n に対して追加で必要とする作業領域は十分に小さい．

5.2 実行効率

DWT の性能はメモリ参照に律速されるため，その実行効率を実効メモリ帯域幅に関して評価した．**表 2** に，$N = 8$K のときのチャンク生成，チャンク並べ替えおよびリフティング処理における実行効率 $E = M/T/P$ を示す．ここで，M，T および P はそれぞれビデオメモリ参照量，カーネル実行時間および GPU のピークメモリ帯域幅を表す．実験で用いた GeForce GTX 680 では，P は 192.2 GB/s である．

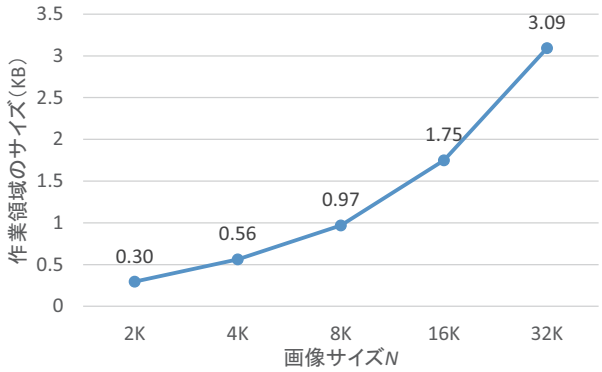


図 5 代表元を格納するために必要な作業領域のサイズ

表 2 実効メモリ帯域幅に基づく実行効率（%）

処理の対象	既存手法	提案手法
チャンク生成	—	69.1
チャンク並べ替え	—	73.2
リフティング処理	72.4	86.7
全体	72.4	78.4

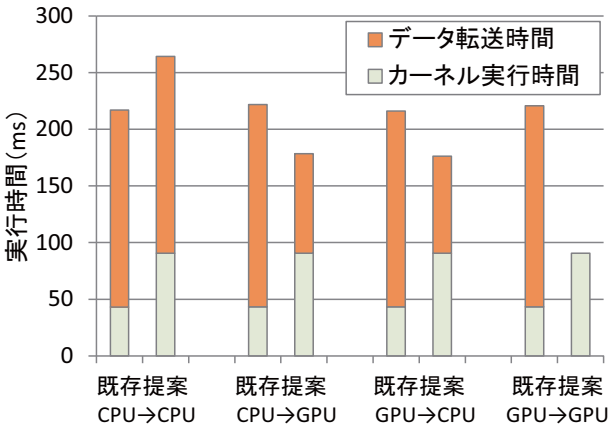
表 3 メモリ参照量（B）

処理の対象	メモリ参照量
チャンク生成	$4N^2$
チャンク並べ替え	$4N^2$
リフティング処理	$4(1+h/r)N^2$

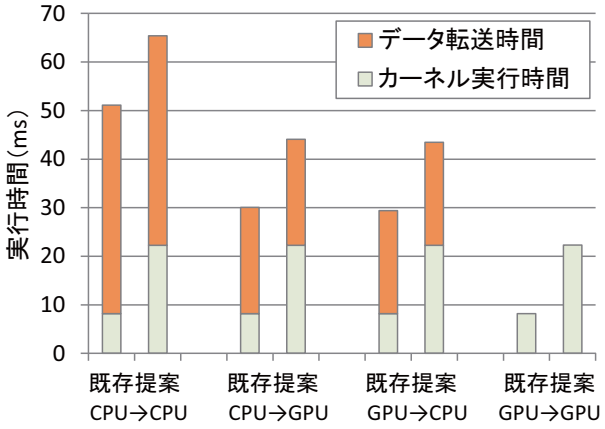
表 2 より，提案手法の実行効率は 78.4%に達し，既存手法の 72.4%よりも高い．特に，連続領域への参照のみで前処理を実現しているため，要素の並べ替えをしているにも関わらず，およそ 70%の高い実行効率を達成している．

なお，メモリ参照量 M は以下の手順で算出した．簡単のため，N 個の要素からなる 1 次元データの場合について考える．まず，チャンク生成では，区間ごとにビデオメモリを 1 回ずつ読み書きするため，その参照量は $2N$ となる．同様に，チャンクの並べ替えについても，その参照量は $2N$ である．次に，リフティング処理では，ある要素の計算は周囲の要素を必要とするため，各スレッドブロックは自身の担当領域だけでなく袖領域を読み込む必要がある．担当領域および袖領域が含む要素数をそれぞれ r および h とすると，N/r 個のスレッドブロックが $r+2h$ 個の要素を読み込む．したがって，リフティング処理のメモリ参照量は $(r+2h)N/r+N=2(1+h/r)N$ となる．なお，実験で用いたウェーブレット Deslauriers-Dubuc (13,7) では $r=6$ である．最後に，2 次元データに対しては，行方向に対する変換の後，列方向に対する変換を処理しているため，各処理のメモリ参照量は**表 3** のようになる．

なお，表 3 が示すように，提案手法の欠点はメモリ参照量が増大することである．リフティングのみを処理する既存手法と比べて，メモリ参照量はおよそ 3 倍となる．



(a) 画像サイズ $N=16$K（データサイズ 1024 MB）



(b) 画像サイズ $N=8$K（データサイズ 256 MB）

図 6 入出力領域の格納場所を変えたときの実行時間（非パイプライン版）

5.3 非パイプライン版の性能

多くの応用において，DWT は一連の処理の一部に過ぎない．したがって，変換の前後における処理を含めて GPU 上で加速することもある．そこで，入出力領域の格納場所として，主記憶（CPU）だけでなくビデオメモリ（GPU）を指定したときの実行時間を計測した．以降では，入力領域および出力領域の格納場所をそれぞれ x および y とし，$x \to y$ $(x,y \in \{\mathrm{CPU},\mathrm{GPU}\})$ と表す．

図 6(a) に，$N=16$K に対する提案手法および既存手法の実行時間を示す．なお，既存手法はそのままではメモリ不足に陥り実行に失敗するため，画像を分割して各々を順に実行している．一方，提案手法は画像を分割していない．また，いずれの手法も非パイプライン版である．

入出力領域が共に GPU 上にある場合（GPU→GPU），CPU・GPU 間のデータ転送を必要としない提案手法は，既存手法よりも 2.4 倍高速である．ただし，提案手法は前処理が必要であるため，既存手法と比べて 2.1 倍のカーネル実行時間を費やしている．なお，入出力領域のいずれかが GPU 上にある場合（GPU→CPU および CPU→GPU）も，提案手法は 1.2〜1.3 倍高速である．これらの場合，両

手法ともにデータ転送が必要である．しかし，既存手法は変換中のデータがビデオメモリを専有してしまうため，残りのデータを CPU 側に退避しておく必要がある．そのためのデータ転送時間が追加のオーバヘッドになり，提案手法の方が高速であった．最後に，入出力領域が共に CPU 上にある場合（CPU→CPU），前処理を必要とする提案手法は 18%低速である．ただし，5.4 節で後述するように，このオーバヘッドはパイプライン化により隠蔽できる．

最後に，両手法ともに画像の分割を必要としない，小さな画像サイズ $N = 8\text{K}$ を用いて実行時間を計測した（図 6(b)）．提案手法は，既存手法と比べて 1.3〜2.7 倍の実行時間を費やしている．分割の必要がない小さなデータに対しては，既存手法は提案手法と同様のデータ転送時間で変換を完了できる．したがって，前処理に起因するオーバヘッドが提案手法の実行時間を増大させていた．なお，提案手法は既存手法と比べて 2.7 倍のカーネル実行時間を要していた．この倍率は，$N = 16\text{K}$ のときの 2.1 倍と比べて増大している．この理由は，大きな画像に対して既存手法が実行効率を低下させたためである．

5.4 パイプライン版の性能

入出力領域が主記憶上にあることを前提として，画像サイズ N を変えながら，提案手法および既存手法の変換スループットを計測した（**図 7**）．なお，CPU・GPU 間のデータ転送が変換の性能ボトルネックになり得るため，CUDA ストリーム [9] を用いてデータ転送を GPU 上の計算とオーバラップするパイプライン版を用意した．また，いずれの手法もパイプライン処理のために画像を分割している．

パイプライン版において，提案手法は既存手法とほぼ同等の変換スループットを実現している．一方，非パイプライン版では，前処理のために既存手法よりも変換スループットが 23%低い．このように，提案手法が必要とする前処理は，GPU 上の変換スループットを低下させてしまうが，ビデオメモリに格納できないほどの大規模データに対してはオーバラップにより隠蔽できる．

6. まとめ

本論文では，大規模データに対する高速な離散ウェーブレット変換の実現を目的として，リフティング方式の CUDA 実装において少ないメモリ使用量および高いメモリ参照効率を両立する並列手法を提案した．提案手法は，入力領域を出力領域と統合したうえで，データサイズ n よりも小さな作業領域のみを用い，メモリ使用量を節約する．単純な統合は，計算結果の上書き時に未処理のデータを消去してしまう．そこで，そのような消去を回避するために，あらかじめデータを並べ替え，小さなチャンクに分割し，チャンク単位の移動を反復する．これらの処理は，いずれも連続領域に対するメモリ読み書きで実現でき，GPU の持つ高いピークメモリ帯域幅を生かせる．

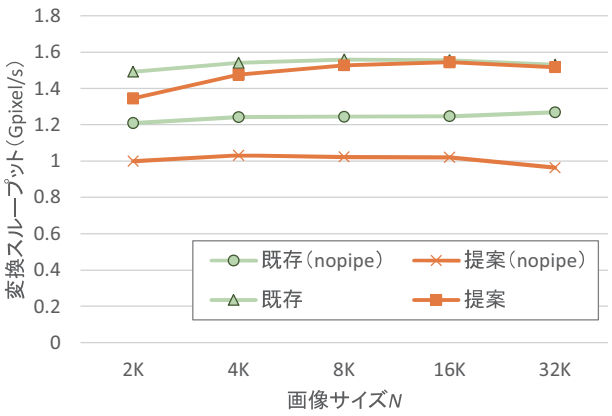


図 7 画像サイズ N ごとの変換スループットの比較

実験の結果，データサイズがビデオメモリ容量の半分を超える場合において，提案手法は Laan らの既存手法と比べて最大で 2.4 倍高速であった．特に，入出力領域の双方，あるいはいずれかが GPU 側にある場合に提案手法は有用である．また，提案手法は 70%程度の高いメモリ参照効率を達成している．しかし，既存手法よりも 3 倍ほどメモリ参照量が多く，カーネル実行時間は増大する．この欠点は，パイプライン処理により隠蔽でき，既存手法とほぼ同等の変換スループットを達成できる．

今後の課題としては，GPU においてリフティング方式の in-place 処理を実現することである．そのためには，一連の代表元を in-place で計算する必要がある．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」および科研費 25136711 の補助による．

参考文献

[1] Ao, J., Mitra, S. and Nutter, B.: Fast and Efficient Lossless Image Compression Based on CUDA Parallel Wavelet Tree Encoding, *Proc. Southwest Symp. Image Analysis and Interpretation (SSIAI'14)*, pp. 21–24 (2014).

[2] Song, C., Li, Y. and Huang, B.: A GPU-Accelerated Wavelet Decompression System With SPIHT and Reed-Solomon Decoding for Satelite Images, *IEEE J. Selected Topics in Applied Earth Observations and Remote Sensing*, Vol. 4, No. 3, pp. 683–690 (2011).

[3] Yıldırım, A. A. and Özdoğan, C.: Parallel WaveCluster: A linear scaling parallel clustering algorithm implementation with application to very large datasets, *J. Parallel and Distributed Computing*, Vol. 71, No. 7, pp. 955–962 (2011).

[4] Sweldens, W.: The Lifting Scheme: A Construction of Second Generation Wavelets, *SIAM J. Mathematical Analysis*, Vol. 29, No. 2, pp. 511–546 (1998).

[5] Mallat, S. G.: A Theory for Multiresolution Signal Decomposition: The Wavelet Representation, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 11,

No. 7, pp. 674–693 (1989).

[6] Angelopoulou, M. E., Cheung, P. Y. K., Masselos, K. and Andreopoulos, Y.: Implementation and Comparison of the 5/3 Lifting 2D Discrete Wavelet Transform Computation Schedules on FPGAs, *J. Signal Processing Systems*, Vol. 51, No. 1, pp. 3–21 (2008).

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

[8] van der Laan, W. J., Jalba, A. C. and Roerdink, J. B.: Accelerating Wavelet Lifting on Graphics Hardware Using CUDA, *IEEE Trans. Parallel and Distributed Systems*, Vol. 22, No. 1, pp. 132–146 (2011).

[9] NVIDIA Corporation: CUDA C Programming Guide Version 6.5 (2014). `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[10] Wong, T.-T., Leung, C.-S., Heng, P.-A. and Wang, J.: Discrete Wavelet Transform on Consumer-Level Graphics Hardware, *IEEE Trans. Multimedia*, Vol. 9, No. 3, pp. 668–673 (2007).

[11] Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).

[12] Tenllado, C., Setoain, J., Prieto, M., Piñuel, L. and Tirado, F.: Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting, *IEEE Trans. Parallel and Distributed Systems*, Vol. 19, No. 3, pp. 299–310 (2008).

[13] Franco, J., Bernabé, G., Fernández, J. and Ujaldón, M.: The 2D wavelet transform on emerging architectures: GPUs and multicores, *J. Real-Time Image Processing*, Vol. 7, No. 3, pp. 145–152 (2012).

[14] Kucis, M., Barina, D., Kula, M. and Zemcik, P.: 2-D Discrete Wavelet Transform Using GPU, *Proc. 26th Int'l Symp. Computer Architecture and High Performance Computing Workshop (SBAC-PADW'14)*, pp. 1–6 (2014).

[15] Sharma, B. and Vydyanathan, N.: Parallel Discrete Wavelet Transform using the Open Computing Language: a performance and portability study, *Proc. 24th IEEE Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW'10)* (2010).

[16] Khronos OpenCL Working Group: The OpenCL Specification Version 1.1 (2011). `http://www.khronos.org/registry/cl/`.

[17] Deslauriers, G. and Dubuc, S.: Symmetric Iterative Interpolation Processes, *Constructive Approximation*, Vol. 5, No. 1, pp. 49–68 (1989).