Vol.2009-HPC-122 No.7
2009/10/9

CUDA アプリケーション向け チェックポイント・リスタート機能の実装と評価

滝沢寛之[†1]　佐藤功人[†1]
小松一彦[†2]　小林広明[†2]

本論文では，CUDA アプリケーションのチェックポイント・リスタートを実現するためのツールとして CheCUDA を提案する．既存のチェックポイント・リスタートシステムを使って CUDA アプリケーションのチェックポイント・リスタートを実現するため，CheCUDA は CUDA の API 呼び出し時に GPU の状態変化をメモリに記録するためのアドオンパッケージとして設計されている．本論文では，CheCUDA を試作し，実際に CUDA アプリケーションのチェックポイント・リスタートを正常に実現できることを明らかにする．また，チェックポイントファイルを生成した PC とは環境の異なる他の PC 上でリスタートできることも確認し，CheCUDA がディペンダビリティの向上だけでなくタスクマイグレーションにも有用であることを示す．さらに，CheCUDA のチェックポイント処理のオーバヘッドを定量的に評価する．

Implementation and Evaluation of a Checkpint/Restart Tool for CUDA Applications

Hiroyuki Takizawa ,[†1] Katsuto Sato ,[†1]
Kazuhiko Komatsu [†2] and Hiroaki Kobayashi[†2]

In this paper, a tool named CheCUDA is designed to enable checkpoint/restart of CUDA applications. To allow an existing checkpoint/restart implementation to checkpoint CUDA applications, CheCUDA is developed as an add-on package working at each CUDA API call to record the GPU status changes onto the main memory. This paper demonstrates that our prototype implementation of CheCUDA can correctly checkpoint and restart some CUDA applications. It is also shown that CheCUDA can restart a CUDA process from a checkpoint file generated on another PC. Accordingly, CheCUDA is useful not only to enhance the dependability of CUDA applications but also to attain task migration of CUDA applications. This paper also shows the timing overhead for checkpointing.

1. はじめに

近年の描画処理ユニット (Graphics Processing Unit, GPU) の高い演算能力を描画処理以外の計算にも利用する試み (General Purpose computation on GPUs, GPGPU) が盛んに行われており[1)]，GPU をアクセラレータとして効果的に利用することによって，多種多様なアプリケーションに対する実効性能が飛躍的に向上することが報告されている．しかし，GPU は近年急速に発展した計算プラットフォームであり，その利用環境が完全には整備されていないため，従来の HPC 分野で広く用いられてきたいくつかの重要な機能が GPGPU では利用できない状況にある．そのような重要な機能の一つとして，チェックポイントリスタート (Checkpoint/Restart, CPR) が挙げられる．

これまでに様々な CPR ツールが開発されてきたが[2)–5)]，GPU の状態を正常にチェックポイントファイルに保持することが可能なツールは未だ開発されていない．このため，GPU の状態も含めてチェックポイントファイルに保持し，リスタート時にそのファイルから正しくプロセスを再開できる CPR ツールを開発することは，GPGPU システムのディペンダビリティを高める上で非常に重要である．一般向けの PC に高性能 GPU を搭載して気軽に試すことができるという点が GPGPU の利点の一つであるが，そのような場合には熱や消費電力などの問題でシステムが一時的に不安定になる可能性があり，ディペンダビリティを高める仕組みが特に重要となる．また，例えば各計算ノードが性能の異なる GPU を搭載する不均質なクラスタシステムにおいては，状況に応じてプロセスを他の計算ノードに移動した方がシステム全体としての性能やエネルギ効率を高めることができる可能性がある．そのような動的タスクスケジューリングなどの実現を考える際に，GPU をサポートする CPR は有用である．

本論文では，NVIDIA 社の GPGPU アプリケーション開発フレームワークである CUDA を対象とし，CUDA アプリケーションの CPR 機能を提供する *CheCUDA* を設計・実装する．CUDA アプリケーションがチェックポイントファイルから正常にリスタートできることを確認するとともに，いくつかの CUDA アプリケーションを用いて実行時オーバヘッド

†1 東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University
†2 東北大学サイバーサイエンスセンター
Cyberscience Center, Tohoku University

やチェックポイント処理のオーバヘッドを定量的に評価する．

2. 関 連 研 究

2.1 チェックポイント・リスタート

チェックポインティングとは，実行中のプロセスの状態をファイルに書き出すことである．また，そのファイルからプロセスの状態を復元して再開する処理を**リスタート**と呼ぶ．これらの機能を実現する CPR の研究は古典的ではあるが，依然としてディペンダブルな高性能計算を実現するために極めて重要な研究課題である．これまでにも，効果的な CPR を実現するための研究が非常に多く報告されている[5]．

Berkeley Lab Checkpoint/Restart(BLCR) パッケージ[2] は，活発に開発されている Linux 向け CPR システムである．マルチスレッドアプリケーションや分散並列アプリケーションを含む多様なアプリケーションに対するカーネルレベル CPR をサポートしている．他の多くの CPR システムと同様に，BLCR ではチェックポインティングの前後やリスタートの直後にユーザ定義のコールバック関数を呼び出すことができる．このコールバック関数を適切に定義することにより，BLCR が本来はサポートしていないデバイスに対するチェックポインティングやリスタートの処理を実行することができる． また，いくつかの制約があるものの，BLCR はタスクマイグレーションもサポートしており，ある計算ノードで生成したチェックポイントファイルに基づいて他の計算ノードでプロセスを再開することも可能である．本論文では，BLCR のこれらの機能を利用することによって CUDA アプリケーションの CPR やタスクマイグレーションの実現することを検討する．

実行中のプロセスの状態を保持・再開する機能を実現するために，Xen[6] などに代表される仮想マシンを使うという選択肢も考えられる．Shi らは仮想マシンに基づいて CUDA アプリケーションの状態の保存と再開を実現する vCUDA を提案している[7]．しかし，vCUDA では全ての CUDA 命令を傍受してホスト OS に転送しているため，その実行時オーバヘッドによって CUDA アプリケーションの実効性能が低下することが報告されている．仮想マシンにょるアプローチでは CUDA アプリケーションの性質に依存して実効性能が大きく低下する可能性があるため，より高速な状態の保存と再開を期待できる CPR によるアプローチを本研究では採用する．

2.2 CUDA による GPGPU

2006 年に発表された NVIDIA 社の *CUDA* (Compute Unified Device Architecture) は，現在最も一般的な GPGPU 向けアプリケーション開発環境である[8]．CUDA では，標準的な C/C++言語にいくつかの予約語と文法を追加することによって，トリッキーなプログラミング技法を用いることなく，GPU を描画以外の汎用計算に利用することが可能になっている．

```
CUdeviceptr ptr;

cuMemAlloc(&ptr, sizeof(float)*1024);
```

図 1 デバイスメモリ上の領域確保
Fig. 1 Device memory allocation.

CUDA では，CUDA Runtime API と CUDA Driver API という 2 種類の API が提供されている．後者は前者よりも低レベルな API である．本論文で提案するアプローチは前者にも適用可能であると考えられるが，GPGPU アプリケーションのより低レベルな挙動を解析するために，本論文では CUDA Driver API で記述された GPGPU アプリケーションを対象とする．

CUDA では，GPU は計算デバイス (compute device) として扱われ，デバイスメモリ (device memory) を固有のメモリとして持っている．一方，汎用プロセッサ (CPU) は計算デバイスを制御するホスト (host) として扱われる．

CUDA Driver API では，コンテクスト (context) やモジュール (module)，デバイスメモリ上に確保されたメモリ領域やテクスチャなど，いくつかの CUDA のリソースをプログラマが明示的に管理する．CUDA のコンテクストは CPU におけるプロセスに相当し，モジュールはダイナミックリンクライブラリに相当する[9]．API を介して CUDA リソースを確保すると，そのリソースに対応する識別子であるハンドル (handle) が得られる．プログラマはそのハンドルを介して各種リソースにアクセスし，デバイスを制御する．CUDA リソースを確保する例として，デバイスメモリ上にメモリ領域を確保するコードを**図 1** に示す．通常の `malloc` によるヒープ領域の確保と同様に，図 1 ではコード中で指示された `sizeof(float)*1024` バイトのデバイスメモリ領域を確保し，その先頭アドレスをハンドルとして変数 `ptr` に受け取っている．

既存の CPR システムを用いる場合，CPU 側のホストメモリの状態を復元することが可能であるが，GPU 側にある各種 CUDA リソースの状態を復元することができない．したがって，既存の CPR システムによって上記の変数 `ptr` の値 (アドレス) は復元されるが，そのアドレスを先頭としたデバイスメモリ領域は確保されておらず，その領域に記憶されてい

たデータも復元されない．このため，既存の CPR システムでは CUDA アプリケーションを正しくリスタートすることができない．

3. CUDA アプリケーション向けチェックポイントリスタート

CUDA アプリケーションの CPR を実現するためのツールとして，本論文では CheCUDA を提案する．一般的に CPR システムを開発し維持・管理するためには多大な人的労力が必要である．また，現在でも CUDA の開発が非常に活発に行われており，そのバージョンは頻繁に更新されるため，CPR 本体と CUDA をサポートするための部分を明確に区別して管理する方が望ましい．このため，CheCUDA は既存の CPR システムである BLCR に対するアドオンパッケージとして設計され，BLCR 自体に修正を加えることなく動作することを目指している．BLCR はホスト側のプロセスの保存と復元だけを担当し，CUDA リソースの保存と復元を CheCUDA が行う．

BLCR で CUDA アプリケーションのチェックポインティングを行う場合，チェックポインティング時にプロセスのメモリマップ上に存在していた GPU のキャラクタデバイス (`/dev/nvidia0` など) をリスタート時に復元するために `mmap` を試みて失敗する．様々な条件下で試した結果，CUDA コンテクストが存在している状態で生成されたチェックポイントファイルから，プロセスが正常にリスタートすることはできなかった．一方，CUDA コンテクストが存在しない場合には，それに付随する他の CUDA リソースも存在しないため，既存の CPR システムでプロセスの CPR を実現することができる．したがって，BLCR のように CUDA をサポートしない CPR システムを用いて CUDA アプリケーションの CPR を実現するための最も確実な方法は，チェックポインティングの直前に一旦全ての CUDA リソースを解放し，チェックポインティング後あるいはリスタート後に全ての CUDA リソースを復元することである．その場合，チェックポインティング時には CUDA コンテクストが存在しないため，BLCR でもそのプロセスを正常にリスタートすることが可能である．CheCUDA がチェックポインティングの前後に行う処理を以下にまとめる．

(1) デバイスメモリ上にあるすべてのユーザデータをホストメモリ上に複製する．
(2) CUDA コンテクストを破棄し，全ての CUDA リソースを解放する．
(3) ホスト上のプロセスの状態をファイルに書き出す．このとき，手順 (1) で複製されたデータもファイルに保存する．
(4) GPU を初期化し，手順 (2) 以前に存在していた CUDA リソースを復元する．
(5) 手順 (1) でホストメモリ上に複製されたユーザデータをデバイスメモリ上に複製する．

リスタート時には手順 (4) から処理が再開される．手順 (4) で CUDA リソースを復元するためには，その復元時に必要な情報を手順 (3) で全て記録しておく必要がある．例えば，`cuMemAlloc` で確保されたデバイスメモリ上の領域 (図 1 参照) を復元するためには，以前と同じサイズの領域を復元するためにサイズを記録しておく必要がある．この記録を暗黙裏に行うため，CheCUDA では `cuMemAlloc` が呼び出された時に同名の別の関数 (**ラッパ関数**) が一旦呼び出される．デバイスメモリ上のアドレスを管理するための変数型である `CUdeviceptr` も同名の別のクラス (**ラッパクラス**) となっており，ラッパ関数内で本来の `cuMemAlloc` が呼び出されてメモリ領域が確保された後に，その先頭アドレスとサイズがラッパクラスのオブジェクト内に保持される．同様に，他の多くの CUDA リソースに関してもリソース確保時の情報を記録しておくことで，チェックポインティング直前に解放しても後で復元することが可能である．

さらに，デバイスメモリ領域の場合には，その領域内に記憶されているデータも記録 (手順 (1)) しておき，デバイスメモリの再確保後に以前のデータを復元 (手順 (5)) する必要がある．このために，CheCUDA はアプリケーション内で確保されたデバイスメモリ領域に関する情報をリスト (以下，**リソース管理リスト**とする) で管理している．`cuMemAlloc` のラッパ関数内で確保されたメモリ領域の情報がリソース管理リストに登録され，領域を開放する `cuMemFree` のラッパ関数内で該当するエントリをリストから削除している．手順 (1) では，リソース管理リストに登録されている全てのデバイスメモリ領域に関して，領域内のデータをホストメモリに複製して保持する．手順 (5) では，リソース管理リストに基づいて，ホストメモリ上に保持されたデータをデバイスメモリ上に複製する．これらのラッパ関数およびラッパクラスの機能を用いることにより，デバイスメモリ領域を一旦解放しても後で再度復元することが可能となっている．CheCUDA は各種 CUDA Driver API のラッパ関数やハンドル群のラッパクラスを提供し，CUDA リソースの復元に必要な情報を暗黙裏に管理することにより，既存の CUDA コードに大幅な修正を加えることなく，CPR を実現することができる．

CheCUDA で効率的な CPR を実現するためは，いつ，どれくらいの頻度でチェックポインティングを実行すべきかも考慮する必要がある．CheCUDA ではデバイスメモリ上のデータもホストメモリ側に複製するなど，チェックポインティングのオーバヘッドが通常の CPR と比較して大きくなることが予想されるため，チェックポインティングのタイミングは重要である．また，一連の API 呼び出しをアトミックに行う必要がある場合，その途中でチェックポインティングを実行すると正常にリスタートできない可能性がある．CUDA

```
#include "CheCUDA.h"

int main(int argc, char** argv)
{
  CUdevice hDev;
  CUcontext hCtx;

  // CheCUDA initialization
  ckptInit();

  // CUDA intialization
  cuInit(0);
  cuDeviceGet(&hDev, 0);
  cuCtxCreate(&hCtx, 0, hDev);

  ... CUDA code ...

  // CheCUDA's checkpointing
  ckptSelf();

  ... CUDA code ...

  cuCtxDetach(hCtx);
  return 0;
}
```

図 2 簡単な CUDA コード中で CheCUDA を利用する例
Fig. 2 A simple CUDA code with CheCUDA.

アプリケーションの CPR の実行可能性を示すため，本論文では適切なチェックポインティングのタイミングをプログラマが指定することを仮定する．ソース中のチェックポインティングを実行すべき箇所で `ckptSelf` という関数を呼び出すことにより，CheCUDA によるチェックポインティングが実行される．

図 2 は，簡単な CUDA コードに CheCUDA の関数を記述した例である．通常の CUDA コードでは，`CUdevide` と `CUcontext` は，それぞれ CUDA デバイスと CUDA コンテクストのハンドルを保持するための変数型である．また，これらのハンドルはそれぞれ `cuDeviceGet` と `cuCtxCreate` という API を呼び出すことによって，実際の CUDA リソースに対応づけられる．CheCUDA では，これらの変数型や API が同名のラッパクラスとラッパ関数に置き換えられており，CUDA リソースの復元に必要な情報を管理している．また，ラッパ関数内で実際の API が呼ばれている．また，`ckptInit` と `ckptSelf` により，それぞれ CheCUDA 初期化処理とチェックポインティング処理の呼び出しを行っている．

表 1 評価システム諸元
Table 1 System Specifications

	Desktop1	Desktop2	Laptop
CPU	Core 2 Quad Q6600	Phenom II X4 940	Core 2 Duo T7250
GPU	GeForce GTX280	GeForce 8800 GTX	GeForce 8600M GT
Main Memory	4GB	2GB	2GB
Video Memory	1GB	768MB	256MB
HDD BW	49 MB/sec	46 MB/sec	33 MB/sec
PCIe BW(HtoD)	2.42 $\times 10^3$ MB/sec	2.60 $\times 10^3$ MB/sec	2.39 $\times 10^3$ MB/sec
PCIe BW(DtoH)	1.52 $\times 10^3$ MB/sec	2.36 $\times 10^3$ MB/sec	4.52 $\times 10^2$ MB/sec
OS	CentOS 5.3 (x86_64), kernel 2.6.18-128.1.14.el5		
CUDA	CUDA 2.2, driver 185.18		

4. 性能評価と考察

本節では，3 節で提案した CheCUDA の試作実装の性能を評価する．CUDA アプリケーションの CPR を実現できることを実証するために，現在の CheCUDA の実装では，CUDA Driver API の基本的な関数群をサポートするために必要なラッパ関数およびラッパクラスが用意されている．本論文では，ホストメモリのスナップショットを取得するためにバージョン 0.81 の BLCR を利用している．以下の実験では，**表 1** に示された 3 種類の PC を使って性能評価を行う．表 1 の HDD BW は，8GB のファイルをハードディスクに書き込むときの平均バンド幅を示している (計測には Bonnie++[10] を使用)．また，PCIe BW(HtoD) および PCIe(DtoH) はそれぞれホスト-デバイス間で 64MB のデータを送受信する場合の平均バンド幅を示している．

4.1 リソース管理リストのための実行時オーバヘッド

まず最初に，CheCUDA のラッパ関数およびラッパクラスの実行時オーバヘッドを評価する．ラッパ関数およびラッパクラスは，CUDA リソースを復元するために必要な情報をリソース管理リストで暗黙的に管理しているため，CUDA リソースの確保や解放の時にリソース管理リストの更新のための実行時オーバヘッドが発生する．実行時オーバヘッドの評価に使用したコードの一部を**図 3** に示す．図 3 はデバイスメモリの確保と解放を繰り返すだけのコードである．本コードをコンパイルして通常の CUDA とリンクした実行ファイルと，CheCUDA とリンクした実行ファイルとの間で実行時間を比較することで，リソース管理リストへのエントリの登録と削除に要する実行時オーバヘッドを評価することができる．

CUDA リソースの数が多い場合，リソース管理リストが長くなるために，その更新に要

```
CUdeviceptr d_A[nHandles];
unsigned int mem_size_A = 256 * sizeof(float);

for(unsigned int h=0;h<nHandles;h++){
  // allocate device memory
  cuMemAlloc( &d_A[h], mem_size_A );
}
for(unsigned int h=0;h<nHandles;h++){
  // deallocate device memory
  cuMemFree(d_A[h]);
}
```

図 3 実行時オーバヘッド評価に用いたコード
Fig. 3 The code used for runtime overhead evaluation.

する CheCUDA の実行時オーバヘッドは大きくなる．**図 4** は，図 3 中のリソース数を変化させてその性能を評価した結果である．この結果より，ラッパクラスとラッパ関数の使用によって，それぞれのリソース確保時に最大で数マイクロ秒程度だけ実行時間が増加することが分かる．リソース管理リストの管理は CPU 側で行われるため，CPU の性能が高い場合には実行時オーバヘッドが小さくなる．CUDA リソースの確保がコード中で頻繁に行われることは稀であり，仮にその確保と解放を 256 回も繰り返した場合でも実行時間の増加は 0.3 ミリ秒以下であることから，一般的な CUDA アプリケーションの総実行時間と比較すると極めて小さいと言える．したがって，実用上，リソース管理リストを維持するために必要な CheCUDA の実行時オーバヘッドが性能に与える影響は，無視できるほど小さいことが明らかになった．

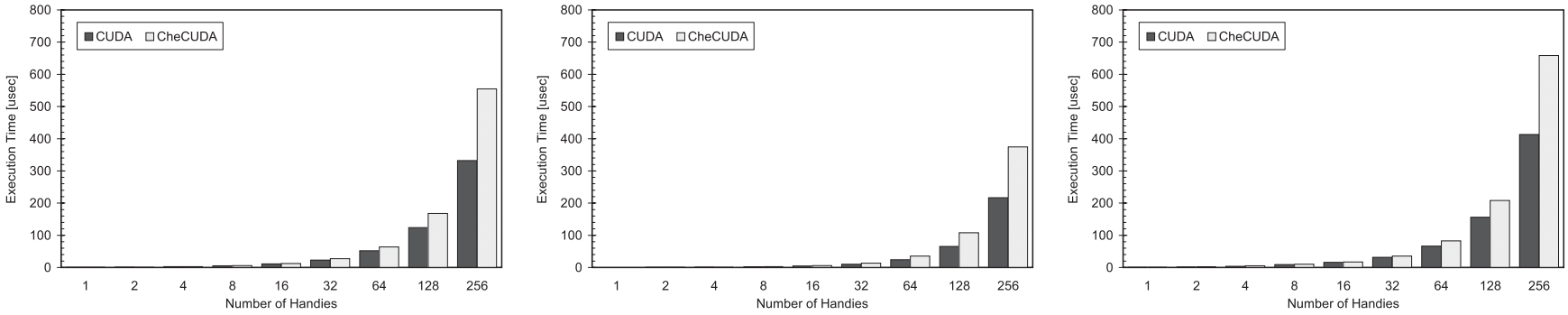
4.2 チェックポインティングのオーバヘッド

次に，NVIDIA CUDA SDK バージョン 2.2 のサンプルコードの一つである行列積計算コード `matrixMulDrv` を使って，CUDA アプリケーションにおけるチェックポインティングのオーバヘッドを評価する．図 2 と同様に，`ckptInit` と `ckptSelf` という 2 種類の関数が元の `matrixMulDrv` に挿入されており，行列積の計算が終わった直後に `ckptSelf` が呼び出される．その関数呼び出し時に内部的に `cuCtxSynchronize` が呼び出されるため，CPU と GPU が同期した後にチェックポイントファイルが生成される．本評価では，さらに `matrixMulDrv` の行列サイズを変更できるようにコードを修正して利用している．これによって，チェックポインティングの前処理，チェックポインティング，およびチェックポインティングの後処理のオーバヘッドと行列サイズとの関係を調べることができる．前処理ではデバイスメモリ上のユーザデータをホストメモリに複製し，チェックポインティングではホストメモリのイメージをファイルに書き出し，後処理では以前デバイスメモリにあったデータをすべて復元する．このため，それらの実行時間は行列サイズに依存している．

図 5 に，チェックポインティングに要する時間を評価した結果を示す．図中で，行列積の計算時間は others に含まれており，GPU の性能が低い Laptop PC では others が長くなっている．この結果より，行列サイズが小さい場合，行列積の計算時間が含まれている others よりも前処理と後処理の時間の方が長いことが分かる．それらの時間の大半は，CheCUDA による CUDA リソースの生成と削除に要する時間であり，CheCUDA が GPU の状態を保持するために要する付加的なオーバヘッドである．さらに，ホストメモリとデバイスメモリ間で行列データの複製を行うため，前処理と後処理の時間は行列サイズの増加とともに少しずつ増えている．しかし，行列サイズが大きい場合には，BLCR によるチェックポインティング自体の実行時間が CheCUDA による付加的なオーバヘッドよりも遙かに大きくなる．この主要因として，ハードディスクへの書き込みがホストメモリとデバイスメモリ間の転送よりも遅いことが挙げられる．このため，ホストメモリとデバイスメモリ間のデータ転送性能が低い Laptop PC(表 1 参照) でも，BLCR によるチェックポインティング自体が CheCUDA による前処理・後処理よりも長い実行時間を必要とする．これらの結果から，CheCUDA によって引き起こされる付加的なオーバヘッドは，BLCR によるチェックポインティングの時間と比較すれば実用上無視できるほど小さいことが示された．

4.3 タスクマイグレーション

最後に，CUDA アプリケーションの実行プロセスをある PC でチェックポインティングし，他の PC でリスタートするタスクマイグレーション機能を確認する．基本的に，BLCR はタスクマイグレーションをサポートしているため，本評価では CheCUDA によって生成されたチェックポイントファイルがホストに依存する情報を保持していないこと，およびその結果として他のホストでも正常にリスタートできることを確認する．本評価では，表 1 に示した 3 台の PC 間で `matrixMulDrv` のタスクマイグレーションを行い，正常にリスタートできることを確認した．したがって，CheCUDA によって CUDA アプリケーションのプロセスのマイグレーションを実現できることが明らかになった．しかしながら，本評価ではわずか 3 台の PC 間でのマイグレーションであり，CUDA や OS のバージョンも同一のものを使っているため，他の環境では CheCUDA による CUDA アプリケーションのタスクマイグレーションが失敗するかもしれない．また，CheCUDA は BLCR のアドオンである



(a) Desktop1　(b) Desktop2　(c) Laptop

図 4 リソース管理リストを維持するための実行時オーバヘッド
Fig. 4 Runtime Overhead for Maintaining a Resource Management List.

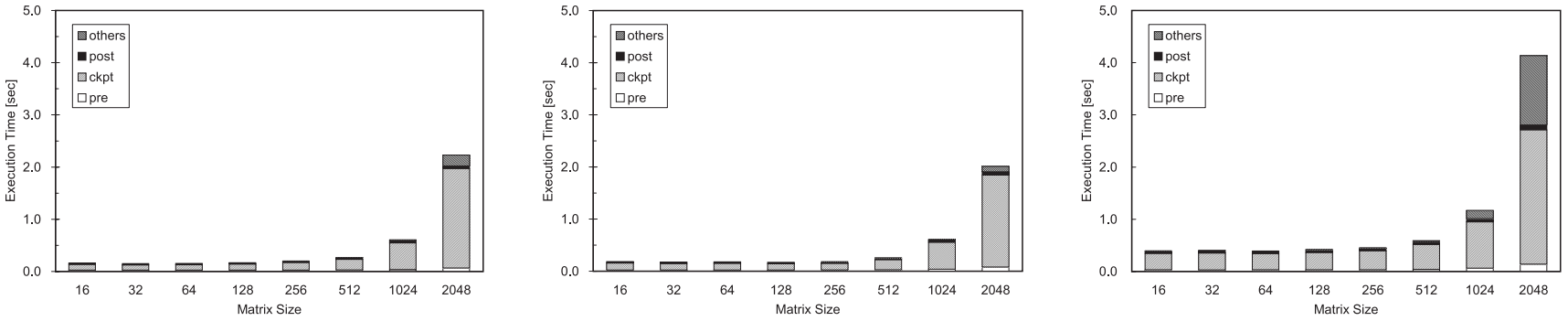
ため，BLCR がリスタートに失敗する条件下[⋆1]ではタスクマイグレーションを実現できない．このため，さらに様々なハードウェア構成，OS，共有ライブラリ，および CUDA アプリケーションを使って，CheCUDA が正常にタスクマイグレーションを実現するための必要要件を明らかにすることが，重要な今後の課題の一つである．

5. まとめと今後の課題

本論文では，CUDA アプリケーションの CPR を実現するためのツールとして CheCUDA を提案した．最も一般的な CPR 実装の一つである BLCR のアドオンパッケージとして，BLCR とは独立に開発できるように CheCUDA は設計・実装されている．CUDA リソースを使用している状態で生成したチェックポイントファイルではリスタートに失敗するため，CheCUDA はチェックポインティングの前処理ですべての CUDA リソースを削除し，後処理で復元するという回避策を採用している．このアプローチはチェックポインティングのオーバヘッドを明らかに増加させるが，BLCR によってチェックポイントファイルを生成するために要する時間と比較すると，CheCUDA による前処理と後処理の時間は無視できるほど小さいことが，性能評価により示された．実用的な環境下では，BLCR のチェックポインティングとわずかな付加的オーバヘッドだけで CUDA アプリケーションのディペンダビリティを向上させることが可能であり，CheCUDA は有用なツールであることが示された．さらに，CheCUDA はある PC 上で動作している CUDA アプリケーションの実行プロセスを他の PC へと移動することが可能であり，CUDA アプリケーションの動的タスクスケジューリングを実現するためにも有用なツールとして利用できる可能性が示された．各ノードに性能の異なる GPU を搭載する PC クラスタにおいて動的タスクスケジューリングを行うことによって，システム全体のスループットやエネルギ効率などを向上させることが可能となる．

現在の CheCUDA の実装では CUDA Driver API の一部しかサポートしていないため，すべての API をサポートするために開発を進めている．特に，非同期の API を使った CUDA アプリケーションでは，チェックポインティング時に CPU と GPU を強制的に同期させるオーバヘッドが性能に大きな影響を与える可能性があるため，そのような CUDA アプリケーションを対象としてオーバヘッドを評価することが今後の重要な課題である．他にも，

⋆1 自明な例としては，32 ビット Linux と 64 ビット Linux 間でのタスクマイグレーションなど．



(a) Desktop1　(b) Desktop2　(c) Laptop

図 5　チェックポインティングとその前処理，後処理の実行時間
Fig. 5　Execution Times of Preprocessing, Checkpointing, and Postprocessing.

CheCUDA をさらに多様な環境下で評価し，CPR に必要な要件を明らかにする予定である．また，元の CUDA コードに一切の修正を加えることなく，再コンパイルするだけでチェックポインティングを実現するために，性能に大きな影響を与えずにチェックポインティングを実行可能なタイミングに関する調査を進めている．さらに，CUDA リソースを削除せずに CPR を実現する可能性について引き続き調査を進めると共に，CUDA Driver API に加えて CUDA Runtime API をサポートすることも予定している．

謝辞　本研究の一部は，科研費若手研究 (B)(21700049) および中山隼雄科学技術文化財団の助成による．

参 考 文 献

1) Owens, J., Houston, M., Luebke, D., Green, S., Stone, J. and Phillips, J.: GPU computing, *Proceedings of the IEEE*, Vol.96, No.5, pp.879–899 (2008).
2) Hargrove, P.H. and Duell, J.C.: Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters, *Proceedings of SciDAC 2006* (2006).
3) Ansel, J., Arya, K. and Cooperman, G.: DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop, *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)* (2009).
4) Plank, J.S., Beck, M., Kingsley, G. and Li, K.: Libckpt: Transparent Checkpointing under Unix, *Proceedings of Usenix Winter 1995 Technical Conference*, pp.213–223 (1995).
5) Elnozahy, E. and Plank, J.: Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery, *IEEE Transactions on Dependable and Secure Computing*, Vol.1, No.2, pp.97–108 (2004).
6) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Prat, I. and Warfiel, A.: Xen and the art of virtualization, *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp.163–177 (2003).
7) Shi, L., Chen, H. and Sun, J.: vCUDA: GPU Accelerated High Performance Computing in Virtual Machines, *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp.1–11 (2009).
8) NVIDIA Corporation: CUDA Zone – The resource for CUDA developers. http://www.nvidia.com/object/cuda_home.html.
9) NVIDIA Corporation: NVIDIA CUDA Programming Guide, version 2.2.1. http://www.nvidia.com/object/cuda_develop.html.
10) Coker, R.: Bonnie++ project page. http://www.coker.com.au/bonnie++/.