Vol.2015-OS-134 No.1
2015/8/4

データ圧縮型仮想マシン移送における GPU アクセラレーション

直井 由樹[†1,a)]　山田 浩史[†1,b)]

概要：
仮想マシン (VM) Migration は異なる物理マシン間で仮想マシン (VM) を移送する技術である．特に Live VM Migration と呼ばれる手法では VM を稼働させたまま移送することが可能であり，サーバ管理者は VM によるサービス提供を維持したままサーバ間の負荷分散やマシンメンテナンスを行うことができる．効率的な VM 移送を実現する手段の一つとして，データ圧縮型 VM Migration と呼ばれる手法が挙げられる．この手法は転送データの圧縮を行うことによって移送にかかる時間や負荷の削減を図るもので，移送対象 VM がメモリ書き換えの激しいワークロードを実行している場合のような，移送が長期化しやすい場合への対応も期待できる．本研究の目的は，データ圧縮型 VM Migration に対する，GPU アクセラレーションの有効性を検証することにある．そこで本論文では，GPGPU 環境を利用して転送データの圧縮・伸張処理を GPU 上で実行する VM Migration，*GMigrate* を開発する．GMigrate では圧縮・伸張処理を GPU へオフロードすることで，移送処理の更なる高速化と圧縮・伸張処理によって生じる CPU 負荷の軽減を図る．GMigrate では GPU へのオフロードの効果をより高めるため，移送処理のパイプライン化を利用した CPU と GPU の並列実行や，CPU・GPU 間での冗長な同期処理を防ぐスレッドスリープなどの仕組みを備えている．GMigrate のプロトタイプは Xen-4.2.1 をもとに 3 種類の圧縮手法について実装を行った．プロトタイプを用いた評価実験では，Xen のデフォルトの Live VM Migration と比べて，GPU オフロードによる移送時間や CPU 使用量の削減効果が，最も効果的な場合で最大 30%確認された．

1. Introduction

VM Migration とは異なる物理マシン間で仮想マシン (VM) を移送する技術であり，クラウド環境などにおいて柔軟なサーバ運用を可能にするものである．特に Live VM Migratation [1] とよばれる手法では，移送時に生じる移送対象 VM の稼働停止時間を非常に小さく抑えることで，VM によるサービス提供や外部からのネットワーク接続を維持しつつ VM を移送可能である．Live VM Migration の利用によって，リアルタイムでのサーバ間の負荷分散やサーバ統合が可能になる．

このようにサーバ管理者にとって有用な技術であるといえる Live VM Migration であるが，移送の際に時間的・資源的コストが大きくなることも少なくない．VM 移送時には VM のメモリ上のデータをはじめ，CPU のレジスタ値や各種デバイスの実行状態を移送先へ転送する必要があり，VM のメモリサイズが増大傾向にある現在の環境下では移送コストの増大が生じやすい．たとえば，Amazon EC2 [2] では数十 GiB にも及ぶメモリを搭載したインスタンスの提供が実際に行われている．また，標準的な Live VM Migration では移送中に書き換えのあったページを再送しなければならないため，メモリ書き換えの激しいワークロード下においては転送データ量はさらに増大する．

Live VM Migration が抱える転送データ量の増大という課題に対して，転送データの圧縮によって対処したものがデータ圧縮型 VM Migration である．この手法では，VM 移送時に移送元マシン上で転送データを圧縮してから転送を行い，移送先のマシンでそのデータを伸張することで，移送の際にかかる時間とネットワーク負荷を軽減する．また，拡張を行う部分は基本的にデータ転送時の圧縮処理のみであるため，移送対象 VM 上の OS やアプリケーションへの改変や制限といったものを必要としない点もこの手法の利点である．この手法はこれまで数々の方式が提案されるとともに，VM Migration を利用した先行研究において広く採用されてきた [3–8]．なお，実環境においては，KVM [9] の Live VM Migration 機構に対してゼロページ

†1 現在，東京農工大学
Presently with Tokyo University of Agriculture and Technology
a) naoi@asg.cs.tuat.ac.jp
b) hiroshiy@cc.tuat.ac.jp

を圧縮する機能が用意されている．

本研究の目的は，データ圧縮型 VM Migration における GPU アクセラレーションの有効性を検証することにある．データ圧縮型 VM Migration の処理の一部を GPU にオフロードすることによって，移送の短時間化と移送時の CPU 消費の低減が期待できる．移送の短時間化は，VM 移送時の負荷によって生じる移送対象 VM や同一マシン上の他 VM のパフォーマンスへの影響 [10] を抑えることにつながるため，サーバ管理者はより手軽に VM Migration を実行することができる．加えて，ネットワーク障害などによって移送が中断される可能性の低下にもつながるため，VM の移送処理自体の信頼性を向上させる効果も期待できる．またさらに，移送時の CPU 消費の低減によって，移送処理と VM のワークロードとの間での CPU 資源の競合を抑制することも可能である．

本論文では高速演算処理を目的とした汎用 GPU 演算 (GPGPU) を活用した VM Migration 機構，GMigrate の提案と実装を示していく．GMigrate では GPU を活用したデータ圧縮を行うプログラムを Live VM Migration へ統合し，転送データの大半を占める VM のメモリ上のデータの圧縮・伸張処理を GPU 上へオフロードさせる．圧縮・伸張処理を GPU へオフロードすることによって，データ圧縮型 VM Migration のアクセラレーションを実現するとともに，圧縮・伸張処理自体によって生じる CPU 負荷の増大への対処を図る．

GMigrate では，GPU プログラミングで培われた処理手法を移送処理に適用し，CPU と GPU の非同期的実行を行っている．データ圧縮型 VM Migration の処理を分割・パイプライン化することで，CPU と GPU が並列的に動作可能な仕組みをとっている．移送元・先マシンにおける移送処理は次の 3 つのフェイズに分けることができる．1) Fetch フェイズ：ハイパーバイザが移送対象 VM のページを抽出し，VM Migration プロセスのアドレス空間にコピーするフェイズ，2) Serialize フェイズ：ハイパーバイザによってコピーされてきたデータをシリアライズするフェイズ，3) Transfer フェイズ：シリアライズされたデータを移送先へと転送するフェイズ．転送データの圧縮処理は Serialize フェイズにて GPU 上で実行され，Fetch フェイズで抽出されたデータは Serialize フェイズで圧縮処理が施されたのちに，Transfer フェイズにて移送先へ転送される．通常の VM Migration ではこれら 3 つのフェイズを一定ページごとに反復的に繰り返す．一方 GMigrate では，これら 3 つのフェイズを複数スレッドを用いてパイプライン処理する．Fetch フェイズを担当するスレッドは，他のスレッドが GPU 上でのデータの圧縮や移送先への転送を行っている間も，次のデータの取得に移ることができる．なお，Serialize フェイズを担当するスレッドは，GPU との同期を行って圧縮処理が終了したことを確認し，圧縮済みのデータを GPU 上からコピーしてくる必要がある．この同期処理の際に CPU 側でビジーウェイトが発生するため，GPU へのオフロードによる CPU 負荷の削減効果を阻害してしまう．この問題に対処するため，GMigrate では圧縮処理時間の予測とその予測に基づくスレッドスリープを行い，冗長な同期処理によるビジーウェイトを防ぐことで，より効率的な GPU オフロードを実現している．

GMigrate のプロトタイプの実装は Xen-4.2.1 をもとに行った．GPU を活用したデータ圧縮型 VM Migration の実装例として，RLE 法によるデータ圧縮を行うもの (GM-RLE)，Delta Compression [7] によるデータ圧縮を行うもの (GM-DeltaC)，Deduplication によるデータ圧縮を行うもの (GM-RLE)，の 3 つの圧縮アルゴリズムについて実装を行った．また，複数のワークロードを用いた評価実験を行い，GMigrate の性能評価を行った．評価実験の結果，GPU への圧縮処理のオフロードによって，移送時間や CPU 使用量の削減されることが確認された．

本論文の構成は以下のような形をとっている．まず，第 2 章では本研究の背景について述べる．続いて第 3 章，第 4 章では提案手法とその実装について記述する．そして第 5 章では提案手法の実装例についての解説と評価実験の結果を示す．第 7 章で関連研究について触れたのにちに，第 8 章では最後に本論文のまとめと今後の課題について述べていく．

2. Background

2.1 Live VM Migration

Live VM Migration とは，移送の際に生じる移送対象 VM の停止時間 (ダウンタイム) を非常に短い時間に抑えた VM Migration である [1]．ダウンタイムが非常に短かく抑えられているため，VM のサービス提供を継続しつつ VM を移送することが可能である．Live VM Migration には複数の方式が存在するが，pre-copy 方式と呼ばれるものがその中でも特に一般的な方式である．pre-copy 方式では iterative-copy フェイズと stop-and-copy フェイズの二つの段階に分けて VM の移送を行う．iterative-copy フェイズでは VM を稼働させつつ VM のメモリ上のデータの転送を行う．このフェイズでは一度 VM の全ページを転送し，その後はページ転送中に書き換えのあったページを反復的に再送していく．再送が必要なページ数が十分に小さくなるか，設定された期間内でページ再送が収束しない場合には stop-and-copy フェイズに入る．stop-and-copy フェイズでは VM の稼働を停止し，移送先と元のマシン間で VM の状態の完全な同期をとる．stop-and-copy フェイズで転送されるデータには，iterative-copy フェイズで再送しきれなかったページや CPU のレジスタ値，デバイスの実行状態などがあたる．stop-and-copy フェイズ終了後は，移送元マシン上の VM は破棄され，移送先マシン上で VM の

稼働が継続される．

pre-copy 方式では，移送中に書き換えのあったページを逐次再送する必要があるため，移送対象 VM のワークロードがメモリ書き換えの激しいものである場合には移送時間の長期化が生じる．また，移送時間が長くなるだけでなく，再送しきれなかったページは stop-and-copy フェイズで転送することになるため，移送における VM のダウンタイムの増大が生じる可能性もある．また，VM の全ページを一度は転送する必要があるため，VM のメモリサイズが大きくなるほど移送時間が長期化することになり，再送ページ数が増大しやすくなる．

本論文で提案する GMigrate は pre-copy 方式の Live VM Migration を前提とした実装となっている．pre-copy 方式は Xen や KVM，VMWare など多くのハイパーバイザで広く採用されている．pre-copy 方式では移送中も移送元マシン上で VM を稼働させ続けることができ，移送が途中で中断された場合にも VM の稼働を継続できる．そのため，その他の post-copy 方式 [11] や双方を組み合わせ方式 [12] に比べて，信頼性を維持しつつ VM の移送を行うことが可能である．

2.2 Compression-based VM Migration

pre-copy 形式が抱えるメモリ転送に関する課題への対策の一つとして，データ圧縮型 VM Migration が挙げられる．この方式では転送データを移送元マシン上で可逆圧縮してから転送し，移送先でその転送データを伸長することで転送データ量の削減を図る．VM の移送に要する時間のほとんどは移送先マシンへのデータ転送部分であるため，転送データ量を小さく抑えることは移送時間の短縮につながる．データ転送に要する時間が短くなることで，転送中に書き換えられるメモリの総量を小さく抑えることができるため，ダウンタイムの短縮といった面でも効果が期待できる．また，転送データ量の縮小は移送時間の削減だけでなく，移送によって生じるネットワーク負荷を小さくすることにもつながる．したがって，同一マシン上でネットワークへの依存が大きなワークロードが動作している場合には，それらのワークロードに対して移送処理が及ぼす影響を抑制することにもつながる．なお，転送データ量の削減による恩恵だけでなく，移送対象 VM 上の OS やアプリケーションに改変が必要ないということもこの手法の利点の一つである．

2.3 GPGPU

汎用 GPU 演算 (GPGPU) とは，従来のような描画処理のみならず，一般的な数値演算に対しても GPU の演算能力を応用していく技術・手法のことである．昨今の GPU は数百から数千にも及ぶプロセッサコアを搭載し，それらを並列的に使用することで非常に強力な並列演算処理が可能である．GPGPU の主な目的は GPU が持つこの高度な演算能力を活かしていくことである．近年では CUDA [13] をはじめとした，高度な GPGPU 環境を提供可能なライブラリも登場し，さまざまなアプリケーションにおける GPU の活用が盛んに行われている．研究段階ではあるものの，GPGPU の活用用途は多岐にわたり，バックアップサーバシステム [14,15] やパケット処理などのネットワークレイヤ [16,17] における GPU 利用の試みが行われている．

本論文で提案する GMigrate では，データ圧縮型 VM Migration における GPGPU の活用を試みている．GMigate では，データ圧縮型 VM Migration における転送データの圧縮・伸張処理を GPU へオフロードすることで，データ圧縮型 VM Migration の恩恵の向上を図っている．GPGPU の適用によって得られるメリットは主に二点である．一点目は圧縮・伸張処理の高速化に伴う移送時間の短縮，二点目は移送時に消費される CPU 資源の削減である．GPU アクセラレーションによる移送時間の短縮は，VM Migration におけるページ反復転送の間隔を短くし，メモリ書き換えの激しいワークロードへの耐性を高めることにつながる．また，GPU への処理のオフロードによって，圧縮処理に使用される CPU 資源の節約が可能になり，その分の CPU 資源を移送処理以外に使用することができる．

また，VM Migration を利用したデータセンタ運用における GPU の活用を動機付ける要因の一つとして，GPU の演算資源を利用可能なクラウドサービスの提供が広まりを見せていることが挙げられる．例えば，AmazonEC2 においては CUDA アプリケーションを利用可能な，GPU 搭載インスタンスの提供を開始している [18]．このようなサービスがより普及し，GPU を搭載したデータセンタの登場が盛んになることで，VM Migration を用いたサーバ運用における GPU の活用がより重要性を増していくものと考えられる．

3. GMigrate

本論文では，データ圧縮型 VM Migration におけるデータ圧縮処理を GPU へオフロードした VM Migration，GMigrate を提案する．ハイパーバイザによって抽出された VM のメモリ上のデータは一度 GPU 上へコピーされ，GPU 上で圧縮処理がなされた後に再度 CPU 上 (ハイパーバイザのアドレス空間) にコピーされ，移送先へと転送される．通常の VM Migration においてボトルネックとなるデータ転送処理のオーバヘッドを，データ圧縮による転送量の削減によって軽減するのがデータ圧縮型 VM Migration である．しかしながら，圧縮効果が十分に高い場合にはネットワークのボトルネックが小さくなり，圧縮処理自体がボトルネックとなる場合が生じる．GPU アクセラレーションによる圧縮処理の高速化によって，圧縮処理がボトルネックとなるのを防ぐことで，データ圧縮型 VM Migration を

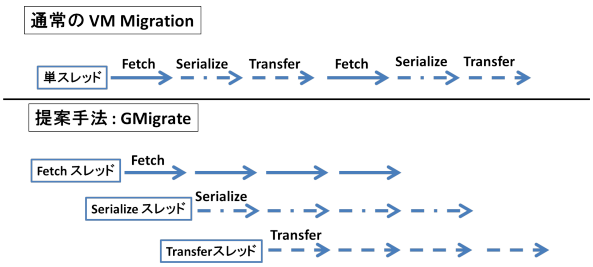

図 1　移送処理のパイプライン化

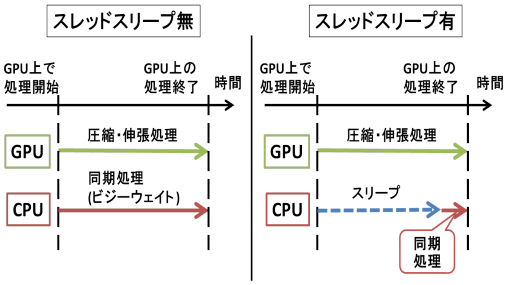

図 2　スレッドスリープ

より効果的なものとすることができる．また，GPU へのオフロードを行うことで圧縮処理によって生じる CPU 負荷を軽減する効果も見込める．

3.1　移送処理のパイプライン化

GMigrate では，CPU と GPU に非同期的な実行を行わせるため，移送処理の分割とパイプライン化を行っている．GPGPU 環境においては，CPU と GPU はそれぞれ独立して処理を実行可能である．したがって，効率的に GPU アクセラレーションを実現するためには，CPU と GPU の非同期的な実行が非常に重要である．

移送元マシン上での移送処理は 3 つのフェイズに分けることができる．1) ハイパーバイザが VM のメモリ上のデータを一定ページ分抽出し，ハイパーバイザのアドレス空間上にコピーするフェイズ (Fetch フェイズ)．2) Fetch フェイズで抽出したデータをハイパーバイザがシリアライズするフェイズ (Serialize フェイズ)．3) シリアライズされたデータを移送先へ転送するフェイズ (Transfer フェイズ)．ハイパーバイザはこの 3 つのフェイズを一連の流れとして一定ページごとに反復することで VM の移送が行われる．移送先マシン上での処理も移送元の場合と同様に，以下の 3 つのフェイズに分割することが可能である．1) 移送元マシンから転送されたデータを受信するフェイズ (Receive フェイズ)．2) 受信したデータをデシリアライズするフェイズ (Deserialize フェイズ)．3) デシリアライズされたデータをもとに VM を復元するフェイズ (Restore フェイズ)．データ圧縮型 VM Migration におけるデータ圧縮処理部分は 3 つのフェイズのうち，Serialize フェイズと Deserialize フェイズがそれに該当する．ハイパーバイザによって抽出された VM のメモリ上のデータはシリアライズの過程で圧縮処理が施され，移送先マシンへと転送される．移送先マシン上ではデシリアライズの過程で受信データの伸張が行われる．

GMigrate では複数スレッドによる並列実行によって，これら 3 つのフェイズの処理のパイプライン化を行う．図 **1** はパイプライン化した場合の処理フローを示したものである．Serialize フェイズ，Deserialize フェイズでは処理の大半を実質的に GPU が実行し，CPU はメタデータの処理時を除いて GPU との同期処理を行う．その他のフェイズは CPU が処理を実行する．パイプライン化によって CPU は GPU との同期を行っている間も，別スレッドで CPU が独立して処理を実行できるため，実質的に CPU と GPU の非同期的な実行が実現される．

なお，移送処理のパイプライン化による並列実行は，PMigrate [19] においてよりきめ細やかな処理の分割が行われており，特別新しいものではない．しかしながら，本論文では CPU と GPU の並列実行を目的としたパイプライン化を行っており，複数 CPU 間を用いた高い粒度での並列実行を目的とした PMigrate のものとは異なる目的のもとに実装されている．

3.2　スレッドスリープ

圧縮または伸張処理が完了したデータを GPU 上から CPU 上にコピーする際には，GPU との同期を行い，GPU 上での処理が完了したことを確認した上でデータをコピーしなければならない．しかしながら，GPU との同期処理時に CPU はビジーウェイトが生じてしまう．したがって，冗長な同期処理は圧縮処理時に必要となる CPU 資源の無駄な増大を招き，GPU へのオフロードによる CPU 不可の軽減という恩恵を著しく弱める結果を引き起こしてしまう．

そこで GMigrate では，GPU 上における圧縮処理の実行時間予測に基づくスレッドスリープを行っている．図 **2** はその実行イメージである．CPU は，GPU 上の処理の開始後ただちに同期処理のビジーウェイトに入るのではなく，一定時間のスリープの後に GPU との同期処理を開始する．このスレッドスリープによって同期処理時に生じるビジーウェイトの時間を短縮し，冗長な CPU 資源使用の抑制を行っている．圧縮処理の実行時間の予測にあたっては，反復的に実行される GPU 上での圧縮処理の実行時間をサンプリングし，サンプリングされた実行時間のうち，最も短かったものをスリープ時間として用いる．最も短かった時間を使用する理由は，圧縮対象データの内容によって圧縮速度にばらつきが出ることが予想されるためである．圧縮に時間がかかってしまった場合の実行時間をスリープ時間として使用した場合，冗長なスレッドスリープが生じ，移送処理全体のスループットを低下させてしまう可能性がある．したがって最も短時間だったサンプルをスリープ時間として使用することで，冗長なスリープを極力排除する

方式をとっている．スリープ時間の決定にはサンプリングした実行時間の平均をとるなどの方式も考えられるが，現在の実装では，ビジーウェイトの削減よりも移送全体のスループットの向上を目指した設計をとっている．実行時間のサンプリングは移送開始から数十回目までの圧縮処理を対象に行い，それ以降の圧縮処理時には予測された時間に基づくスレッドスリープが適用される．

4. Implementation

GMigrate のプロトタイプの実装は Xen-4.2.1 をベースに行った．Xen [20] は実環境において広く用いられているハイパーバイザの一つであり，Amazon EC2 などのクラウドプラットフォームにおいて利用されている．Xen-4.2.1 では LAN 内での移送を前提とした pre-copy 形式での Live VM Migration を提供しており，メモリと CPU のレジスタ値の転送を行う．GMigrate は Xen のこの Live VM Migration プロセスを拡張することによって実装を行っている．GPU 上でのデータ圧縮プログラムは CUDA [13] を用いて作成されていて，このプログラムを移送プロセス中で実行することで GMigrate は実現される．なお，この CUDA プログラムの実行は，洗練された GPGPU 環境を提供する Gdev [21] ライブラリと Linux のメインストリーム GPU ドライバである Nouveau を介して行われる．

VM Migration のパイプライン化は POSIX 準拠のマルチスレッディングをもとに実装を行っている．パイプライン化にあたっては 3 スレッドを用意し，それぞれは 3.1 項で記述した 3 つの各フェイズに対応している．各スレッドは共有の作業用バッファを保持しており，それぞれがこのバッファに対して自身の担当処理を行っていく．Fetch スレッドは VM のメモリ上のデータを 1024 ページごとに取得し，共有バッファへと保存していく．Serialize スレッドは共有バッファ内のデータの圧縮を Gdev ライブラリを介して実行する．Serialize スレッドは対象データを GPU へ転送したのちに同期を行い，GPU 上における圧縮処理の終了を確認したら圧縮後のデータを GPU 上から共有バッファへコピーし，バッファの内容を更新する．そして Transfer スレッドは，Serialize スレッドによって圧縮処理が施された共有バッファのデータを移送先へと転送する．現在の実装では 3 スレッドによるパイプライン化を効率的に機能させるため，共有バッファを 3 つ用意する形をとっている．バッファを 3 つ用意することで，各スレッドは他スレッドの処理の終了を待つことなく，次のバッファを使用することで即座に作業に移ることが可能である．ここで述べた一連の流れは移送元マシン上における処理についてであるが，移送先マシン上でも各フェイズに対応した 3 スレッド (Recieve，Deserialize，Restore スレッド) によるパイプライン処理が行われる．Receive スレッドは受信したデータを共有バッファへとコピーし，Deserialize スレッドはそのデータの伸張処理を GPU 上で実行する．そして伸長されたデータは Restore スレッドによって VM のページとして復元されていく．

第 5 章では，データ圧縮型 VM Migration への GPU 適用例として，GM-RLE，GM-DeltaC，GM-Dedup の 3 種類の実装パターンを示す．それぞれの圧縮アルゴリズムについて述べるとともに，GPU を活用した実装についての記述を行い，それらの性能を調査した評価実験の結果についても述べていく．評価実験は 2 台の物理マシン間で移送を実行することで行う．各物理マシンは Intel Xeon CPU E5-2479 2.3GHz を 8 コア搭載し，各コアはハイパースレッディングによって実質的に 2 コア分の処理能力を保持している．物理メモリは 32GB を搭載し，圧縮・伸張処理用の GPU としては NVIDIA Quadro 6000 を搭載している．物理マシン間はギガビットイーサネットで接続されているが，Xen の Live VM Migration プロセスでは SSH プロトコルを用いてデータ転送を行う関係で，移送時のデータ転送スループットは実質的には 58MB/s 前後に制限される．各マシン上のドメイン 0 とドメイン U のカーネルには Linux 3.13.0 を使用している．また，移送対象 VM が使用するディスクイメージは NFS サーバを介して両ホスト間で共有されている．

5. Case Studies

5.1 Case Study : GM-RLE

5.1.1 GM-RLE

GM-RLE では，RLE 法による転送データの圧縮を GPU 上で行いつつ，VM のページの移送を行う．RLE(Run Length Encoding) 法とは非常に単純なデータ圧縮アルゴリズムの一つであり，ある数値が連続したデータを数値とその個数に変換することでデータサイズの縮小を行うものである．例えば，“00001” というデータを RLE 法を用いて圧縮を行うと，“041” というデータに変換される．なお，GM-RLE では一つ前にあるデータとの差分をとった後に，数値 0 にのみ注目して RLE 法を適用する形をとっている．例えばこの場合，“11113” というデータは “10002” と変換され，その後に “1032” というデータに圧縮される．この形をとることで圧縮効果をやや落とす代わりに圧縮処理中に生じる条件分岐を減らし，GPU が不得意とする条件分岐命令による処理時間の増大を防いでいる．

5.1.2 Implementation

GM-RLE のプロトタイプは，CUDA を利用して作成した RLE 法によるデータ圧縮プログラムを，Gdev ライブラリによって提供されるインタフェースを介して VM Migration で実行する形で実装されている．Serialize スレッドは Fetch スレッドによってフェッチされてきた VM のページを 256Byte ごとのチャンクに分割し，各チャンクに対して 1 つの GPU のスレッドが対応するように GPU 上でタ

スクスレッドを用意し実行する．現実装ではハイパーバイザによって一度にフェッチされるページ数は 1024 ページ (4MB) であるため，GPU 上では 16K スレッドが一度の圧縮処理で動作する．伸張処理時にも同数のスレッドが動作し，圧縮処理時に一つのスレッドが圧縮したデータを伸長するためのスレッドが必ず一つ存在する．圧縮されたデータは Transfer スレッドによって移送先へ転送される．移送先では移送元で圧縮処理を行った場合とは逆順で処理が進み，受信されたデータは伸張処理が施された後に VM のメモリとして復元されていく．

圧縮処理においては各スレッドは自身のスレッド ID をもとにデータの読み込み開始位置を特定し，圧縮処理を開始することが可能である．しかしながら，伸張処理時には圧縮側のスレッドごとに圧縮後のデータサイズがまちまちであるために，伸張側の各スレッドはデータ読み込み開始位置を一意に判別することができない．そこで GM-RLE では伸張処理時用のヘッダデータを各スレッドに対して一つ用意している．CPU へ圧縮後のデータをコピーする前に，各圧縮スレッドはそのヘッダデータに各自の圧縮後のデータサイズを記録する．CPU 側では圧縮データの転送の前にこれらのヘッダデータに記録された情報をもとに，ヘッダデータの内容を伸張時の各スレッドの読み込み開始位置に変換を行う．伸張スレッドはそれぞれに対応したヘッダデータを読み込み，自身が伸張処理を開始するアドレスを特定したうえで伸張処理を開始する．なお，この方式は GM-RLE に限ったものではなく，GM-DeltaC, GM-Dedup においても同様の実装がなされている．

RLE 法のアルゴリズムの関係上，圧縮処理後のデータサイズが元のサイズよりも大きくなる場合がある．この問題に関してはヘッダデータの上位ビットをサイズ超過フラグとして使用することで対処している．圧縮処理後のデータサイズが元のデータよりも大きくなってしまった圧縮スレッドは，ヘッダデータのサイズ超過フラグを有効にし，圧縮処理前のデータをそのまま転送する．伸張処理時にはこのフラグをチェックし，フラグが有効である場合には伸張スレッドは特別な処理を行わず，そのデータをそのまま伸張データとして用いる．

5.1.3 Experiments

GM-RLE の基本的な性能を評価し，GPU へのオフロードによる効果を調べるため，移送対象 VM に 3 種類のワークロードを課した場合の移送時間を計測した．なお，ダウンタイムの結果については今回は割愛している．GM-RLE はページ再送部分での使用を想定したものではないため，stop-and-copy フェイズで圧縮処理を行わないため，ダウンタイムにほとんど差が生じないためである．3 種類のワークロードのうち 2 種類は VM のメモリを任意の状態で固定する単純なワークロード，残りの一つは Memcached と呼ばれるインメモリデータベースと Twitter を模したデータセット [22] を用いた，実環境を意識したワークロードとなっている．2 種類の単純なワークロードのうち，VM のメモリ上のデータのほとんどを RLE 法が最も機能する内容で埋め尽くした状態のものを Best ワークロード，全く機能しない内容で埋め尽くした状態のものを Worst ワークロードとここでは呼ぶようにする．この二つのワークロードを用いて，圧縮が十分に機能する場合としない場合における GPU オフロードの効果の検証を行う．なお，この二つのワークロードでは移送中にメモリ書き換えがほとんど発生しないため，再送されるページ数は非常に少ないものとなっている．一方 Memcached を使用したワークロードでは，メモリ上に展開されたデータベースの内容がしきりに更新されるため，非常に多くの再送ページが生じる．このワークロードでは VM のメモリの約半分を用いてデータベースを展開し，そのデータベースに対して移送元・先マシンとは独立したクライアントマシンから，読み込みと書き込みリクエストが 9:1 の割合で発行される．

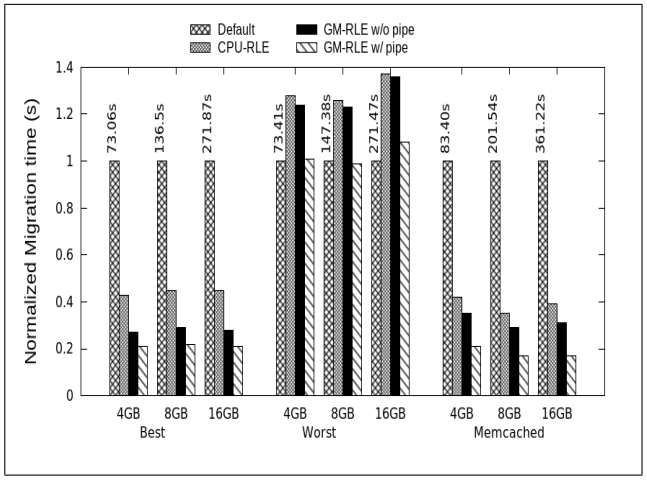

図 3　各パターンの移送時間 (GM-RLE)

計測項目は移送時間，ネットワーク転送量，CPU 使用量とし，VM のメモリサイズを 4GB，8GB，16GB と変化させながら計測を行っていく．比較を行う移送方式としては，Xen のデフォルトの Live VM Migration，CPU 上で RLE による圧縮を行う Live VM Migration(CPU-RLE)，パイプライン化を無効にした GM-RLE(GM-RLE w/o pipe)，パイプライン化を有効にした GM-RLE(GM-RLE w/- pipe) の 4 パターンで比較を行う．なお, 特に述べない限りは圧縮処理時間の予測に基づくスレッドスリープは常に GM-RLE に適用されているものとする．

測定された移送時間を示すグラフを図 3 に示す．GM-RLE は Best と Memcached を使用したワークロードにおいて効果を発揮しているがうかがえる．Best ワークロードでは VM のメモリ上のデータのほとんどが同一数値の連続で満たされているため，RLE 法によるデータ圧縮が十分に効果を発揮しているものと考えられる．CPU-RLE でも約 60%の移送時間の短縮に成功しているが，GM-RLE では

GPU アクセラレーションによってさらに 20%前後の短縮，そこからさらにパイプライン処理によって約 10%の移送時間の短縮効果を得られることが分かる．Memacached を用いたワークロードでも近い結果が得られているが，この結果は memcached によって使用されていないページや，再送されたページ内で使用されていない領域などで RLE が効果を発揮しているためであると考えられる．逆に Worst ワークロードでは圧縮処理を行ってもデータ量が減少しないため，圧縮処理がそのままオーバヘッドとなり，移送時間の増大が生じている．しかしながら，CPU-RLE ではデフォルトよりも 20%ほどの増大が見られるのに対して，パイプライン化を有効にした GM-RLE ではデフォルトと同等の移送時間に抑えられている．なお，VM のメモリ量が変化しても結果の傾向に差が見られないことから，メモリ量に対してある程度スケールして効果を発揮することが確認できる．

表 **1** は VM のメモリを 8GB とした場合に，実験から得られた各パターンの資源使用状況を示すものである．Worst 以外の二つのワークロードでは，圧縮処理の拡張によって平均ネットワーク転送量に減少が見られている．GM-RLE では CPU を用いた場合に比べて圧縮データの生成速度が高速であるため，転送スループットの向上が見られる．一方で Worst の場合には圧縮処理がボトルネックとなるため，圧縮処理の拡張によって転送スループットの低下が見られる．しかしながら，パイプライン化を有効にした GM-RLE では処理の効率化によってデフォルトと同等のスループットを実現している．CPU 資源の使用に関する GPU へのオフロードによる効果は，全てのワークロードにわたってほぼ同様の傾向が表れている．CPU-RLE とパイプライン化を行わない GM-RLE を比べてた場合，単位時間あたりの CPU 使用率に減少が生じている．さらにパイプライン化を有効にした場合には，単位時間あたりの CPU 使用率に若干の増大が生じる一方で，全体の処理時間が短くなるため総 CPU 使用量には低減が見られる．Worst については結果的にデフォルトよりも性能が低下してしまっているが，オフロードによる効果は他のワークロードと同様である．これらの結果より，圧縮処理が全く機能しない場合であっても，GPU へのオフロードによってそのオーバヘッドを小さく抑えることが可能であると考えられる．なお，他のメモリサイズの場合においても見られる傾向は同様である．

表 **2** はスレッドスリープを有効にした場合と無効にした場合の移送時間と資源使用量の差を示すものである．スレッドスリープによって，平均 CPU 使用率，総使用量ともに低減されていることがわかる．特に Memcached をワークロードとした場合の低減効果は大きく，平均 CPU 使用率に 40%以上の低下が見られる．一方で移送時間には大きな差は見られないことから，冗長なスレッドスリープは非常に少ないことが確認できる．

5.2 GM-DeltaC

5.2.1 Delta Compression

DeltaCompression [7] では，移送中に転送したページの古いバージョンをキャッシュしておき，次回以降の同一ページの転送時にはその差分のみを転送することでデータ転送量の縮小を行う．移送先マシン上にも同様のキャッシュが保存されており，転送されてきた差分とキャッシュのデータをもとに最新の状態のページの復元を行う．

実装上は排他的論理和 (XOR) と RLE 法によるデータ圧縮を利用したものとなっており，この方式は XOR binary RLE(XBRLE) として元となった論文中で提案されている．この方式ではまず，キャッシュと最新の状態のページの XOR をとることにより双方の共通部分が 0 ページとして変換される．次に，XOR を取ったこのデータに対して RLE 法による圧縮を行うことによって，0 ページとなっている共通部分が削除され，差分データのみが抽出される．移送先ではこの逆の流れで処理が行われ，RLE 法に基づく伸張を行った後にそのデータとキャッシュの XOR をとることで元データの復元がなされる．GM-DeltaC はこのアルゴリズムを用いて Delta Compression を実現している．

5.2.2 Implementation

GM-DeltaC についても GM-RLE と同様に，移送プロセス内で CUDA による圧縮プログラムを Gdev ライブラリを利用して実行することで実装されている．圧縮・伸張処理に関しては，基本的に GM-RLE を発展させる形で実装されている．XOR をとった後のデータは共通部分が 0 として出力されるため，その後の RLE 法による圧縮処理については GM-RLE をそのまま流用することが可能である．GM-DeltaC では圧縮処理を GPU 上で行うため，ページの古いバージョンを保存するためのキャッシュ領域は GPU 上に確保する形をとっている．キャッシュは反復的なページ圧縮を行っていく過程で自動的に蓄積されていく．キャッシュに保存されていないページに関しては単純な RLE を行った後にキャッシュへ追加していく．すでに保存されているページに対しては XBRLE による差分圧縮が適用される．なお，キャッシュの形式には 2-way セットアソシエイティブ形式 [23] を採用し，キャッシュサイズは 512MB に設定して実装を行っている．したがって，ページサイズは 4KB となっていることから，GPU 上のキャッシュは全 128K エントリから成る 2-way セットアソシエイティブ形式で管理されたキャッシュとなる．

キャッシュに保存されているページの管理には，ハイパーバイザによって設定された各ページのページ番号と，キャッシュに関するメタデータを管理するテーブルを使用する．管理テーブルは GPU 上に確保された領域に保存されていて，現在キャッシュされているページのページ番号

表 1　各パターンの資源使用量 (GM-RLE)

	Best			Worst			Memcached		
	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]
Default	57.76	102.73	15040	57.38	101.79	15002	56.11	104.55	21073
CPU-RLE	6.39	71.76	4481	45.25	103.64	19309	15.73	91.16	6489
GM-RLE w/o pipe	9.95	64.52	2579	46.49	94.05	17159	20.58	75.78	4541
GM-RLE w/- pipe	12.92	79.73	2443	57.20	115.16	16964	31.67	119.96	4290

表 2　スレッドスリープによる効果 (GM-RLE)

	Best			Worst			Memcached		
	移送時間 [s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	移送時間 [s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	移送時間 [s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]
GM-RLE w/o sleep	30.53	136.45	4166	147.29	125.23	18446	35.59	165.83	5902
GM-RLE w/- sleep	30.64	79.73	2443	147.30	115.16	16964	36.19	119.96	4290

や圧縮処理時のヒット状況の確認を行うためのメタデータなどを管理している．各ページのページ番号は，ページ自体のデータとともにヘッダデータに付随する形で GPU 上に転送される．

GM-DeltaC による圧縮処理は三段階に分けて GPU スレッドによって並列的に実行される．一段階目ではページのキャッシュヒットの確認を行う．CPU 上から転送されてきたヘッダデータとページ番号をもとに，各 GPU スレッドは同じページ番号を持つキャッシュが存在するのかをチェックする．このとき，キャッシュのヒット状況は各ページに対応したヘッダデータに記録される．二段階目ではヘッダデータに記録されたキャッシュのヒット状況をもとに各ページの圧縮処理を行う．ヒットしたページには XBRLE による圧縮を，ヒットしなかったページには RLE による圧縮を適用する．圧縮前のデータはキャッシュに保存する必要があるため，圧縮時には入力用と出力用の二つのバッファを用意し，圧縮処理後も元データは入力用バッファの中に保存されている状態となっている．三段階目ではキャッシュと管理テーブルの更新処理を行う．各 GPU スレッドはキャッシュのヒットの有無にかかわらず，各ページに対応したキャッシュ領域をそのページの最新状態で更新する．また，管理テーブルについてもその時点でのキャッシュの状況を正確に反映した状態になるように更新処理がなされる．これら 3 つの処理は GPU スレッド間でのキャッシュ更新領域の衝突を防ぐ必要があるため，各段階の終了ごとに全 GPU スレッド間で同期をとってから次の段階へと進むような実装がなされている．

2-way セットアソシエイティブ形式でのキャッシュにおいては，ページ番号が異なるページ間でもキャッシュの保存位置に衝突が生じ，GPU スレッドによる並列的な更新処理の際に妨げとなる可能性が考えられる．GM-DeltaC では GPU スレッド間でのキャッシュ更新領域の衝突を防ぐため，そのキャッシュエントリを更新使用としているスレッドの数をカウントする仕組みを管理テーブル上に用意している．キャッシュヒットの確認処理の終了時点で各キャッシュエントリの更新を行おうとするスレッド数が管理テーブルのメタデータとして保存される．このスレッド数の値からキャッシュ更新領域の衝突が予想される場合には，競合している一部の GPU スレッドに対して，キャッシュ更新を行わないようにさせる処理を行う．競合スレッドのどのスレッドに更新を許可するのかは，各スレッドに対応したヘッダデータ領域の値を用いて調整を行う．

5.2.3 Experiments

GM-Delta の評価実験には LMbench [24] と，GM-RLE の評価にも使用した Memcached と Twitter を模したデータセットを用いたワークロードを用いる．LMbench で使用すのは bw_mem と呼ばれるベンチマークで，単純なメモリ書き換えをメモリ上の一定の領域で繰り返し行う処理を行うワークロードであり，Delta Compression [7] の論文中の評価実験をもとに設定を行っている．VM のメモリ量を 4GB, 8GB, 16GB と変化させながら, 移送時間, ダウンタイム, ネットワーク転送量, CPU 使用量の測定を行う．比較を行う移送方式は，Xen のデフォルトの Live VM Migration, CPU 上で圧縮を行うもの (CPU-DeltaC)，パイプライン化を無効にした GM-DeltaC(GM-DeltaC w/o pipe)，パイプライン化を有効にした GM-DeltaC(GM-DeltaC w/- pipe) の 4 パターンで比較を行う．なお，Delta Compression が有効にはたらくのはページの再送時であるため，Live VM Migration がページの再送フェイズに入った段階から圧縮処理を開始するように設定を行っている．したがって，各ページ初回の転送時にはいかなる圧縮処理やキャッシュへの保存も行わない．

図 4 と表 3 は各ワークロードを与えた際の移送時間，ダウンタイムをまとめたものである．どちらのワークロードもメモリの書き換えが激しいため移送時間，ダウンタイムの増大が生じている．特に LMbench ではメモリの書き換えが非常に多く，数十秒のダウンタイムが生じている．Delta Comression によるデータ圧縮やパイプライン

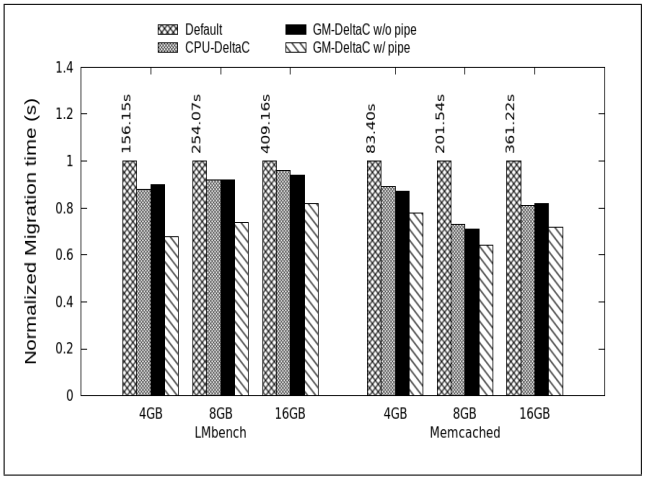

図 4　各パターンの移送時間 (GM-DeltaC)

表 3　各パターンのダウンタイム (GM-DeltaC)

	ダウンタイム [s]					
	LMbench			Memcached		
	4GB	8GB	16GB	4GB	8GB	16GB
Default	34.83	75.55	75.49	0.01	0.02	0.03
CPU-DeltaC	15.54	48.97	49.03	0.15	0.16	0.35
GM-DeltaC w/o pipe	14.37	46.12	33.36	0.03	0.04	0.25
GM-DeltaC w/- pipe	6.19	17.42	16.49	0.02	0.03	0.20

化による短時間化が大きく表れていることは確認できるが，GPU オフロードによる効果は微小なのものであり，CPU-DeltaC とパイプライン化を行わない GM-DeltaC の間にはほとんど差がない．圧縮処理が比較的近い GM-RLE ではオフロードの効果が確認できたことを考えると，差分抽出のために用いられるキャッシュ関連の処理が影響しているものと考えられる．まず，キャッシュの更新の衝突の調整処理は CPU 上で実行する必要があり，これは GPU へオフロードした場合にはそれによるオーバヘッドが生じる．また，キャッシュの更新処理はデータコピーが主体の処理であるため，GPU スレッドによって並列的にデータの移動が可能であるとはいえ，GPU の高い演算能力を発揮できるような処理でないこともオフロードの効果が弱い原因ではと考えられる．差分を求めるために用いられる XOR 演算も非常に単純な演算処理であり，アクセラレーションの効果は小さいものと考えられる．キャッシュ更新の衝突の管理は GPU を使用するうえで避けることのできないものであり，CPU との連携部分についてより効率的な方法を模索する必要があると考えられる．また，ダウンタイムが短時間で済んでいる Memcached の場合においては，Delta Compression の拡張によってダウンタイムが 10 倍以上増大している．このダウンタイムの増大は，対象となる転送データのサイズが十分に小さいにもかかわらず圧縮を行っているため，圧縮処理のオーバヘッドが目立つ形となった結果であると考えられる．ページの初回転送時での圧縮処理を想定した GM-RLE や GM-Dedup と比べ，GM-DeltaC はページの再送時，stop-and-copy 時のページ転送での使用が主であるため，ワークロードによってはこのオーバヘッドが無視できないものになる．特に GPU を使用する場合には CPU-GPU 間でのデータコピーによる固定的なオーバヘッドが存在するため，この問題はより重要である．単純な対策として，stop-and-copy フェイズに入るまでのデータ転送量を観察し，ワークロードによるメモリに書き換えに対してデータ転送速度が十分に高速で，stop-and-copy フェイズでのデータ転送量が小さくなると予想されるときには圧縮処理を無効にするといった方法が考えられる．

表 4 は VM のメモリを 8GB としたときの，各パターンの資源使用量をまとめたものである．まず，Delta Compression による転送データの圧縮によってネットワーク負荷が減少していることが確認できる．どちらのワークロードでも微量ではあるが，GPU へのオフロードによって転送効率の向上と CPU 使用量の減少がみられる．データ圧縮による影響が大きい LMbench では，CPU-DeltaC とパイプライン化を行わない GM-DeltaC の CPU 総使用量には 18%ほどの差が生じている．一方で，パイプライン化を有効にした場合には，転送効率の向上が更に大きくなっているが，CPU 資源の使用量の面ではオーバヘッドが大きくなっていて，LMbench の場合には，CPU 総使用量が CPU-DeltaC よりも大きくなる結果が生じている．この原因としては GPU のオフロードの効果が十分に大きくないことも考えられるが，LMbench のワークロード特性も関係していると考えられる．LMbench では一定の領域内でメモリ書き換えを非常に速い速度で繰り返す処理を行う．したがって，パイプライン化によって移送処理の効率化が生じても，メモリ書き換えの速度が非常に速いために，再送されるページ数に減少が生じていない可能性がある．その結果冗長なページ転送を高速に実行してしまい，CPU 資源を結果的に多く消費する結果となっているのではと考えられる．

表 5 は GM-DeltaC における CPU スリープによる効果を示すものである．どちらのワークロードにおいてもスレッドスリープの効果が発揮されており，ページ再送量が多く，Delta Compression が有効となっている時間が長い LMbench の場合には平均 CPU 使用率，総使用量を 10%近く低減している．また，移送時間にも大きな差はなく，冗長なスリープが少ないことも確認できる．

5.3 GM-Dedup

5.3.1 Deduplication

Deduplication とは，あるデータの中で重複した部分が存在する場合に，そのうちの一つのみを残して，他の重複部分をその残したデータへのポインタに置き換えることでデータサイズの縮小を図る手法である．この圧縮手法は

表 4　各パターンの資源使用量 (GM-DeltaC)

	LMbench			Memcached		
	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]
Default	56.89	105.34	26765	56.11	104.55	21073
CPU-DeltaC	35.30	106.69	25208	48.54	105.69	15638
GM-DeltaC w/o pipe	36.71	92.59	21657	49.54	102.59	14850
GM-DeltaC w/- pipe	49.79	136.68	25719	54.11	115.59	15128

表 5　スレッドスリープによる効果 (GM-DeltaC)

	LMbench			Memcached		
	移送時間 [s]	平均 CPU 使用率 [%/s]	CPU 総使用量 [%-sec]	移送時間 [s]	平均 CPU 使用率 [%/s]	CPU 総使用量 [%-sec]
GM-RLE w/o sleep	189.38	152.79	28937	127.02	119.78	15215
GM-RLE w/- sleep	188.16	136.68	25719	130.87	115.59	15128

VM Migration を利用した先行研究において，転送データの圧縮アルゴリズムとして広く用いられている [6, 8, 25]．GM-Dedup ではこの Deduplication によるデータ圧縮を GPU を活用して実行する．GM-Dedup では固定長チャンキング方式と呼ばれる Deduplication 手法を利用する．この方式では圧縮対象データを固定長のチャンクに区切って取扱い，過去に登場したチャンクは逐次キャッシュに保存されていく．模試圧縮対象のデータ中にキャッシュに保存されたチャンクと内容が等しいチャンクが表れた場合は，そのチャンクの部分のデータを該当キャッシュのインデックスを示すデータに置き換える処理を行う．データの伸長側にも同様のキャッシュが用意されていて，データを伸張していくごとにキャッシュが更新されていき，圧縮側と同様のキャッシュが常に維持されるため，キャッシュとそのインデックスを示すポインタの情報から常に正しいデータを復元可能である．

5.3.2 Implementation

GM-Dedup では固定長チャンキングと 2-way セットアソシエイティブ形式のキャッシュを用いて Deduplication の実装を行っている．1 チャンクのサイズは 256byte に設定されており，1 チャンクの処理に対して 1 つの GPU スレッドが対応し，重複の有無の確認やデータの置き換え処理を行う．キャッシュ関係については GM-DeltaC の場合と同様の設計をとっており，512MB のキャッシュが GPU 上に確保され，2-way セットアソシエイティブ形式で管理される．また，キャッシュ管理用のメタデータを取り扱うテーブルも GPU 上に用意されてる．キャッシュの管理テーブルや更新処理の制御も GM-DeltaC の場合と同様の構造がとられているため，キャッシュ管理の粒度が 1 ページ単位 (4KB) から 1 チャンク単位 (256byte) へ変化したものといえる．

重複の有無の確認は，ハッシュ値による比較とチャンク内データの単純比較を組み合わせて行う．チャンク内のデータの単純比較を行う前に，チャンク内のデータから生成したハッシュ値を利用したデータ内容の比較を簡易的に行う．このハッシュ値が GM-DeltaC の場合でのページ番号にあたるものであり，2-way セットアソシエイティブ形式のキャッシュにおけるデータ格納位置のインデックスの決定に使用される．したがって，GM-DeltaC において各ページのデータが自身のページ番号と紐付けられて保存されていたように，各チャンクは自身のハッシュ値と紐付けられてキャッシュ上に保存されている．ハッシュ値の生成には Super Fast Hash Algorithm [26] と呼ばれるハッシュ値生成アルゴリズムを使用する．このアルゴリズムはオープンソースの軽負荷・高速なハッシュ値生成アルゴリズムである．また，ハッシュ値生成の過程で条件分岐がほとんど発生しないため，条件分岐命令の実行が不向きな GPU 上でも効率的に処理可能であると考えられる．同値のハッシュ値をもつキャッシュデータが存在する場合には，そのデータとキャッシュは同様のデータ内容である可能性が高いため，そこで初めてチャンクとキャッシュ間で全データの単純比較を実行する．単純比較の結果チャンクとキャッシュのデータが等しいことが確認された場合には，チャンクのデータをキャッシュのインデックス番号に置き換える．同じハッシュ値を持つキャッシュが存在しない場合，またはチャンクのデータとキャッシュのデータが等しくなかった場合にはチャンクデータをそのまま処理適用後のデータとして用いる．

キャッシュに関するメタデータの管理を行うテーブルの取り扱いやキャッシュ更新の衝突の制御は，GM-DeltaC のものと同様の構造である．したがって，キャッシュのヒット状況やデータの参照位置の特定や，キャッシュ更新処理の制御は圧縮対象データに付与されたヘッダデータと管理テーブルの情報を元に行われる．また，GM-Dedup による圧縮処理全体も GM-DeltaC の場合と同様に 3 段階に分けて実行される．したがって，1 段階目にハッシュ値の生成とそれを利用したチャンクとキャッシュデータの重複の判定，2 段階目に重複の判定に応じたチャンクデータの操作，

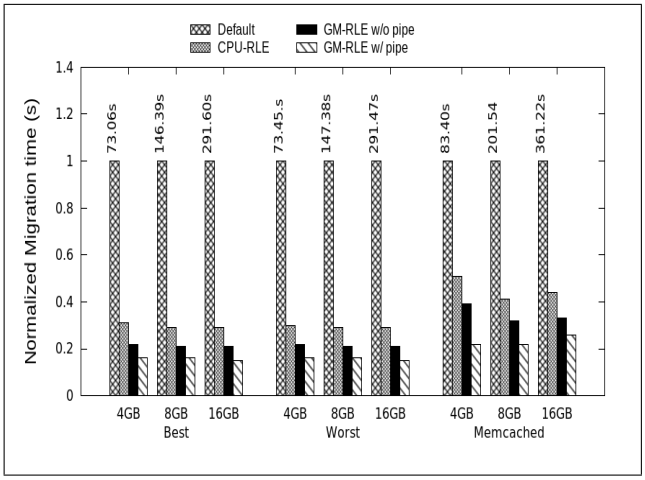

図 5　各パターンの移送時間 (GM-Dedup)

3 段階目でキャッシュの更新を実行する，といった流れとなる．なお，キャッシュの更新ポリシーとして LRU 方式を採用している．したがって，ハッシュ値の衝突が起こらなかったチャンクをキャッシュに追加する場合には，該当するインデックスに保存されたキャッシュのうち，最も使用されていないものを上書きする形でキャッシュ更新が実行される．

5.3.3 Experiments

GM-Dedup については GM-RLE と同様のワークロード (Best, Worst, Memcached) を用いて評価を行った．測定項目についても同様で, 移送時間, ネットワーク使用量, CPU 使用量について, VM のメモリ量を 4GB, 8GB, 16GB と変えながら測定を行う．比較を行う移送方式についても，Xen のデフォルトの Live VM Migration(Default)，CPU 上で Deduplication を行う Live VM Migration(CPU-Dedup)，パイプライン化を無効にした GM-Dedup(GM-Dedup w/o pipe)，パイプライン化を有効にした GM-Dedup(GM-Dedup w/-pipe) の 4 パターンで比較を行う．

図 **5** は各パターンの移送時間をまとめたものである．Best ワークロードと Worst ワークロードではほぼ同じ結果となっていて，CPU による Deduplication によって約 70%移送時間が短縮され，そこから GPU へのオフロードによってさらに 10%，パイプライン化によって 3%ほどの短縮が生じている．GM-RLE と比較した場合に，Worst の場合でも十分な効果が発揮されていることから，Deduplication によるデータ圧縮が RLE 法に比べてより広い場面で有効であることがうかがえる．一方で，オフロードによる移送時間の削減効果が GM-RLE に比べて小さくなっているのは，キャッシュの更新衝突の調整処理等を CPU 上でも行うため，相対的に GPU の処理の占める割合が小さくなっていることがまず原因として考えられる．また，Deduplication では圧縮・伸張処理，特に伸張処理においてキャッシュからのメモリコピーが主な処理となるため，GPU の高い演算性能を十分に活かしきれていない可能性が考えられる．しかしながら，オフロードによる移送の高速化は確かに実現されており，Memcached を用いたワークロードの場合にも，Best の場合と同等またはそれ以上の効果が発揮されている．同様にキャッシュを用いる処理を行う Delta Compression の時と比較して効果が表れているように見えるのは，ハッシュ値生成などのアクセラレーションの効果が期待できる処理が相対的に多いためであると考えられる．また，GM-RLE のときと同様にメモリ量が変化しても結果に差が生まれていないことから，GM-Dedup もメモリ量に対してスケールすることが確認できる．

表 **6** は GM-RLE の項で示したものと同様の各パターンの資源使用量をまとめたものである．移送時間に関して見られた傾向と同様に，GM-RLE と比べて GPU へのオフロードによる効果は小さいものであるが，全てのワークロードに対して GPU へのオフロード，パイプライン化による資源使用量の削減効果が表れている．

表 **7** は，GM-Dedup におけるスレッドスリープを有効にした場合と無効にした場合の資源使用量の差をまとめたものである．GM-Dedup では，圧縮処理中に占める CPU の処理の割合が GM-RLE に比べて大きいため，スレッドスリープによる恩恵がやや小さな結果となっている．しかしながら，平均 CPU 使用率 20%ほど減少していることからも，スレッドスリープによる CPU 資源使用量の削減は有効であることがうかがえる．

6. Related Work

GMigrate ではデータ圧縮型 Live VM Migration に対して，RLE 法，Delta Compression [7]，Deduplication の 3 種の圧縮方式を対象に，GPGPU を活用したアクセラレーションを行った．これらの圧縮方式は一般的なアルゴリズムを用いたものであり，細かな実装の違いこそあれど，先行研究 [3,8,25] における VM Migration で使用されている圧縮アルゴリズムと非常に近いものである．したがって，本手法をもとにした GPU アクセラレーションが，従来の多くのデータ圧縮型 VM Migration にも適用できる可能性は十分に考えられる．

PMigrate [19] では複数の CPU や NIC を効果的に使用するため，移送処理の高度なパイプライン化を行っている．パイプライン処理による高速化の効果を十分に高めるため，ゲストカーネルのメモリアクセス機構にも改変を加えることで，VM のページ抽出処理の並列処理化までもを行っている．GMigrate でも移送のパイプライン処理を利用した効率化を行っているが，その目的は異なり，CPU と GPU の並列実行の実現を第一に考えた実装となっている．

また，移送の高速化を目的とした VM 移送手法は，データ圧縮型 VM Migration の他にもいくつか存在する．JAVMM [27] では Java VM と協調することによって，VM のメモリ上に存在する garbage オブジェクトを識別し，転

表 6　各パターンの資源使用量 (GM-Dedup)

	Best			Worst			Memcached		
	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	平均ネットワーク転送量 [MB/s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]
Default	57.76	102.73	15040	57.38	101.79	15002	56.11	104.55	21073
CPU-RLE	11.29	110.79	4793	11.29	110.66	4785	24.86	106.70	9141
GM-RLE w/o pipe	15.24	100.91	3199	15.26	101.44	3219	28.43	101.91	6756
GM-RLE w/- pipe	20.00	128.90	3073	20.02	129.75	3079	40.64	145.43	6456

表 7　スレッドスリープによる効果 (GM-Dedup)

	Best			Worst			Memcached		
	移送時間 [s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	移送時間 [s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]	移送時間 [s]	平均 CPU 使用率 [%]	CPU 総使用量 [%-sec]
GM-RLE w/o sleep	22.44	148.70	3337	22.45	148.90	3343	48.88	155.38	7595
GM-RLE w/- sleep	23.84	128.90	3073	23.73	129.75	3079	44.39	145.43	6456

送データから除外することで転送データ量の削減を行っている．SonicMigration [28] では，バッファキャッシュや非使用ページといった，VM の稼働に必ずしも必要でないカーネルオブジェクトを OS が管理するメモリ情報をもとに識別し，それらを転送データから取り除くことでデータ転送量を削減している．これらの手法は，データ圧縮型 VM Migration とは異なるアプローチで転送データ量の削減を行い，移送の高速化を実現している．目的は同じであるがデータ削減処理が作用するタイミングが圧縮型 VM Migration とは異なるため，GMigrate とも基本的には共存可能なものであると考えられる．

これまで述べた先行研究の一部には，VM の OS やアプリケーションレイヤに制限や改変が必要となるものも存在する [3,19,27]．これに対して，データ圧縮型 VM Migration は基本的に VMM より上のレイヤへ依存せずに効果を発揮可能である．したがって，手法を適用できる場面が多いといった点も，データ圧縮型 VM Migration の重要な利点の一つであると考えられる．

昨今では GPGPU の利用対象の研究は盛んに行われており，大規模な科学演算やバックアップシステム [15] における GPU の活用だけでなく，ネットワーク処理の高速化 [16,17] など，その試みは広範囲にわたる．GMigrate もその流れの一つに含まれ，Live VM Migration への GPU アクセラレーションの適用の試みを行っている．

7. Conclusion

本論文ではデータ圧縮型 VM Migration に対して GPGPU によるアクセラレーションを適用した，GMigrate の提案を行った．GPU によるアクセラレーションはデータ圧縮型 VM Migration におけるデータの圧縮・伸張処理部分を対象に行い，その実装例として RLE 法，Delta Compression，Deduplication の 3 つの圧縮手法に関して実装を行った．また，より効果的な GPU アクセラレーションを実現するため，GPU と CPU の並列実行を可能にする移送処理のパイプライン化や，冗長な CPU-GPU 間の同期処理を減らすスレッドスリープ機能の実装を行った．そして評価実験を行った結果，GPU へのオフロードによって最大 30%以上の移送時間と CPU 使用量の削減が確認された．

本手法の実装はまだ進行中のものである．Case Study にて示した各実装パターンの評価結果からも読み取れるように，GPU アクセラレーションによる効果が十分でない場合が存在している．特に GM-DeltaC や GM-Dedup におけるキャッシュ関連の処理ではオフロードの恩恵がまだまだ弱く，CPU と GPU の行う処理の構成やスレッドスリープの効果を考慮した効率化をさらに行っていく必要がある．また，CPU-GPU 間でのデータコピーのオーバヘッドへの対処が足りず，VM のダウンタイムに影響を及ぼす可能性も生じているため，CPU-GPU 間のデータコピーに関する効率化も行う必要がある．また，実環境下を想定したワークロードを用いた評価実験も十分とは言えない．例えば，SQL サーバなどのワークロードを与えた場合についても評価を行い，本手法の更なる課題の発見と克服を行っていく必要がある．

参考文献

[1] Clark, C., Fraser, K., Hand, S., Hanseny, J. G., July, E., Limpach, C., Pratt, I. and Wareld, A.: Live Migration of Virtual Machines, *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pp. 273–286 (2005).

[2] Amazon: Amazon EC2 (Amazon Elastic Compute Cloud), http://aws.amazon.com/jp/ec2/.

[3] Jui-Hao Chiang, Han-Lin Li, T.-c. C.: Introspection-based Memory De-duplication and Migration, *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*, pp. 51–62 (2013).

[4] Rai, A., Ramjee, R., Anand, A., Padmanabhan, V. N. and Varghese, G.: MiG: Efficient Migration of Desktop VMs Using Semantic Compression, *Proceedings of the 2013 USENIX Conference on Annual Technical Con-*

ference (USENIX ATC '13), pp. 25–36 (2013).

[5] Jin, H., Wu, S. and Pan, X. S. X.: Live Virtual Machine Migration with Adaptive Memory Compression, *IEEE International Conference on Cluster Computing (CLUSTER '09)*, pp. 1–10 (2009).

[6] Wood, T., Ramakrishnan, K. K., Shenoy, P. and van der Merwe, J.: CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines, *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '11)*, pp. 121–132 (2011).

[7] Svard, P., Hudzia, B., Tordsson, J. and Elmroth, E.: Evaluation of delta compression techniques for efficient live migration of large virtual machines, *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11)*, pp. 111–120 (2011).

[8] Zhang, X., Huo, Z., Ma, J. and Meng, D.: Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration, *IEEE International Conference on Cluster Computing (CLUSTER '10)*, pp. 88–96 (2010).

[9] KVM: KVM, http://www.linux-kvm.org/page/Main_Page.

[10] Koto, A., Yamada, H., Ohmura, K. and Kono, K.: Towards Unobtrusive VM Live Migration for Cloud Computing Platforms, *Proceedings of the Third ACM SIGOPS Asia-Pacific Conference on Systems (APSys '12)*, pp. 7–7 (2012).

[11] Hines, M. R. and Gopalan, K.: Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning, *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*, pp. 51–60 (2009).

[12] Lu, P., Barbalace, A. and Ravindran, B.: HSG-LM: Hybrid-copy Speculative Guest OS Live Migration Without Hypervisor, *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*, pp. 2:1–2:11 (2013).

[13] NVIDIA: CUDA - NVIDIA Developer Zone, https://developer.nvidia.com/category/zone/cudazone.

[14] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. and Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication, *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, pp. 161–174 (2008).

[15] Pramod Bhatotia, Rodrigo Rodrigues, A. V.: Shredder: GPU-Accelerated Incremental Storage and Computation, *0th USENIX Conference on File and Storage Technologies (FAST '12)*, p. 14 (2012).

[16] Vasiliadis, G., Koromilas, L., Polychronakis, M. and Ioannidis, S.: GASPP: A GPU-Accelerated Stateful Packet Processing Framework, *2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pp. 321–332 (2014).

[17] Han, S., Jang, K., Park, K. and Moon, S.: PacketShader: Massively Parallel Packet Processing with GPUs to Accelerate Software Routers, *ACM SIGCOMMN 2010 Conference (SIGCOMM '10)*, pp. 321–332 (2010).

[18] Amazon: Announcing New Amazon EC2 (GPU Instance Type), http://aws.amazon.com/jp/about-aws/whatsnew/2013/11/04/announcingnewamazonec2-gpuinstancetype/.

[19] Song, X., Shi, J., Liu, R., Yang, J. and Chen, H.: Parallelizing Live Migration of Virtual Machines, *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '13)*, pp. 85–96 (2013).

[20] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and Art of Virtualization, *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 164–177 (2003).

[21] Kato, S., McThrow, M., Maltzahn, C. and Brandt, S.: Gdev: First-class GPU Resource Management in the Operating System, *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12)*, pp. 37–37 (2012).

[22] PARSA-EPFL: Data Caching Benchmark, http://parsa.epfl.ch/cloudsuite/memcached.html.

[23] Hill, M. D.: Aspects of cache memory and instruction buffer performance, *PhD thesis, University of California, Barkeley* (1987).

[24] McVoy, L.: LMbench - Tools for Performance Analysis, http://www.bitmover.com/lmbench/.

[25] Singh, R., Irwin, D., Shenoy, P. and Ramakrishnan, K.: Yank: Enabling Green Data Centers to Pull the Plug, *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pp. 143–155 (2013).

[26] Hsieh, P.: Hash functions., http://www.azillionmonkeys.com/qed/hash.html.

[27] Hou, K.-Y., Shin, K. G. and Sung, J.-L.: Application-assisted Live Migration of Virtual Machines with Java Applications, *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, pp. 15:1–15:15 (2015).

[28] Koto, A., Yamada, H., Ohmura, K. and Kono, K.: Towards Unobtrusive VM Live Migration for Cloud Computing Platforms, *Proceedings of the Asia-Pacific Workshop on Systems (APSys '12)*, pp. 7:1–7:6 (2012).