

D-01

CUDA における離散ウェーブレット変換の in-place 計算のためのデータ並べ替え手法

A data arrangement method for in-place computation of discrete wavelet transform on CUDA

生澤 拓也[†] 伊野 文彦[†] 萩原 兼一[†]
Takuya Ikuzawa Fumihiko Ino Kenichi Hagihara

1. はじめに

離散ウェーブレット変換 (DWT: Discrete Wavelet Transform) 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の 1 つであり、入力を低域成分および高域成分に分離して出力する。DWT の主なアルゴリズムとしてリフティング方式がある。リフティング方式の利点は in-place 計算が可能であり、計算量が少ない点である。ここで、本稿における in-place 計算とは、計算のために必要なメモリ領域のサイズが入力用のメモリ領域を除いて定数で済む計算を指す。

ノイズ除去やデータ圧縮などの応用では、DWT の実時間処理が求められている。DWT はメモリ集中型の計算であるため、高いメモリ帯域幅を持つ GPU (Graphics Processing Unit)¹⁾ を用いる高速化手法が提案されている。Laan^ら²⁾ 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)³⁾ 上に実装している。この手法は、GPU におけるメモリ参照の実行効率を高めるために、低域成分および高域成分それぞれがメモリ上の連続領域に格納されるよう並べ替える。これにより各成分の参照において GPU における実効メモリ帯域幅を最大化できる。ただし、この並べ替えは未処理の入力を出力で上書きすることを避けるために、入力および出力のメモリ領域を区別する必要がある。したがって、GPU 上で一度に処理できる入力のサイズはビデオメモリ容量の高々半分程度に制限されてしまい、in-place 計算の利点を損なう。仮に、入力を分割して各々を順に GPU 上で処理したとしても、CPU・GPU 間のデータ転送が全体の性能を低下させる。

そこで本稿では、GPU におけるリフティング方式の in-place 計算を実現するために、リフティング方式のためのデータ並べ替え手法を提案する。提案手法は in-place 計算を実現することにより、CPU・GPU 間のデータ転送量を削減し、サイズの大きな入力に対する高速化を図る。また、GPU の実効メモリ帯域幅を最大化するために、一連のスレッドがメモリ上の連続領域を読み書きできるよう、GPU の得意とする参照パターンのみを用いて要素を並べ替える。具体的には、入力の要素に対する並べ替えを部分的に制限し、メモリ上の連続領域に要素の集合 (以降、チャンクと呼ぶ) を格納する。その後、チャンク単位で要素を並べ替えるとともに、チャンクに対する操

作を交換に限定することにより、in-place 計算を実現する。さらに、交換操作を互換とみなせることに着目し、互換の積を巡回置換に置き換えることにより、メモリ参照量を削減する。

2. リフティング方式のウェーブレット変換

リフティング方式の入力および出力をそれぞれ x_0, x_1, \dots, x_{n-1} および y_0, y_1, \dots, y_{n-1} とする。ここで、 n は入力および出力を構成する要素の数を表す。このとき、出力の低域成分 y_{2t} および高域成分 y_{2t+1} ($0 \leq t < \lceil n/2 \rceil$) は式 (1) および (2) で与えられる。

$$y_{2t} = x_{2t} + \sum_{i=-k_0}^{k_0-1} u_i y_{2(t+i)+1} \quad (1)$$

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

ここで、定数 u_i ($-k_0 \leq i < k_0$)、 p_j ($-k_1 < j \leq k_1$)、 k_0 および k_1 はウェーブレットにより定まる。

リフティング方式は、低域成分 y_{2t} および高域成分 y_{2t+1} をメモリ上に交互に配置する。一方、CUDA 互換の GPU は 128 バイト単位でメモリを参照する。したがって、この配置に対して、一連のスレッドが低域成分を同時に参照する場合、ストライド幅が 1 であるような参照パターンが原因でピークメモリ帯域幅の半分程度しか性能を生かせない。そこで、メモリ参照に関して実行効率を高めるために、入力あるいは出力を並べ替える必要がある。

3. 並べ替え手法

提案手法は、入力を in-place で並べ替え、偶数番目の要素および奇数番目の要素それぞれをメモリ上の連続領域に格納する。メモリ参照に関して実行効率を高めるために、この並べ替えは以下に挙げる 2 つのフェーズに分けて処理される。

- (1) チャンクの生成: 入力 x_0, x_1, \dots, x_{n-1} を m 個のグループに分割する (m は自然数)。グループ内で要素を並べ替え、偶数番目の要素の集合および奇数番目の要素の集合を生成する (図 1)。以降では、前者を偶数チャンクと呼び、後者を奇数チャンクと呼ぶ。なお、チャンクの大きさが 128 バイトの倍数になるようにチャンクを形成する。図 1 に示すように、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。
- (2) チャンクの並べ替え: すべての偶数チャンクのあとに奇数チャンクが並ぶように、チャンクを並べ替える。この際、チャンク (128 バイト) 単位で読み書きすることに

[†] 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

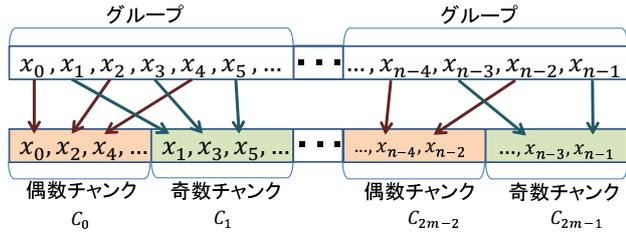


図1 フェーズ1におけるチャンクの生成

Algorithm 1 フェーズ1の擬似コード

Input および Output: n 個の要素からなる入力 $x_0 \sim x_{n-1}$

```

1: for  $u \leftarrow 0$  to  $m - 1$  do           ▷ 入力をグループへ分割
2:    $l \leftarrow \lceil n/m \rceil$ 
3:   for  $v \leftarrow 0$  to  $l - 1$  do
4:      $w_v \leftarrow x_{u \cdot l + v}$ 
5:   end for                               ▷ チャンクの生成
6:   for  $v \leftarrow 0$  to  $l - 1$  do
7:     if  $v$  が偶数 then
8:        $x_{u \cdot l + v/2} \leftarrow w_v$    ▷ 偶数チャンク  $C_{2v}$ 
9:     else
10:       $x_{u \cdot l + \lceil l/2 \rceil + \lfloor v/2 \rfloor} \leftarrow w_v$  ▷ 奇数チャンク  $C_{2v+1}$ 
11:    end if
12:  end for
13: end for

```

より、メモリ参照の実行効率を最大化できる。

Algorithm 1 にフェーズ1の擬似コードを示す。フェーズ1は、要素を $l (= \lceil n/m \rceil)$ 個ずつ処理する。まず、 l 個の要素を作業用変数 $w_0 \sim w_{l-1}$ にコピーする (3~5 行目)。次に、偶数番目の要素および奇数番目の要素がそれぞれ連続領域に格納されるように、 $w_0 \sim w_{l-1}$ を並べ替えて入力のメモリ領域 x に書き戻す (7~10 行目)。

以降では、フェーズ2をチャンクの交換操作のみで実現する手法1、および手法1のメモリ参照量を削減する手法2を示す。

3.1 手法1: 交換による並べ替え

手法1は、フェーズ2において in-place 計算を実現するために、チャンクの交換操作のみで要素を並べ替える。操作をチャンクに対する交換に限定することにより、必要となる作業領域はチャンク1個分で済む。

Algorithm 2 に、手法1の擬似コードを示す。手法1は、フェーズ2を $\log m$ ステップで処理できる。各実行ステップでは、隣接する偶数チャンクおよび奇数チャンクをそれぞれ1つの集合とみなし、この集合を交換する。 s ($0 \leq s \leq \log m$) 番目の実行ステップにおける偶数チャンクの集合および奇数チャンクの集合のうち、先頭から t 番目のものをそれぞれ $E_{s,t}$ および $O_{s,t}$ とする (図2)。このとき、フェーズ2の初期状態において $s = 0$ 、 $L_{s,t} = \{C_{2t}\}$ 、 $H_{s,t} = \{C_{2t+1}\}$ である。各実行ステップにおける処理の内容を以下に示す。

Algorithm 2 手法1の擬似コード

Input および Output: チャンク $C_0 \sim C_{2m-1}$

```

1: for  $s \leftarrow 0$  to  $\log m$  do           ▷ 各実行ステップの処理
2:   for  $t \leftarrow 0$  to  $m/2^{s+1} - 1$  do
3:     for  $r \leftarrow 0$  to  $2^s - 1$  do
4:        $W \leftarrow C_{2^s(4t+1)+r}$ 
5:        $C_{2^s(4t+1)+r} \leftarrow C_{2^s(4t+2)+r}$ 
6:        $C_{2^s(4t+2)+r} \leftarrow W$ 
7:     end for
8:   end for
9: end for

```

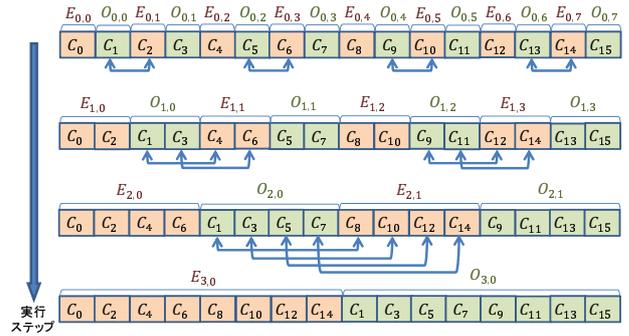


図2 フェーズ2の処理例 ($m = 8$ の場合)

- (1) チャンクの集合の交換: $O_{s,2t}$ と $E_{s,2t+1}$ を交換する ($0 \leq t \leq m/2^{s+1} - 1$)。交換後、 $E_{s,2t}$ は $E_{s,2t+1}$ と隣接し、 $O_{s,2t}$ は $O_{s,2t+1}$ と隣接する (図2)。
- (2) 集合の更新: 新たに隣接した2つの偶数チャンクの集合を統合する。すなわち、 $E_{s+1,t} = E_{s,2t} \cup E_{s,2t+1}$ ($0 \leq t \leq m/2^{s+1} - 1$) となる。奇数チャンクについても同様である。 $O_{s+1,t} = O_{s,2t} \cup O_{s,2t+1}$ ($0 \leq t \leq m/2^{s+1} - 1$)。
- (3) $s = s + 1$ 。

上記の処理 (1) ~ (3) を $E_{s,t}$ および $O_{s,t}$ がそれぞれ1個になるまで、すなわち $m/2^s = 1$ が成立するまで繰り返す。

手法1では実行ステップを進めるたびに、チャンクの集合が大きくなり、集合の数は減少する。処理の負荷を分散するために、 $O_{s,2t}$ と $L_{s,2t+1}$ の交換を 2^s 個の交換に分担する (Algorithm 2 の 4~6 行目)。すなわち、各交換はチャンクどうしを交換する。これにより、常に $m/2$ 個のチャンクを並列に交換できる。

3.2 手法2: 巡回置換による並べ替え

手法2は、任意の巡回置換が互換の積で表現できることに着目し、実行ステップ数を半減する。チャンクの交換および巡回シフトをそれぞれ互換および巡回置換に対応させることにより、2つの実行ステップにまたがる交換を、1つの実行ステップで処理できる巡回シフトに変換する。

チャンク C_f および C_g の交換を互換 (fg) で表すと、以下が成立する。

$$(f h)(f g) = (f g h) \quad (3)$$

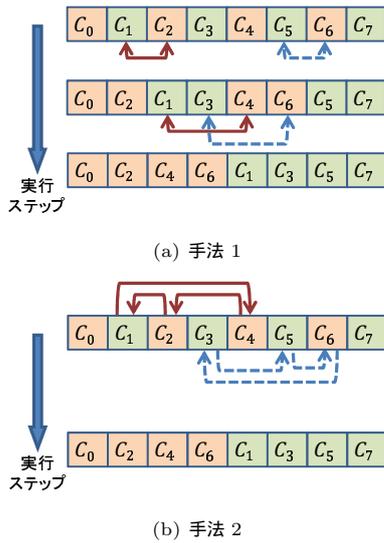


図3 フェーズ2の動作例 ($m = 4$ の場合)

ここで, $(f\ g\ h)$ はチャンク C_f, C_g および C_h の巡回シフトを巡回置換として表したものである. 式 (3) における左辺に相当する一連の交換は, q 番目および $q+1$ 番目の実行ステップ q ($0 \leq q \leq \log m$) に出現する (図3). 例えば, 図3(a) における一連の交換は以下で表せる.

$$(2\ 4)(3\ 5)(1\ 2)(5\ 6) \quad (4)$$

式 (4) は式 (5) および式 (6) のようにまとめることができる.

$$(2\ 4)(1\ 2) = (1\ 4\ 2) \quad (5)$$

$$(3\ 5)(5\ 6) = (5\ 6\ 3) \quad (6)$$

式 (5) および式 (6) はそれぞれ図3(b) の実線および点線の矢印に対応する.

Algorithm 3 に, 手法2の擬似コードを示す. 手法1と同様, 手法2が必要とする作業領域はチャンク1個分の W のみであり, in-place 計算を実現できる. また, 2つの実行ステップにまたがるチャンクの交換 (Algorithm2の4~6行目) が12回のメモリ参照を必要とするのに対し, チャンクの巡回シフト (5~8および12~15行目) は8回のメモリ参照で実現できる. したがって, 手法1と比べて手法2はメモリ参照量を $2/3$ に削減できる.

4. 評価実験

提案手法を評価するために, 提案手法1, 2および既存手法²⁾の各々が処理できる入力サイズ, およびそのときの実行時間を計測した. 表1に, 実験に用いた計算機の仕様を示す. 実験では, 3種類のGPUを使い分けた. GF110およびGK110はPCIe 2.0に接続しているのに対し, GK104はPCIe 3.0に接続している. OSはWindows 7であり, コンパイラとしてVisual Studio 2010を用いた. なお, 1画素あたり4バイトの値を持つ2次元画像を入力として与えた.

図4に, 各環境における実行時間を画像サイズごとに示す. なお, CPU・GPU間のデータ転送時間は除いている. 環境GF110およびGK104において, 既存手法は16K×16K画素の画像に対してメモリ不足が原因で実行に失敗した. 一方, 提

Algorithm 3 手法2の擬似コード

Input および Output: チャンク $C_0 \sim C_{2m-1}$

```

1: for  $s \leftarrow 0$  to  $(\log m)/2$  do   ▷ 各実行ステップの処理
2:   for  $t \leftarrow 0$  to  $m/2^{2s+1} - 1$  do
3:     if  $t$  が偶数 then
4:       for  $r \leftarrow 0$  to  $2^{2s} - 1$  do
5:          $W \leftarrow C_{2^{2s}(4t+1)+r}$ 
6:          $C_{2^{2s}(4t+1)+r} \leftarrow C_{2^{2s}(4t+2)+r}$ 
7:          $C_{2^{2s}(4t+2)+r} \leftarrow C_{2^{2s+1}(2t+2)+r}$ 
8:          $C_{2^{2s+1}(2t+2)+r} \leftarrow W$ 
9:       end for
10:    else
11:      for  $r \leftarrow 0$  to  $2^{2s} - 1$  do
12:         $W \leftarrow C_{2^{2s}(4t+1)+r}$ 
13:         $C_{2^{2s}(4t+1)+r} \leftarrow C_{2^{2s+1}(2t+1)+2^{2s}+r}$ 
14:         $C_{2^{2s+1}(2t+1)+2^{2s}+r} \leftarrow C_{2^{2s}(4t+2)+r}$ 
15:         $C_{2^{2s}(4t+2)+r} \leftarrow W$ 
16:      end for
17:    end if
18:  end for
19: end for

```

案手法1および2はin-place計算によりその画像サイズを処理できている. ただし, これらの実行時間は, 既存手法の実行時間に対して2.6~3.6倍および2.4~3.2倍であり, 提案手法は既存手法よりも性能が低い. 性能低下の原因は並べ替えのオーバーヘッドにあり, ビデオメモリ参照量がそれぞれ最大6.3倍および5.4倍に増加したためである.

次に, 既存手法が大きな画像を処理できるように, 画像を分割して各々を順に処理した. 図5に, 分割実行版の既存手法を用いたときの実行時間を示す. 画像サイズは環境GF110, GK104およびGK110においてそれぞれ16K×16K, 16K×16Kおよび32K×32K画素である. PCIe 2.0に接続する環境GF110およびGK110において, 提案手法は既存手法よりも高速であり, その速度向上率はそれぞれ1.06~1.08倍および1.11~1.12倍である. 一方, PCIe 3.0に接続する環境GK104では, 提案手法は既存手法よりも6~12%ほど低速である. 提案手法はCPU・GPU間のデータ転送量を削減できるが, 並べ替えに起因するオーバーヘッドがGPU上の実行時間を増大させてしまう. したがって, CPU・GPU間データ転送が高速なPCIe 3.0においては, オーバヘッドの増大がデータ転送量の削減効果を上回り, 既存手法よりも低速となった.

フェーズ2の実行時間に関して提案手法2および1を比較すると, すべての環境において, 前者は後者のおよそ3/4の実行時間で処理を終えている. この高速化は, 巡回置換によるビデオメモリ参照量の削減による. 手法1に対して手法2はメモリ参照量を $2/3$ に削減しているが, 実行時間の短縮は $3/4$ に留まっている. この理由は, 両手法とも作業領域 W にビデオメモリより100倍以上高速にアクセスできるレジスタ

表 1 実験環境

環境	GF110	GK104	GK110
GPU	NVIDIA GeForce GTX 580	NVIDIA GeForce GTX 680	NVIDIA Tesla K20
メモリ容量 (MB)	1536	2048	5376
バス	PCIe 2.0	PCIe 3.0	PCIe 2.0
CPU	Intel Core i7-3770K		
CUDA	5.0		
ドライバ	320.49		

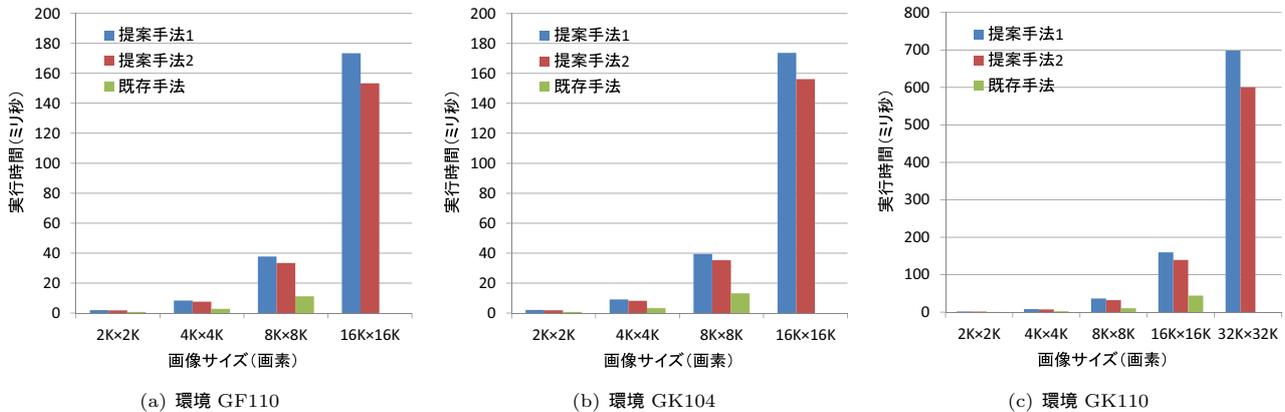


図 4 各環境における実行時間

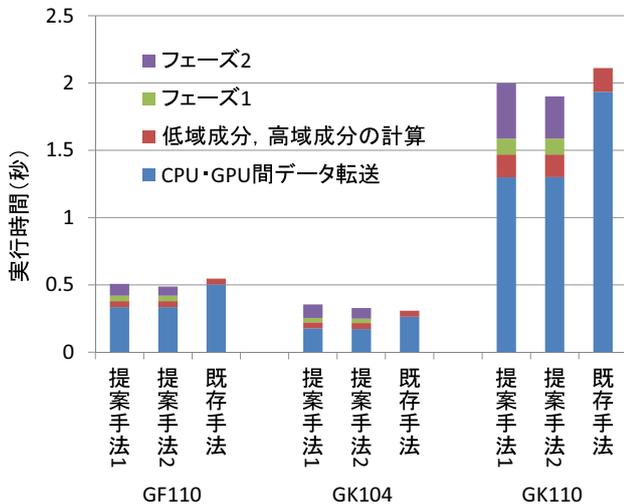


図 5 分割実行版の既存手法との比較

を使用しており、2/3 の削減にはレジスタへの参照が含まれているためである。ビデオメモリのみに関する参照量では 3/4 に削減されている。

5. まとめ

本稿では、CUDA におけるリフティング方式の in-place 計算を実現することを目的として、リフティング方式のためのデータ並べ替え手法を提案した。提案手法は in-place 計算によりサイズの大きな入力に対する高速化を図る。in-place 計算を実現するために、提案手法は一度に要素を並べ替えるのではなく、並べ替えの対象を部分的に制限することにより GPU

の得意とする参照パターンのみを用いて要素を並び替える。また、互換の積を巡回置換に置き換えることができることに着目し、ビデオメモリ参照量を 3/4 に削減する。

実験の結果、PCIe 2.0 に接続する GPU において提案手法は既存手法よりも最大で 1.12 倍高速であった。一方、PCIe 3.0 に接続する GPU では、提案手法は 6~12% 低速であった。

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」、科研費 23300007、および 23700057 の補助による。

参考文献

- 1) 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.
- 2) W.J. van der Laan, A.C. Jalba, and J.B.T.M. Roerdink. Accelerating Wavelet Lifting on Graphics Hardware Using CUDA. *IEEE Trans. Parallel and Distributed Systems*, Vol.22, No.1, pp. 132-146, January 2011.
- 3) NVIDIA Corporation. CUDA C Programming Guide Version 5.0, October 2012. <http://docs.nvidia.com/cuda/>.