CPU-GPU アーキテクチャを用いた Content-Defined Chunking の実装と評価

松宮 遼[†] 平井成海[†] 佐々木 慎[†]
高橋一志[†,††] 大山恵弘[†,††]

Content-Defined Chunking（CDC）は，ファイルを中身に基づいて可変長のブロック（チャンク）に分割する方式であり，ファイルシステムやストレージに対するキャッシュ機構や重複除外システム等によって利用されている．データインテンシブサイエンスにおける大規模データ処理のための計算機は，ギガバイトスケールのファイルを扱うことがあり，これらのファイルを扱う上で，CDC のオーバーヘッドを減らすことが課題となっている．既存研究では現在の計算機が多くの CPU 及び GPU 資源を有していることに着目した手法が提案されてきたが，それらの計算資源を有効に利用できているとはいえない．本稿では，計算資源を有効に利用することで，CDC のオーバーヘッドを低減する手法を 2 つ提案する．また，CPU のみで並列化を行った場合や，GPU のみで並列化を行った場合と提案手法の比較実験を行うことで，提案手法の評価を行ったのでそれを報告する．

Implemention and Evaluation of Content-Defined Chunking using CPU-GPU Architecture

Ryo Matsumiya,[†] Narumi Hirai,[†] Shin Sasaki,[†]
Kazushi Takahashi[†,††] and Yoshihiro Oyama[†,††]

Content-Defined Chunking (CDC) is a method of dividing a file into variable-size blocks (chunks) based on its contents, and used in caching and deduplication mechanisms for file systems and storage. In data-intensive science, computers for large-scale data processing can analyze gigabyte-scale files, and reducing the overhead of CDC is an important issue. Although previous work has proposed methods to achieve it by focusing the rich CPU and GPU resource available in these computers, the methods cannot use the resource efficiently. In this work, we propose two methods for reducing the overhead of CDC by using CPU and GPU resource efficiently. We evaluated the proposed methods through experiments in which the method was compared with parallelization methods that use CPU only or GPU only.

1. はじめに

Content-Defined Chunking（以下，本稿では CDC とする）とは，ファイルの内容に基づいて，そのファイルを可変長のブロック（チャンク）に分割する方式であり，Content-based Chunking とも呼ばれている．CDC は重複したファイルの一部分をそのファイル上での位置によらず検出できるため，低帯域幅のための分散システムである LBFS[1] をはじめとして，多くのファイルシステム [2]，バックアップシステム [3] や重複除外システム [4] 等に対して広く用いられている．

CDC は有効な手法であるが，特に巨大なデータを扱う時には多くの CPU パワーや時間を消費する．そのため高速化は大きな課題である．

この課題に対するアプローチの 1 つとしてマルチコア CPU[4)5)] 及び Graphics Processing Unit（以下，本稿では GPU とする）[6)7)] による並列化が挙げられる．しかし，その研究の多くは CPU あるいは GPU の一方のみでしか計算していない．また，双方で計算している研究 [8)9)] も，CPU 又は GPU に待ち時間が多く発生してしまっており，CPU と GPU の両方を持つ計算機にとっては計算資源を有効に利用できているとは言えない．これらの計算資源を有効に利用することで，更なる高速化が見込まれる．

そこで本研究では CPU と GPU を有効に利用することで CDC を高速化する手法を提案する．その手法

† 電気通信大学
The University of Electro-Communications
†† 独立行政法人科学技術振興機構，CREST
JST，CREST

ⓒ 2013 Information Processing Society of Japan

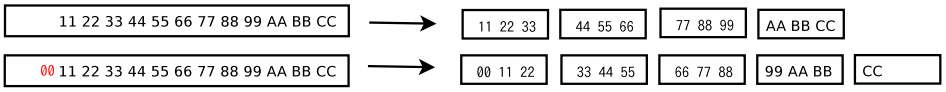

図 1　固定長チャンク分割の例
Fig. 1　Example of FSC

はさらに 2 つの手法に分かれており，1 つは静的な負荷分散によるもの，もう 1 つは動的な負荷分散によるものである．また，CPU のみで並列化を行った場合や，GPU のみで並列化を行った場合及び CPU シングルスレッドの逐次処理との比較実験を行うことで，提案手法の評価を行った．

本稿の構成を以下に示す．2 章で CDC の概要について述べ，3 章で提案手法，4 章で実装，5 章で本研究で行った実験についてそれぞれ述べる．6 章で関連研究について述べ，最後に 7 章でまとめと今後の課題について述べる．

2. CDC の概要

2.1 FSC

Fixed Size Chunking（以下，本稿では FSC とする）とは，ファイルを固定長のチャンクに分割するものであり，Google File System[10] や Venti[11] 等において利用されている．しかし FSC には，もし 1 バイトであっても，ファイルの先頭や途中においてデータの追加や削除が行われると編集箇所以降のチャンクがすべて変化してしまうという問題がある．例として図 1 を用いる．左上の四角形で示されたようなファイルを FSC にて 3 バイトの固定長で分割すると，右上のようになる．左下の四角形は，左上で示されたファイルの先頭に，0x00 という 1 バイトを追加したファイルを表している．左下の四角形で示されたファイルを同様に FSC にて分割した結果のチャンクが右下に示されている．1 バイトの追加により，右上とは全く違うチャンクが生成されてしまう．その結果，左上のファイルと左下のファイルには，ほとんど同じデータが含まれるにもかかわらず，それらのファイルからは，内容が同じであるチャンクが一つも作られない．

2.2 CDC

CDC はファイルの一部が変更されても，他のチャンクへの影響が少ない．CDC はチャンクの境界線を求めるために固定長の sliding window を用いる．これは，ファイルの先頭から 1 バイトずつスライドさせるものである．Sliding window 内におけるファイルデータのフィンガープリントを計算していく．そのフィンガープリントの下位数ビットが特定の値と一致した時に，その時の sliding window の終点をチャンクの境界線とする．

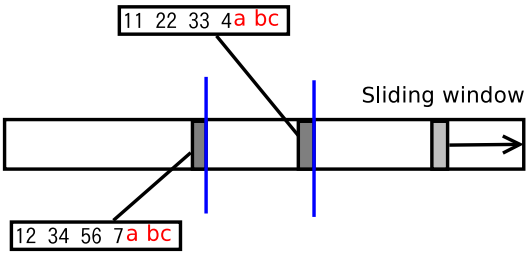

図 2　CDC の例
Fig. 2　Example of CDC

CDC によりファイルをチャンクに分割する例を図 2 に示す．中央の大きく白い四角形はファイルの全体を表し，薄い灰色の四角形は sliding window を表す．なお，フィンガープリントのサイズは 40 ビットである．sliding window が濃い灰色の箇所にあるとき，sliding window 内におけるフィンガープリントの下位 12 ビットが特定値（この例においては 0xabc とする）であったとき，青の線で示しているように，その sliding window の終端がチャンクの境界線となる．したがって，CDC によるファイルの分割においては，ファイルに 1 バイトの変更があったとして，影響を受けるチャンクはウィンドウがその 1 バイトを含む範囲に限られる．

このように CDC は重複除外の精度を向上させることができる．しかし最低でも 1 度はファイルを 1 バイトずつ読んでいく必要があるため，FSC に比べ処理速度が落ちてしまうという欠点がある．

2.3 Rabin fingerprint

CDC においてフィンガープリントを求めるアルゴリズムとして，LBFS では Rabin fingerprint[12] が用いられている．Rabin fingerprint とは，入力である α ビットのビット列を $m_1 m_2 \cdots m_\alpha$ としたときに

$$f(m_1, m_2, \ldots, m_\alpha) = \sum_{n=1}^{\alpha} m_n x^{\alpha - n}$$

とし，この $f(m_1, m_2, \ldots, m_\alpha)$ を素数 k における $k-1$ 次既約多項式 $p_k(x)$ で割った余りの多項式を求め，その多項式から得られるビット列である．また，この $f(m_2, m_3, \ldots, m_\alpha, m_{\alpha+1})$ は，

ⓒ 2013 Information Processing Society of Japan

$$
\begin{aligned}
& f(m_2, m_3, \ldots, m_\alpha, m_{\alpha+1}) \\
&= \sum_{n=2}^{\alpha+1} m_n x^{\alpha+1-n} \\
&= x \left[\left(\sum_{n=1}^{\alpha} m_n x^{\alpha-n} \right) - m_1 x^{\alpha-1} \right] \\
&+ m_{\alpha+1} \\
&= x \left[f(m_1, m_2, \ldots, m_\alpha) - m_1 x^{\alpha-1} \right] \\
&+ m_{\alpha+1}
\end{aligned}
$$

のように変形することが出来る．これは，ウィンドウを 1 バイトずつずらしていった時に，前に計算したウィンドウについての計算結果を利用することによる高速化が行えることを示している [13]．本研究ではこの高速化手法を利用している．

3. 提案手法

本章ではマルチコア CPU と GPU の双方を利用した並列化手法を提案する．その並列化手法は，(1) 静的な負荷分散を用いた手法と，(2) 動的な負荷分散を用いた手法，の 2 つからなる．

3.1 静的な負荷分散を用いた手法

静的な負荷分散を用いた手法は，ファイルのデータを固定長セクションに区切る．セクションの長さは，用いる CPU や GPU のコア数でファイルサイズを割った値にほぼ等しくなる．区切られた各セクションは各 CPU コア（又は GPU）に割り当てられる．各 CPU コア（又は GPU）は，自分に割り当てられたセクションのデータを CDC によりチャンクに分割する．

なお，ファイルのデータを単純に区切って各セクションの処理を独立に行うと，セクションの開始位置付近に存在するチャンクの境界線を検出できないという問題が生じる．そこで，本手法では，$w-1$ バイトずつ，前のセクションと被せるようにセクションに区切る．ここで，w は sliding window のサイズである．この区切り方により，すべてのチャンクの境界線を検出できるようになる．

3.2 動的な負荷分散を用いた手法

動的な負荷分散を用いた手法も，ファイルのデータを固定長セクションに区切る．ただし，この段階ではそのセクションに対するチャンク分割を GPU が行うのか CPU が行うのかが確定できない．したがって静的な負荷分散を用いた手法に比べて，セクションの長さをより短くし，セクションの個数をより多くする．この手法では，各セクションのデータをチャンクに分割する処理をタスクというデータ構造で表現する．その処理を実行するためのスレッド（ワーカスレッド）を，CPU の各コアと GPU に，0 個以上生成する．タスクは，ワーカスレッドごとに作られる両端キューのようなデータ構造（タスクキュー）によって管理する．スレッド間の負荷分散は，ワークスチールによって実現する．ワークスチールは，自分のタスクキューが空になったワーカスレッドが，他のワーカスレッドのタスクキューからタスクを奪うことにより負荷分散を行う方式である．本手法のタスクキューは，図 3 に示すように，ワーカスレッドが自分のタスクキューにアクセスする場合は LIFO として動作し，他のスレッドからタスクを奪う場合には FIFO として動作する．

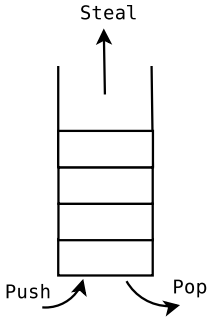

図 3 タスクキューの構造
Fig. 3 Structure of a task queue

本手法では，ワーカスレッドに加えて，ワーカスレッドが行う仕事を管理するスレッド（マスタースレッド）を 1 個生成する．マスタースレッドは，まず，ファイルのデータを固定長セクションに区切り，各セクションに対して行う処理を表現する構造体を作成する．この構造体がタスクとして用いられる．タスクの中には，セクションのサイズ，セクションのファイル内オフセット，セクション内のデータ等が保存される．次に，マスタースレッドは CPU 用のタスクキュー同士で均等，GPU 用のタスクキューでも均等になるようにタスクを各ワーカスレッドのタスクキューに push する．

各ワーカスレッドは自分のタスクキューからタスクを pop し，そのタスクが表現するファイル部分に対してチャンク分割を行う．ワーカスレッドは，自分のタスクキューが空になり，実行できるタスクがない場合には，ワークスチールを実行する．CPU 上で動作しているワーカスレッドは，CPU 上で動作している他のワーカスレッドのタスクキューから優先的にタスクを奪う．一方，GPU 上で動作しているワーカスレッドは，GPU 上で動作している他のワーカスレッドのタスクキューから優先的にタスクを奪う．優先される複数のタスクキューのうちどのタスクキューを選ぶかは，ランダムに決定する．CPU と GPU の間でタスク

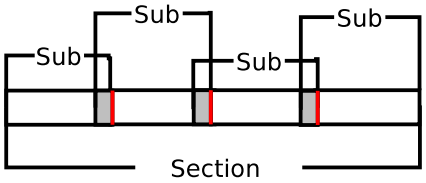

図 4　サブセクションへの分割
Fig. 4　Division into subsections

を極力移動させない理由は，CPU 側のメモリと GPU 側のメモリの間のデータ転送に長い時間がかかるためである．CPU 上で動作するワーカスレッドのためのタスクキューは CPU 側のメモリ上に作られ，GPU 上で動作するワーカスレッドのためのタスクキューも CPU 側のメモリ上に作られる．

本手法でも単純にセクションに区切ると，静的な負荷分散を用いた手法と同様に，セクションの開始位置付近に存在するチャンクの境界線を検出できないという問題が生じる．この問題を解決するために，セクション間の境界線を中心とした $2(w-1)$ バイトからなる部分もセクションとし，それらのセクションの処理を表現するタスクもタスクキューに入れるようにした．w は sliding window のサイズである．また，固定長セクションに区切る際，ファイルの末端には固定長に満たない部分が残ることがあるが，その末端部に対しても 1 つのセクションとして同様にタスクを作成する．

マスタースレッドは固定長セクションの処理を行うタスクを GPU のタスクキューに入れる．ファイルの末端部とセクションの境界線部分の処理を行うタスクを CPU のタスクキューに入れる．

4. 実　　装

提案手法の実装を行った．また，提案手法において GPU を用いてる部分も CPU で行うもの，提案手法において CPU を用いてる部分も GPU で行うものについても比較のために実装した．

提案手法の実装を行った．実装には CUDA[14]，C++11 の Thread support library 及び OpenMP（GCC OpenMP support library）を利用した．

各ワーカスレッドはチャンクの境界線を検出するたびに，境界線の情報を保存しておくためのメモリ領域にその情報を書き込む．そのメモリ領域は各ワーカスレッドが確保する．

GPU 上で動作するワーカスレッドは，まず，チャンクに分割するデータを保存するメモリ領域及びチャンク境界線の情報を保存するメモリ領域を，関数 `cudaMalloc` を用いて GPU の Global Memory に確保する．後者のメモリ領域については，後述する関数 `cudaHostAlloc` を用いて CPU 側のメモリにも確保する．GPU 上で動作するワーカスレッドはタスクを獲得すると，そのタスクに対応するセクションのデータを GPU 側のメモリに転送し，さらに，チャンク境界線の情報を保存するための領域を初期化する．その後，ワーカスレッドは GPU を用いて転送されたデータに対してチャンク分割を行い，検出された境界線の情報を，所定のメモリ領域に書き込む．GPU によるチャンク分割の処理が終了したら，ワーカスレッドは境界線の情報を GPU 側のメモリから CPU 側のメモリに転送する．

続いて，GPU で行われる処理について説明する．GPU はセクションの情報を受け取ると，セクションを更に細かい単位（サブセクション）に分割する．1 つのサブセクション毎に 1 つの GPU スレッドが割り当てられる．サブセクションへの分割の方法を，図 4 に示す．まず，赤線で示すようにセクションを固定長のサブセクションに分割する．しかしこの分割方法では，サブセクションの境界線付近にあるチャンクの境界線を検出できない．そこで，静的な負荷分散を用いた手法と同様に，各サブセクションの先頭と，その 1 つ前のサブセクションの末尾を，$w-1$ バイトだけ重複させる．ただし，w は sliding window のサイズである．図では重複部分を灰色の四角で示している．

基本的に，CPU-GPU 間ではお互いのメモリ上のデータを直接参照することは出来ない．したがって，CPU 側のメモリ上にあるデータを GPU に渡す，あるいは GPU 側の Global Memory 上にあるデータを CPU に渡す時には，関数 `cudaMemcpy` などを用いて CPU と GPU の間で明示的にデータを転送しなければならない．境界線の情報を保存するためのメモリ領域は，関数 `cudaHostAlloc` を用いて page-locked なメモリ領域（pinned memory ともいう）に確保している． このメモリ領域に対しては OS によるスワップが行われなくなるため，データ転送速度が向上することがある．また，本研究の実装では，GPU 上で動作するワーカスレッドがタスクを獲得した際に，チャンク分割を行うデータが載っているメモリ領域を，関数 `cudaHostRegister` によって page-locked なメモリ領域に変えて，CPU-GPU 間のデータ転送を高速化している．

以上で述べたものが，提案手法を実装したプログラムである．本研究では，比較のために，それに加え

ⓒ 2013 Information Processing Society of Japan

表 1　実験環境
Table 1　Environment of experiments

CPU	Intel Core i7 4cores 8threads 3.50GHz
OS	Gentoo Linux
Linux kernel	3.10.7
Memory	32GB
HDD	SATA3 3TB 7200rpm cache 64MB
GPU	NVIDIA GeForce GTX TITAN
GPU driver	331.13
CUDA	5.5
GCC	4.8.1

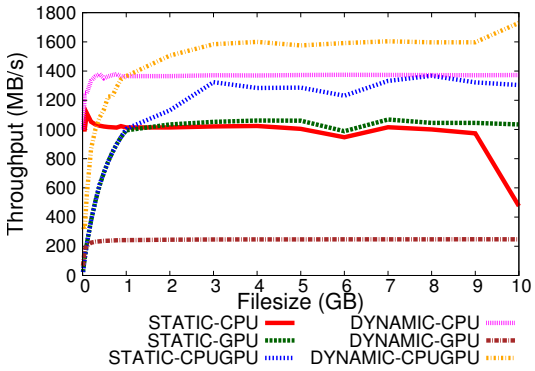

図 5　プログラムのスループット
Fig. 5　Throughput of the program

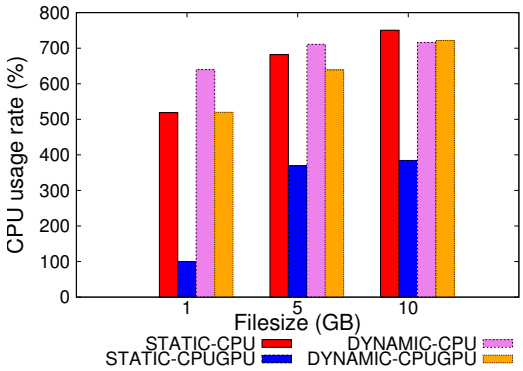

図 6　プログラムの CPU 使用率
Fig. 6　CPU usage rate of the program

て，提案手法において GPU が処理していたタスクも CPU に処理させるプログラム（CPU のみでチャンク分割を行うプログラム）と，提案手法において CPU が処理していたタスクも GPU に処理させるプログラム（GPU のみでチャンク分割を行うプログラム）も実装した．

5. 実　　験

提案手法の効果を検証するために，ランダムな内容を持つ様々なサイズのファイルを作成し以下の実験を

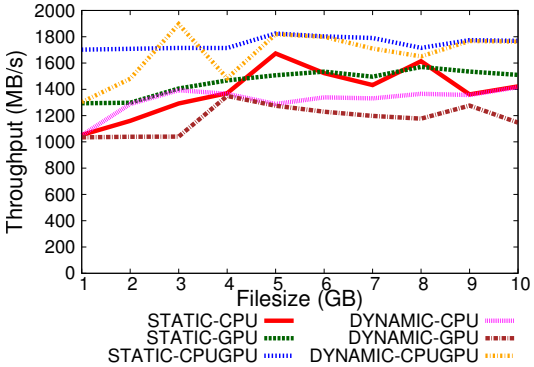

図 7　フィンガープリント処理のみのスループット
Fig. 7　Throughput of fingerprinting only

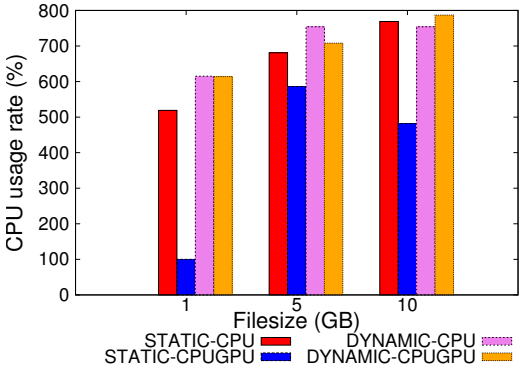

図 8　フィンガープリント処理のみの CPU 使用率
Fig. 8　CPU usage rate of fingerprinting only

行った．実験環境を表 1 に示す．

5.1　プログラムのスループットと CPU 使用率

実装したプログラムを用いて CDC によるファイル分割を行い，プログラムのスループットと CPU 使用率を測定した．スループットを求めるための時間測定には Linux の time コマンドを利用した．CPU 使用率の測定には `clock_gettime` のクロックタイプを `CLOCK_PROCESS_CPUTIME_ID` にしたものによってプロセスの CPU 時間を計測し，それを処理時間で割ったものを利用した．

スループットの測定結果を図 5 に示す．静的な負荷分散において，マルチコア CPU のみの並列化（STATIC-CPU）では最大 1111MB/s，GPU のみの並列化（STATIC-GPU）では最大 1068MB/s，双方による並列化（STATIC-CPUGPU）では最大 1367MB/s のスループットが得られた．動的な負荷分散において，マルチコア CPU のみの並列化（DYNAMIC-CPU）では最大 1379MB/s，GPU のみの並列化（DYNAMIC-GPU）では最大 247MB/s，双方による並列化（DYNAMIC-CPUGPU）では最大

ⓒ 2013 Information Processing Society of Japan

1729MB/s のスループットが得られた．

CPU 使用率の測定結果を図 6 に示す．静的な負荷分散を行うマルチコア CPU と GPU の双方を利用した並列処理の場合 CPU 使用率が最大でも 384%となった．他の 3 つの場合については特にファイルサイズが 10GB のとき，CPU 使用率が 700%以上となった．また，ファイルサイズが増大するにつれて CPU 使用率も増大している．

5.2 フィンガープリント処理のみのスループットと CPU 使用率

ファイルの内容を事前にメモリ上に移した上で CDC によるファイル分割を行うように，実装したプログラムを書き換えた．そして CDC によるファイル分割を行う部分だけのスループットと CPU 使用率を測定した．スループットの測定には `clock_gettime` でクロックタイプは `CLOCK_MONOTONIC` を用いた．CPU 使用率の測定には `clock_gettime` のクロックタイプを `CLOCK_PROCESS_CPUTIME_ID` にしたものによってプロセスの CPU 時間を計測し，それを処理時間で割ったものを利用した．

スループットの測定結果を図 7 に示す．静的な負荷分散において，マルチコア CPU のみの並列化（STATIC-CPU）では最大 1710MB/s，GPU のみの並列化（STATIC-GPU）では最大 1570MB/s，双方による並列化（STATIC-CPUGPU）では最大 1824MB/s のスループットが得られた．動的な負荷分散において，マルチコア CPU のみの並列化（DYNAMIC-CPU）では最大 1416MB/s，GPU のみの並列化（DYNAMIC-GPU）では最大 1349MB/s，双方による並列化（DYNAMIC-CPUGPU）では最大 1896MB/s のスループットが得られた．

CPU 使用率の測定結果を図 8 に示す．静的な負荷分散を行うマルチコア CPU と GPU の双方を利用した並列処理の場合 CPU 使用率が最大でも 586%となった．他の 3 つの場合については特にファイルサイズが 10GB のとき，CPU 使用率が 750%以上となった．また，ファイルサイズが増大するにつれて CPU 使用率も増大している．

5.3 考　　察

実験の結果，ギガバイトスケールのファイルにおいて，CPU と GPU の双方を利用することはどちらか一方を利用することよりも高速であるということが言える．VM イメージや MPEG2 Transport Stream[15] といった大きなファイルサイズを前提としたストレージやバックアップシステム等に対して，提案手法は有用であると言える．

CPU 使用率は静的な負荷分散で CPU と GPU の双方を利用すると，動的な負荷分散を用いた手法や静的な負荷分散で CPU のみを利用した場合に比べて低い値となった．このことは，静的な負荷分散を行うと CPU を効率的に利用できない場合があることを表している．一方，動的な負荷分散を用いた手法では CPU 使用率が十分に高く，動的な負荷分散を用いた手法は，ワークスチールを使うことで CPU を効率的に利用できていると言える．ファイルサイズが増大するにつれて CPU 使用率が増大しているのは，スレッドを生成するといった，逐次的な処理における処理時間が，並列処理における処理時間に対して微小なものとなっていくためであると考えられる．

2 つの提案手法におけるスループットが，フィンガープリントのみの処理では有意な差が見られなかったのに対し，プログラムのスループットにおいては有意な差が見られる．これはファイルを読み込む際のバッファサイズ等が関連していると考えられる．

最後に，以下に示す 3 つの点について，より詳細な検討が今後必要である．

- 静的な負荷分散を行うマルチコア CPU と GPU の双方を利用した並列処理の場合，ファイルサイズが 5GB のときに比べて 10GB のときの CPU 使用率が下がっている点．
- 動的な負荷分散を用いた GPU のみの並列処理で，フィンガープリント処理のスループットに対し，プログラムのスループットが極端に遅くなっている点．
- 静的な負荷分散を用いたマルチコア CPU のみの並列処理で，ファイルサイズが 10GB のファイルを分割するとき，他のファイルサイズに比べてプログラムのスループットが極端に遅くなっている点．

6. 関 連 研 究

GPU による並列化によって高速化を図る方法を示した研究は多岐にわたる．例えば，CUDA や OpenACC[16]，Brook[17] といった GPU 向け開発環境により，ネットワークに関する処理を高速化した研究[18][19] が存在する．ストレージ及びファイルシステム分野も例外ではない．Gharaibeh らの研究[20] では Content Addressed Storage において，ファイルの内容をハッシュ化する際に GPU を使うことにより高速化を行っている．

マルチコア CPU と GPU の双方を用いた並列化についても現在研究が進められている．OpenCL[21] は

ⓒ 2013 Information Processing Society of Japan

マルチコア CPU と GPU の双方を用いる環境のための開発環境である．Ogata ら [22] はマルチコア CPU と GPU の双方を用いて高速フーリエ変換を行うライブラリを作成した．Benner ら [23] はマルチコア CPU と GPU の双方を用いて行列符号関数の高速化を行った．本研究ではマルチコア CPU と GPU の双方を用いる手法を CDC に適用した．

坪内らの研究 [24] では FSC での利用を想定し，SHA-1 の計算をマルチコア CPU 及び SSE によって行っているが，本研究は CDC に関する研究である．

EndRE[25] では，Rabin fingerprint を使わずに SampleByte というものを使うことで CDC の速度向上を図っている．しかしチャンクのサイズが大きくなる場合はチャンクの境界線を発見しにくいという欠点がある．

村上らの研究 [4] や P-Dedupe[5] では CDC 及びその関連処理における高速化を CPU の並列化によって行っているが，これらの研究では GPU を一切用いていない．Shredder[6] や GHOST[7] では CDC 及びその関連処理における高速化を GPU の並列化によって高速化しているが，CPU を CDC に用いていない．

CPU と GPU の双方を利用した関連研究として Ma らの手法 [8] と Tang らの手法 [9] が挙げられる．Ma らの手法は，パイプラインのような手法を取り入れ，GPU で Rabin fingerprint によってチャンク境界線を求め，その後 CPU で各チャンクの SHA-1 を計算するというものである．図 9 は Ma らの手法を表したものである．上 2 つの四角形は GPU が順に行う処理，下 2 つの四角形は CPU が順に行う処理を表している．GPU はまず File A のチャンク境界線を求め，求まり次第，File B のチャンク境界線を求める．CPU は，GPU で File A のチャンク境界線が求まるまで待ち，求まり次第，File A に関する各チャンクの SHA-1 ハッシュ値を計算する．この手法は，CPU 及び GPU の双方を利用したものであるが，この手法では最初のファイルについてチャンク境界線を求めている間は CPU 資源が余剰となってしまう．したがって，この手法は計算資源を有効に利用できているとは言えない．Tang らの手法は，特定値以上のサイズであるファイルの CDC 及びその関連処理を GPU に，そうでないファイルの CDC 及びその関連処理を CPU に行わせるという方法である．この手法は特定値以上のサイズであるファイルのみについて CDC の処理を行う場合は，全てのファイルの CDC 及びその関連処理を GPU に行わせるため CPU 資源が余剰となってしまう．これはギガバイトスケールのファイルを扱う事

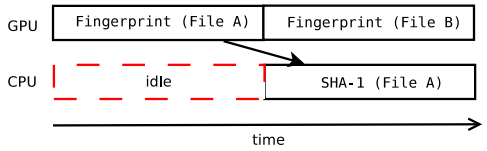

図 9 Ma らの手法
Fig. 9 Ma et al.'s method

を前提としている本研究とは異なるものである．

Blumofe らの手法 [26] ではワークスチールを行うタスクキューをランダムに決定することにより，効率を向上させている．彼らの研究においては無差別にタスクキューを決定しているが，本研究においては CPU は CPU のタスクキューから優先的に，GPU は GPU のタスクキューから優先的にワークスチールを行うことで，CPU あるいは GPU にとって適切な仕事を割り当てるようにしている．

7. まとめと今後の課題

本稿では，CPU 及び GPU の双方の計算資源を有効に利用することで，どちらか一方を用いた場合に比べて CDC の処理を高速化できること，その場合，ワークスチールによって CPU を有効に利用できるようになることを示した．以下に今後の課題について述べる．

理想的には，マルチコア CPU のみによる並列化のスループットと，GPU のみによる並列化のスループットを足した値がマルチコア CPU と GPU の双方を利用した並列化のスループットとなるべきである．しかしながら本稿で行った実験においてはその値に到達できなかった．ファイルの読み込み方法や，ワークスチールの方法を見直す必要がある．

本研究における手法の GPU 処理では Global Memory のみを使用し，そのレイテンシに阻害されてしまっていると考えられる．したがって，高速な GPU メモリである Shared Memory も使用した高速化手法も検討するべきである．また SSE 等を用いたベクトル化による SIMD 並列化手法等を取り入れ更なる高速化が実現できる可能性がある．そしてこれらの高速化手法を実際のファイルシステムやバックアップシステムに取り入れ，その効果を調べていきたい．

本研究では，チャンク分割を行う計算機が全ての CPU と GPU を使えること，すなわち，分散ファイルシステムの処理専用に使える計算機があることを想定している．しかし，実際には，分散ファイルシステムの処理を行う計算機上で同時にアプリケーションを動作させる運用もしばしば行われている．したがって，チャンク分割を行う計算機が．ファイルシステム専用

ⓒ 2013 Information Processing Society of Japan

の計算機でない場合の手法を検討する必要がある．

謝辞　本研究を行うにあたって，有益な助言をして頂いた筑波大学の建部修見准教授及び建部研究室の方々に深く感謝する．また本論文を執筆するにあたり，貴重なコメントをして頂いた査読委員の方々にも深く感謝する．本研究は，科学技術振興機構戦略的創造研究推進事業（JST CREST）の研究課題「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」の支援を受けている．

参　考　文　献

1) Muthitacharoen, A., Chen, B. and Mazières, D.: A low-bandwidth network file system, *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, New York, NY, USA, ACM, pp. 174–187 (2001).
2) Ungureanu, C., Atkin, B., Aranya, A., Gokhale, S., Rago, S., Calkowski, G., Dubnicki, C. and Bohra, A.: HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system, *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, Berkeley, CA, USA, USENIX Association (2010).
3) Kruus, E., Ungureanu, C. and Dubnicki, C.: Bimodal content defined chunking for backup streams, *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, Berkeley, CA, USA, USENIX Association (2010).
4) 村上じゅん, 石黒駿, 大山恵弘: Content-Defined Chunking を用いた重複除外キャッシュ機構の実装と評価, 情報処理学会研究報告ハイパフォーマンスコンピューティング, Vol. 2012-HPC-137, No. 4, pp. 1–7 (2012).
5) Xia, W., Jiang, H., Feng, D., Tian, L., Fu, M. and Wang, Z.: P-Dedupe: Exploiting Parallelism in Data Deduplication System, *Proceedings of the 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, NAS '12, Washington, DC, USA, IEEE Computer Society, pp. 338–347 (2012).
6) Bhatotia, P., Rodrigues, R. and Verma, A.: Shredder: GPU-accelerated incremental storage and computation, *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, Berkeley, CA, USA, USENIX Association (2012).
7) Kim, C., Park, K.-W. and Park, K. H.: GHOST: GPGPU-offloaded high performance storage I/O deduplication for primary storage system, *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, New York, NY, USA, ACM, pp. 17–26 (2012).
8) Ma, J., Zhao, B., Wang, G. and Liu, X.: Adaptive pipeline for deduplication, *Proceedings of 2012 IEEE 28th Conference on Massive Data Storage*, MSST '12, Washington, DC, USA, IEEE Computer Society, pp. 1–6 (2012).
9) Tang, Z. and Won, Y.: MultiChread Content Based File Chunking System in CPU-GPGPU Heterogeneous Architecture, *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing*, CCP '11, Washington, DC, USA, IEEE Computer Society, pp. 58–64 (2011).
10) Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google file system, *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, ACM, pp. 29–43 (2003).
11) Quinlan, S. and Dorward, S.: Venti: A New Approach to Archival Storage, *Proceedings of the First USENIX Conference on File and Storage Technologies*, FAST'02, Berkeley, CA, USA, USENIX Association (2002).
12) Rabin, M. O.: *Fingerprinting by random polynomials*, Center for Research in Computing Techn., Aiken Computation Laboratory, Univ. (1981).
13) Broder, A. Z.: Some applications of Rabin's fingerprinting method, *Sequences II*, Springer, pp. 143–152 (1993).
14) NVIDIA: CUDA - NVIDIA Developer Zone. `https://developer.nvidia.com/category/zone/cuda-zone`.
15) ISO/IEC 13818-1: Information technology — Generic coding of moving pictures and associated audio information: Systems (1994).
16) Wienke, S., Springer, P., Terboven, C. and an Mey, D.: OpenACC: first experiences with real-world applications, *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, Berlin, Heidelberg, Springer-Verlag, pp. 859–870 (2012).
17) Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware, *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, New York, NY, USA, ACM, pp. 777–786 (2004).
18) Han, S., Jang, K., Park, K. and Moon, S.: PacketShader: a GPU-accelerated software router, *Proceedings of the ACM SIGCOMM*

ⓒ 2013 Information Processing Society of Japan

2010 Conference, SIGCOMM '10, New York, NY, USA, ACM, pp. 195–206 (2010).

19) Jang, K., Han, S., Han, S., Moon, S. and Park, K.: SSLShader: cheap SSL acceleration with commodity processors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, Berkeley, CA, USA, USENIX Association (2011).

20) Gharaibeh, A., Al-Kiswany, S., Gopalakrishnan, S. and Ripeanu, M.: A GPU accelerated storage system, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, New York, NY, USA, ACM, pp. 167–178 (2010).

21) KHRONOS: OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

22) Ogata, Y., Endo, T., Maruyama, N. and Matsuoka, S.: An efficient, model-based CPU-GPU heterogeneous FFT library, *Proceedings of the 2008 IEEE 22th International Parallel and Distributed Processing Symposium*, IPDPS '08, Washington, DC, USA, IEEE, pp. 1–10 (2008).

23) Benner, P., Ezzatti, P., Quintana-Ortí, E. S. and Remón, A.: Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function, *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par '09, Berlin, Heidelberg, Springer-Verlag, pp. 132–139 (2010).

24) 坪内佑樹, 伊野文彦, 置田真生, 山川聡, 柏木岳彦, 萩原兼一: 重複排除ストレージのための SHA-1 計算システムの SSE 命令による高スループット化, 電子情報通信学会論文誌 D, Vol. J96-D, No. 10, pp. 2101–2109 (2013).

25) Aggarwal, B., Akella, A., Anand, A., Balachandran, A., Chitnis, P., Muthukrishnan, C., Ramjee, R. and Varghese, G.: EndRE: an end-system redundancy elimination service for enterprises, *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, Berkeley, CA, USA, USENIX Association (2010).

26) Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *J. ACM*, Vol. 46, No. 5, pp. 720–748 (1999).

ⓒ 2013 Information Processing Society of Japan