GPGPUを用いたリポジトリマイニングの高速化手法 — プロセスメトリクスの算出への適用 —

永野　梨南[1,a)]　中村　央記[1,b)]　亀井　靖高[1,c)]　ブラム　アダムス[2,d)]　久住　憲嗣[1,e)]
鵜林　尚靖[1,f)]　福田　晃[1,g)]

概要：リポジトリマイニング分野は，版管理システムやバグ管理システム等のリポジトリに保管されているデータを統合・分析し，ソフトウェア開発者に有用な情報を提供する分野である．リポジトリのデータサイズは大きくなり続けていることから，リポジトリマイニングのスケールアップは本研究分野の主な課題の 1 つである．近年では，スーパーコンピュータやクラウドコンピューティングのような従来手法が用いられているが，スーパーコンピュータの導入は高価であり，クラウドコンピューティングの導入は設定や調整に手間がかかるという問題が存在する．本論文では，既存のビデオカードを用いることで導入可能な GPGPU(General-Purpose computing on Graphics Processing Units) を用いたリポジトリマイニングのスケールアップを提案する．GPGPU を用いることにより，Eclipse プロジェクトのバージョン履歴上における代表的なリポジトリマイニングのケーススタディにおいて，CPU のみを用いた手法に比べ 43.9 倍高速化することができた．

Using the GPGPU for Large-Scale Mining Software Repositories Studies — An Experience Report of Calculating Process Metrics —

Rina Nagano[1,a)]　Hiroki Nakamura[1,b)]　Yasutaka Kamei[1,c)]　Bram Adams[2,d)]
Kenji Hisazumi[1,e)]　Naoyasu Ubayashi[1,f)]　Akira Fukuda[1,g)]

Abstract: Scalable analysis is an important issue in the Mining Software Repositories (MSR) field, which aims to integrate and analyze data stored in large software repositories such as source control repositories and bug repositories. Recently, researchers have experimented with conventional techniques like a supercomputer or cloud computing, but these are either too expensive or too hard to configure. The goal of this paper is to improve the scaling of MSR analysis techniques by using general-purpose computing on graphics processing units (GPGPU) on off-the-shelf video cards. Through a representative MSR case study (i.e., measuring co-change factors) on version history from the Eclipse project, we find that the GPGPU approach is up to a factor of 43.9 faster than a CPU-only approach.

1. はじめに

現在，ソフトウェア開発ではそのプロジェクトで計測可能な多くの情報 (ソースコード，バージョン管理情報，開発者間でやり取りされたメール等) が版管理システムやバグ管理システムといったソフトウェアリポジトリに保管されている．それらリポジトリには，実際のソフトウェア開発履歴が蓄積されていることから，プロジェクト特有の有用な情報が豊富に含まれていると考えられている [1]．この考えから，ソフトウェアリポジトリに対するデータマイ

[1] 九州大学
Kyushu University
[2] モントリオール理工科大学
École Polytechnique de Montréal
[a)] rina@f.ait.kyushu-u.ac.jp
[b)] hiroki@f.ait.kyushu-u.ac.jp
[c)] kamei@ait.kyushu-u.ac.jp
[d)] bram.adams@polymtl.ca
[e)] nel@slrc.kyushu-u.ac.jp
[f)] ubayashi@ait.kyushu-u.ac.jp
[g)] fukuda@ait.kyushu-u.ac.jp

ニング（以降，リポジトリマイニング）は近年活発に行われており [2]，プロジェクトにおいて開発者が意思決定をする際のエビデンスを提供している．例えば文献 [3] では，規模の大きいファイルに対するバグ修正回数は品質に重大な影響を及ぼすバグを見つける良い指標であるため，頻繁にバグ修正が行われているファイルはより慎重にテストする必要があると報告されている．

リポジトリマイニング分野における課題の 1 つは，どのようにして大規模なデータを取り扱うかである [1][4]．それは，利用されるリポジトリの種類が増加しており，それらのデータサイズが大きくなり続けているためである．Mockus[5] は，オープンソースソフトウェア開発プロジェクトのうちオンライン上で参照可能な版管理システムを調査した結果、ソースコードだけでもテラバイトクラスの規模であったことを報告している．したがってこの分野の進展のためには，リポジトリマイニングをスケールアップするための新たなアプローチが求められる．

現在，リポジトリマイニングをスケールアップする方法として，主に次の 2 つの方法が使われている．1 つはスーパーコンピュータの利用，もう 1 つは Hadoop や Pig のようなクラウドベースの環境の利用である [4]．これらの方法はある状況ではその有用性が示されているものの，一方で欠点も存在する．どちらの方法もリポジトリマイニング向けの実装への変更が必要であり，さらにスーパーコンピュータの導入は高価である．Hadoop や Pig の導入にはスーパーコンピュータ程の費用はかからないが，設定や調整に手間がかかるという問題が存在する [6].

そこで本論文では，リポジトリマイニング分野に対する GPGPU（General-Purpose computing on Graphics Processing Units）の利用に着目する[*1]．GPU(Graphics Processing Unit) はリポジトリマイニング分野において求められる大規模な並列アーキテクチャをサポートしている．GPGPU を利用する利点は，環境構築が簡単であるということである．GPU を汎用計算機に取り付け，ドライバをインストールするだけですぐに利用することができる．プログラムの実現には一部，GPGPU 特有の実装が必要ではあるが，導入の容易さから計算生物学や暗号化等のドメインにおいて数多くのアプリケーションに利用されている [8][9]．しかしながら，我々の知る限り，現在までソフトウェア工学分野において GPGPU を用いた研究はほとんどなく，特にリポジトリマイニング分野において用いた事例はない．

本論文では，GPGPU を用いたリポジトリマイニングの有用性を評価するために，リポジトリマイニング研究の 1 つとして広く取り組まれているテーマの 1 つである，不具合モジュール予測 [14][17][18][*2]に用いるためのデータの作成（プロセスメトリクスの算出）を適用事例として取り上げ，GPGPU の適用方法について示し，その効果を実験によって確かめる．実験では，Eclipse プロジェクトのバージョン管理システムのデータを対象として，CPU のみを用いた場合と比べて実行時間がどの程度短縮できるかを評価する．

以降, 第 2 章ではリポジトリマイニング, および, GPGPU について述べる．第 3 章ではリポジトリマイニングに対して GPGPU を適用する方法について述べる．第 4 章で実験設定，および，実験結果について報告する．第 5 章で関連研究を説明する．第 6 章で本論文の制約を述べる．第 7 章で本論文のまとめと今後の課題について述べる．

2. 準備

2.1 リポジトリマイニング

リポジトリマイニングとは，版管理システムやバグ管理システムといったソフトウェアリポジトリに蓄積されたデータを分析することにより，ソフトウェア開発における有用な情報を得たり，大規模で複雑なシステムの開発を手助けすることである [4][10]．リポジトリマイニングの工程は図 1 に示すように大きく分けて 2 つの工程に分類される [4].

工程 1：**データ準備**　ソフトウェアリポジトリから分析に必要なデータ（版管理システムのコミットログや，バグ管理システムのバグ報告）を抽出し，プログラム上で扱いやすいデータ形式（CSV や XML）に変換する．さらにその際に，変換されたデータに対して，工程 2 の分析で必要となるメトリクスについても算出し，データウェアハウスに保存する．

工程 2：**データ分析**　工程 1 で準備したデータをもとに分析を行う．分析には，例えば，R[11] や Weka[12] といった統計解析ツールを用いる．

ソフトウェアリポジトリのデータサイズは増加し続け，テラバイトクラスにも及ぶ．そのため，工程 1 において多くの時間を要しているのが現状であり，これはリポジトリマイニング分野における現在の課題の 1 つである．本論文

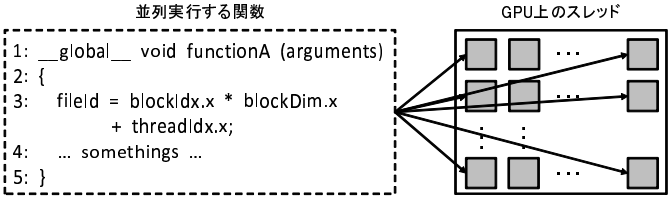

図 2　GPU による並列化
Fig. 2　Parallel Computing using the GPU

[*1] 本論文は筆者らの ICSE2012 ポスター原稿 [7] に実験を追加し、論文としてまとめたものである．

[*2] モジュールの特性値を説明変数とし，モジュールの不具合の有無を目的変数とする数学的モデル

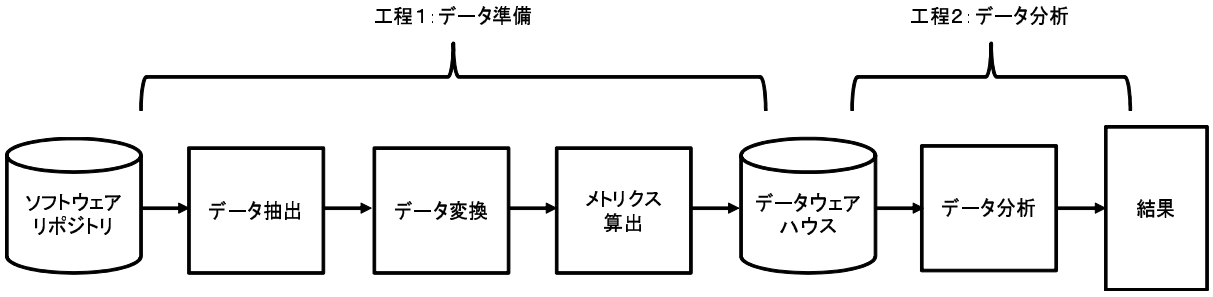

図 1 リポジトリマイニングの工程
Fig. 1 Process of mining software repositories

では，工程 1 におけるメトリクス算出の部分の高速化を達成するために，GPGPU の適用に着目する．

2.2 GPGPU

GPU はコンピュータシステムにおいて画像処理を担うハードウェア部品である [13]．3D の描画等の特定の処理において非常に高い性能を有する．また，GPU は多くのプロセッサの集合体であり，大量のデータを複数のプロセッサで並列に処理できる強力な演算装置である．GPGPU とは，この GPU を汎用的な目的に利用することにより，CPU(Central Processing Unit) では処理しきれない大量のデータを並列処理する技術のことである．

GPGPU を用いたアプローチは，一般的なアプローチ（つまり，CPU のみを用いた場合）に比べ，分岐の多い処理の実行が苦手である．これは，GPU が実行単位ごとに同じ命令を実行するため，全ての分岐処理を実行しなければならなくなるためである．

CUDA(Compute Unified Device Architecture)[*3] を用いた場合の処理の並列化について述べる．CUDA にはスレッドと呼ばれるプログラム実行単位が存在する．スレッドは，動作するプログラムの最小単位であり，このスレッドが GPU 上で大量に並列実行される．GPU 上で並列処理を行うには，並列実行する部分を関数（図 2 の function A）として実装する．functionA 中の “blockIdx.x * blockDim.x + threadIdx.x” という記述によって，各スレッドが担当するデータを一意に特定する．そして，各スレッドは GPU のプロセッサ（コア）上で実行される．

具体的な処理の流れとしては，まず並列処理に必要なデータを CPU 側から GPU 側に転送し，並列処理を開始する．GPU 上で，あらかじめ指定しておいた数のスレッド（図 2 右の灰色四角）が起動され，実行に必要なデータの準備が完了したスレッドからプロセッサ上で実行されていく．GPU は CPU とは異なり複数のプロセッサ（本研究で用いた GPU のコア数は 480）を有しており，各プロセッサがスレッドを独立に実行できるため，GPU 上ではプロセッサの数のスレッドが同時に実行される．全てのスレッドの実行が終わると，その結果を CPU 側に転送する．この流れにおいて CPU—GPU 間のデータ転送が行われ，一定の時間が余分にかかるため，GPU を用いるからといって必ずしも実行時間の短縮につながるとは限らない．

[*3] GPU 向けの C 言語の統合開発環境.

3. GPGPU を用いたリポジトリマイニングの並列化

3.1 概要

本研究では，図 1 に示したポジトリマイニングの工程のうち，工程 1 の部分に着目する．工程 1 は大きく分けて，以下の 3 つの部分に分かれる．

手順 1．データ抽出 リポジトリから分析の対象データ（版管理システムのコミットログや，バグ管理システムのバグ報告）を取り出す．

手順 2．データ変換 手順 1 で取り出したデータをプログラム上で扱いやすいデータ形式（CSV や XML）に変換する．

手順 3．メトリクス算出 手順 2 で変換したデータを用いメトリクスを算出する．

GPGPU をリポジトリマイニングに適用する際に様々な分野があるが，今回は不具合モジュール予測 [14][17][18] に用いるためのデータの作成を対象に選んだ．不具合モジュール予測は，リポジトリマイニング研究の 1 つとして広く取り組まれているテーマの 1 つであり，ソフトウェアテストやレビューの効率化に寄与する有用な分野の 1 つである．また，データの作成にはリポジトリのコミットログを用いるものがあり，その算出には GPGPU 導入について一定の効果があるものと期待される．

一般に不具合モジュール予測において用いられるメトリクスは，プロダクトメトリクス（例えば，ソースコード行数やサイクロマティック複雑度）とプロセスメトリクス（例えば，コードの変更回数）の 2 つに区分される．プロダクトメトリクスはプログラム解析が必要である一方，プロセスメトリクスはプログラム解析を行う必要がなく，版管理システムのコミットログから計測可能であるため，本研究ではプロセスメトリクスの算出に着目する．なお，不具合モジュール予測においてはプロセスメトリクスを用いるほ

表 1 プロセスメトリクスの一覧
Table 1 List of process metrics

カテゴリ	メトリクス	説明	計算量 *4
Traditional Factors	pre_defects	リリース前のバグの数	$O(m*n)$
	pre_changes	リリース前の変更回数	$O(m*n)$
	file_size	ファイルの行数	$O(m)$
Co-change Factors	num_co-changed_files	コミットごとの同時変更ファイル数	$O(m*n^2)$
	size_co-changed_files	コミットごとの同時変更ファイルのサイズ	$O(m*n^2)$
	modification_size_co-changed_files	コミットごとの同時変更ファイルの変更行数	$O(m*n^2)$
	num_changes_co-changed_files	ファイルごとの同時変更されたファイルの変更回数の和	$O(m*n+m*n^2)$
	pre_defects_co-changed_files	コミットごとの同時変更ファイルのリリース前のバグ数	$O(m*n+m*n^2)$
Time Factors	latest_change_before_release	最後の変更からリリースまでの時間	$O(m*n)$
	age	ファイルの年齢	$O(m*n)$
Change Factors	modification_size_files	ファイルの合計追加・削除行数	$O(m*n)$
	modification_size_package	パッケージの合計追加・削除行数	$O(m*n)$
	num_authors_files	ファイルごとの，そのファイルを変更した編集者数	$O(m*n)$

*4 m は全ファイル数，n は全コミット数を表す．

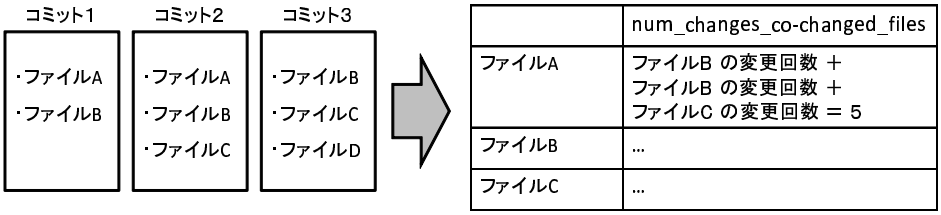

図 3 num_changes_co-changed_files の算出
Fig. 3 Calculation of num_changes_co-changed_files

うが適しているといわれている [14][15][16][17].

リポジトリマイニングの全工程は図 1 に示した通りであるが，今回の事例では手順 3 のメトリクスの算出に GPGPU を適用した．その理由は，本研究ではデータセットとして CVS リポジトリのコミットログを用いるが，CVS リポジトリの場合，手順 1 のデータ抽出を簡単に実行可能なコマンド (cvs log) が存在するためである．また，手順 2 については，コミットログを正規表現などで構文解析しデータ変換するといったパース処理は，分岐の処理が多く GPU のアーキテクチャに不向きであると考えたからである．これらの適用範囲の拡大に関しては今後の課題で詳細を述べる．

3.2 プロセスメトリクス算出のケース

不具合モジュール予測において用いられる主なプロセスメトリクスを表 1 に示す [3]．これらのプロセスメトリクスは、版管理システムのリポジトリにおけるコミットログから計測される．プロセスメトリクスはファイル単位で計測され，例えば，num_co-changed_files は，コミットごとの同時変更ファイル数である．一定期間に複数回変更されている場合は，それぞれの同時変更ファイル数の合計となる．同時に多くのファイルを変更するほど，変更ファイルに欠損やバグが含まれる可能性が高いと考えられる．

Num_changes_co-changed_files は，あるファイルが変更された際，同時に変更された他のファイルの変更回数を全て足し合わせた数値である．例えば図 3 のようにコミット 1，2，3 があるとき，ファイル A はコミット 1 でファイル B と，コミット 2 でファイル B，ファイル C と同時に変更されている．ファイル A の num_changes_co-changed_files の値はコミット 1 での同時変更回数 + コミット 2 での同時変更回数= (ファイル B の変更回数) + {(ファイル B の変更回数)+(ファイル C の変更回数)} である．ファイル B はコミット 1，2，3 で変更されているためファイル B の変更回数は 3，ファイル C はコミット 2，3 で変更されているため変更回数は 2 である．したがって，ファイル A の num_changes_co-changed_files の値は 5 (=3+3+2) である．頻繁に変更されるファイルと同時に変更されるファイルの num_changes_co-changed_files の値は大きくなるため，この値が大きいファイルは欠損やバグが含まれている可能性が高いと考えられる．

表 2 プロセスメトリクスの一部を逐次処理により求める時間
Table 2 Time to calculate the part of the process metrics using only CPU

	modification_ size_files	num_ authors_files	num_changes_ co-changed_files
実行時間 [秒]	0.000	0.010	1162.330

本論文の実験で用いる CVS のような非トランザクションベースの版管理システムでは，1 コミットの単位は単一のソースコードファイルであり，本来，同時変更された

Algorithm 1 ステップ 1 の逐次処理アルゴリズム

```
 1: changeNum ← 0
 2: for all files do
 3:    for all commits do
 4:       if file.id = commit.id then
 5:          changeNum ← changeNum + 1
 6:       end if
 7:    end for
 8:    changeNums[file.id] ← changeNum
 9:    changeNum ← 0
10: end for
```

ファイル群が 1 つのコミットとしてまとめられていない．そこで，一般的な経験則を利用して，本来，同時変更されたファイル群を 1 つのコミットとしてグループ化する [19]．本研究では，Zimmermann ら [20] 同様に，開発者，および，コミット時のメッセージが同一であり，かつ，ほぼ同じ時刻に行われたコミット群を 1 つのコミットとしてみなした．また，200 秒以内に行われたコミット群をほぼ同じ時刻に行われたコミットとした [20]．

3.3 従来手法によるプロセスメトリクスの算出法

表 1 に示したプロセスメトリクスのうち，最も計算量が小さいグループから modification_size_files, num_authors_files を，最も大きい計算量が大きいグループから num_changes_co-changed_files の計 3 つ[*1] を選択し，それぞれ従来の CPU のみを用いる手法で実装し，実行時間を計測する．4.3 節で示すコミットログの 100,000 行分について，これらのプロセスメトリクスを求めるために要した時間の計測結果を表 2 に示す．

表 2 より，3 種類のメトリクスの中で num_changes_co-changed_files を求める処理に多くの時間がかかっていることがわかる．これは，num_changes_co-changed_files を逐次処理で求める処理は $O(m*n+m*n^2)$ の計算量が必要であり，他の 2 つより多くの計算量が求められるためだと考えられる．

要した時間の最も大きかった num_changes_co-changed_files を従来の CPU のみによる実装方法で求める場合について詳細を説明する．本研究では num_changes_co-changed_files の計算を以下の 2 つのステップで求める．

ステップ 1　各ファイルの変更回数を計算

- $O(m*n+m*n^2)$ の $m*n$ の部分

ステップ 2　num_changes_co-changed_files を計算

- $O(m*n+m*n^2)$ の $m*n^2$ の部分

まずステップ 1 で，各ファイルの変更回数を計算する．それには，ファイルごとに，編集されているコミット数を調べるため，逐次処理で実装するには Algorithm 1 に示すような 2 重ループが必要となり，（ファイル数）×（更新履歴数）回ループする．4 行目の commit.id は，当該コミットで編集されているファイルの ID を示す．

[*1] file_size は計算量が最も小さいものの，メトリクス計算時に更新履歴を用いないので，初期実験の対象から外した．

Algorithm 2 ステップ 2 の逐次処理アルゴリズム

```
 1: changeNum ← 0
 2: for all files do
 3:    for all commits do
 4:       if file.id = commit.file.id then
 5:          for all commits′ do
 6:             if commit.id = commit′.id then
 7:                if file.id != commit′.file.id then
 8:                   changeNum+ = changeNums[file.id]
 9:                end if
10:             end if
11:          end for
12:       end if
13:    end for
14:    NumChangesCoChangedFiles[file.id] ← changeNum
15:    changeNum ← 0
16: end for
```

そしてステップ 2 で，num_changes_co-changed_files を計算する．Algorithm 2 に示すように，ファイルごとに，そのファイルがどのコミットで編集されているかを調べ（4 行目），次に，そのコミットそれぞれに対して，同時に編集されている他の全てのファイルの変更回数を加算する（7 行目）．ここでの変更回数とはステップ 1 で求めておいたものなので，値を取得するのみである．ステップ 2 では 3 重ループが必要となり，（ファイル数）×（更新履歴数）×（更新履歴数）回ループする．

なお，本研究では，num_changes_co-changed_files を計算する際に，事前に各ファイルの変更回数を計算するためにステップ 1 と 2 に分割するアプローチを取った．もし事前に各ファイルの変更回数を求めない場合，Algorithm 2 の 8 行目で毎回，Algorith 1 の 3 行目から 7 行目を実行するので，より多くの計算量 $O(m*n^3)$ が必要である．そのため，本研究ではステップ 1 と 2 に分割するアプローチを取った．

3.4 GPGPU の適用・実装方法

GPGPU を適用し，GPU の 1 スレッドで 1 ファイルに対する処理を行うこと（並列化すること）で，num_changes_co-changed_files を求める処理の高速化を行う．まずステップ 1 では，図 4 に示すように GPU の 1 スレッドで 1 ファイルの変更回数を計算する．

これにより，Algorithm 1 の 3 行目から 9 行目に関する処理を並列実行することができる．ステップ 1 の並列処理部の CUDA プログラムをソースコード 1 に示す．これによって（ファイル数）×（更新履歴数）/（同時並列実行数）回分逐次実行することになり，高速化できる．ソースコー

ソースコード 1　ステップ 1 の CUDA コード

```
1  int i, fid, changeNum = 0;
2  fid = blockIdx.x*blockDim.x+threadIdx.x;
3  for(i = 0; i < N; i++){
4    if(commit[i] == fid){
5      changeNum++;
6    }
7  }
8  changeNums[fid] = changeNum;
```

ソースコード 2　ステップ 2 の CUDA コード

```
1  int i, j, fid, cid, changeNum = 0;
2  fid = blockIdx.x*blockDim.x+threadIdx.x;
3  for(i = 0; i < N; i++){
4    if(commit[i*2+1] == fid){
5    /* commitの奇数番目にcommit.idを記録,
6       commitの偶数番目にfile.idを記録 */
7      cid = commit[i*2];
8      for(j = 0; j < N; j++){
9        if(commit[j*2] == cid &&
10         commit[j*2+1] != fid){
11         changeNum +=
12         NumChangesCoChangedFiles[commit[j*2+1]*2+1];
13       }
14     }
15   }
16 }
17 NumChangesCoChangedFiles[fid] = changeNum;
```

ド 1 において，blockIdx.x はブロック ID，blockDim.x は 1 つのブロック内のスレッドの数，threadIdx.x はスレッド ID である．ブロック ID，スレッド ID はそれぞれのブロック，スレッドに割り当てられる ID であり，ブロック同士，スレッド同士の識別に使用される．ここでは，スレッドのスレッド ID とそのスレッドが所属するブロックのブロック ID により，そのスレッドがどのファイルを担当するかを決定している（2 行目）．その後更新履歴を調べ（4 行目），担当ファイルの ID を見つけるごとに変更回数に 1 を加算する（5 行目）．ここでは更新履歴のデータは全て行列に格納している．

次にステップ 2 では，ステップ 1 と同様に GPU の 1 スレッドで 1 ファイルに対する処理を行う．具体的には，そのファイルがどのコミットで編集されているかを調べ，そのコミットそれぞれに対して，同時に編集されている他の全てのファイルの変更回数の合計を求める．ステップ 2 の並列実行部の CUDA プログラムをソースコード 2 に示す．ソースコード 2 では，ステップ 1 のときと同様に Algorithm 2 の 3 行目から 13 行目に関する処理を並列実行する．まず各スレッドが担当するファイルを決定する（2 行目）．その後，コミットログを調べ（3 行目），担当のファイル ID を見つけると（4 行目），その変更のコミット ID を記憶する（7 行目）．そしてコミットログを先頭から調べ直し（8 行目），担当ファイルとは別のファイルを更新している同コミットを見つけると（9 行目），そのファイルの合計変更回数（ステップ 1 で求めたもの）を結果に足していく（10 行目）．これによって各スレッド内で 2 重ループを実行することになり，（ファイル数）×（更新履歴数）×（更新履歴数）/（同時並列実行数）回分逐次実行することになり，高速化できる．

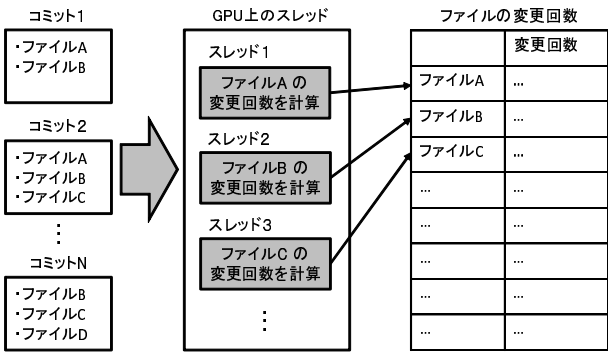

図 4　GPGPU を用いたプロセスメトリクス算出の概要

Fig. 4　Overview of calculation for process metrics using GPGPU

4. 評価実験

4.1 概要

実験の目的は，GPGPU を用いたリポジトリマイニングの有用性を評価することである．実験として，リポジトリマイニング研究の 1 つとして広く取り組まれているテーマの 1 つである，不具合モジュール予測 [14][17][18] に用いるためのデータの作成（プロセスメトリクスの算出）を取り上げる．具体的には，num_changes_co-changed_files を，従来の CPU のみの逐次処理で求めた場合と，3.4 節で述べた GPGPU を用いて求めた場合について比較し，評価する．評価尺度は，実行時間である．

4.2 環境

評価を行う実行環境は，CUDA 4.0 を用いた．使用した CPU は，Intel(R) Core(TM) i7 CPU 930，GPU は，NVIDIA(R) GeForce GTX 480 である．CPU の環境詳細を表 3 に，GPU の環境詳細を表 4 に示す．GPGPU を用いる手法は，GPU を使用する部分を除き，CPU のみを用いる手法と同じ環境である．

4.3 データセット

今回用いたリポジトリのデータは，MSR Mining Challenge 2008[*5]により提供された Eclipse の CVS リポジトリのコミットログである．このデータセットの統計を表 5 に示す．

[*5] http://msr.uwaterloo.ca/msr2008/challenge/index.html

表 3　実験に用いたハードウェア環境
Table 3　Hardware environment used in the experiment

	GPGPU を用いる手法	CPU のみを用いる手法
CPU	Intel(R) Core(TM) i7 CPU 930 (2.80GHZ), 4 cores	Intel(R) Core(TM) i7 CPU 930 (2.80GHZ), 4 cores
Memory	16 GB	16 GB
OS	Ubuntu 11:10f	Ubuntu 11:10f
Disk type	SCSI 2TB	SCSI 2TB
GPU	NVIDIA(R) GeForce GTX 480 (1.40GHZ), 480 cores	-

表 4　用いた GPU の性能
Table 4　Performance of the GPU used in the experiment

項目	スペック
GPU	NVIDIA(R) GeForce GTX 480
SM	15 個
SM 当たりの SP	32 個
CUDA プロセッサーコア	480 個
GPU クロック	1.40GHz
メモリクロック	1,848.00Mhz
ワープ	32
グローバルメモリ	1,536 MBytes
1 ブロックのレジスタ数	32,768
1 ブロックの最大スレッド数	1,024
ブロックの各次元の最大サイズ	1,024 × 1,024 × 64
グリッドの各次元の最大サイズ	65,535 × 65,535 × 65,535

4.4　結果

num_changes_co-changed_files を，従来の CPU のみの逐次処理で求めた場合と，GPGPU を用いて求めた場合の実行時間を比較する．結果を表 6 に示す．

表 6 の通り，従来手法（CPU のみ）を用いた場合、50,000 行のデータに対してはデータ変換（図 1 の手順 2 に該当）で 2.69 秒，メトリクス算出（手順 3 に該当）で 58.47 秒の時間を要した．また，100,000 行のデータに対しては，それぞれで 14.98 秒と 1162.33 秒の時間を要した．一方で，GPGPU を用いた場合，データ変換には従来手法と同じく CPU を用いたため要した時間は同じであるが，メトリクス算出では 50,000 行のデータに対しては 5.21 秒，100,000 行のデータに対しては 11.81 秒の時間しかかからなかった．つまり，提案手法では num_changes_co-changed_files を求める時間が，50,000 行分については 7.7 倍，100,000 行分については 43.9 倍速くなることがわかった．

5.　関連研究

ソフトウェア工学以外の分野においては，生物学や暗号化の分野では既に GPGPU の効果が認められている．Trapnell らは，GPU を用いることで DNA に関する大量の計算を高速化している [9]．スタック無しの深さ優先探索を用いることで CPU による計算よりも 13 倍高速化することができたと報告している．また，Manavski, S.A. らは，対象鍵暗号に関するソリューションに対して GPU を用いることで高速化が可能であることを示した [8]．提案手法では OpenSSL よりも 20 倍の速さで実行できたと報告している．

ソフトウェア工学の分野では，静的コード解析の一種であるポインタ解析に GPU を利用している [21]．しかし，ランダムに生成したグラフを入力とした場合，GPU による実装は CPU による実装よりも，実行時間が中央値で 5 倍ほどかかることが報告されている．また，実際のソフトウェアである gdb に対しても適用している．gdb から得られるグラフは頂点数 335,667，辺数 334,374 であり，GPU による実装は CPU による実装に比べて，実行時間が約 3.11 倍かかっている．深谷らはこれを，行列計算は GPU に適しているものの，グラフを表す行列の要素は 0 と 1 しか存在しないため，演算にかかるコストが高いためであると考察している．

表 5　データセットの統計
Table 5　Statistics of the data set

	50,000 行	100,000 行
ログの平均期間	2001.4-2001.7	2001.4-2001.10
ファイル数	6,362	17,517
変更情報数	34,327	71,660
平均変更コミット数/ファイル	5	4

表 6　num_changes_co-changed_files を求める実行時間の比較
Table 6　Comparison of time to calculate the num_changes_co-changed_files

50,000 行	データ変換 [秒]	メトリクス算出 [秒]	合計 [秒]
CPU のみ	2.690	58.470	61.160
GPU 使用	2.690	5.210	7.900
100,000 行	データ変換 [秒]	メトリクス算出 [秒]	合計 [秒]
CPU のみ	14.980	1162.330	1177.310
GPU 使用	14.980	11.810	26.790

6.　本論文の制約

GPGPU による実装と従来の実装との比較はプログラミングの能力に依存する．本実験では，著者らのうち二人の博士課程前期の学生が実装した．両名とも GPGPU による実装は 3 ヶ月程度であるものの，実行時間を 43.9 倍向上させた．なお，今回の GPGPU の適用においては，高速化のための共有メモリの利用やパラメータの調整を行っていない．そのため，GPGPU での実装に要した工数と従来の実装に要した工数の間に大きい違いはなかった．

また，本論文の制約として，一つのデータセットに対して一つのケーススタディにおいて実装した点が挙げられる．本研究の結果を一般化するには，他のデータセットや代表的なケーススタディに対しても解析する必要がある．また，

GPGPU による実装をスーパーコンピュータや Hadoop などの MapReduce とも比較することは，重要な課題である．

7. おわりに

本論文では，GPGPU を用いたリポジトリマイニングの有用性を評価するために，リポジトリマイニング研究の 1 つとして広く取り組まれているテーマの 1 つである，不具合モジュール予測に用いるためのデータの作成（プロセスメトリクスの算出）を適用事例として取り上げ，GPGPU の適用方法について示し，その効果を実験によって確かめた．実験から，GPGPU を用いることにより，num_changes_co-changed_files を求める処理を，コミットログ 50,000 行分については 7.7 倍，100,000 行分については 43.9 倍高速化することがでることがわかった．

今後の課題としては，他のデータセットや代表的なケーススタディについての分析を行い，本研究の知見を一般化すること，および，GPGPU の性能とスーパーコンピュータや Hadoop 等の MapReduce の性能とを比較することである．

謝辞 本研究の一部は, 科学技術振興事業団「JST」の戦略的基礎研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」の研究課題「ポストペタスケール時代のスーパーコンピューティング向けソフトウェア開発環境」による助成を受けた．また、本研究の一部は，日本学術振興会科学研究費補助金（若手 A：課題番号 24680003）による助成を受けた．

参考文献

[1] A. E. Hassan: “The road ahead for mining software repositories,” *Proc. Frontiers of Software Maintenance (FoSM)*, pp.48–57 (2008).

[2] 小林隆志, 林晋平: “データマイニング技術を応用したソフトウェア構築・保守支援の研究動向,” コンピュータソフトウェア, Vol.27, No.3, pp.13–23 (2010).

[3] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan: “High-impact defects: A study of breakage and surprise defects,” *Proc. European Softw. Eng. Conf. and Symp. on the Foundations of Softw. Eng. (ESEC/FSE)*, pp.300–310 (2011).

[4] W. Shang, B. Adams, and A. E. Hassan: “Using pig as a data preparation language for large-scale mining software repositories studies: An experience report,” *Journal of Systems and Software*, Online (2011).

[5] A. Mockus: “Amassing and indexing a large sample of version control systems: Towards the census of public source code history,” *Proc. Int’l Working Conf. on Mining Software Repositories (MSR)*, pp.11–20 (2009).

[6] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan: “Mapreduce as a general framework to support research in mining software repositories,” *Proc. Int’l Working Conf. on Mining Software Repositories (MSR)*, pp.21–30 (2009).

[7] R. Nagano, H. Nakamura, Y. Kamei, B. Adams, K. Hisazumi, N. Ubayashi, and A. Fukuda: “Using the GPGPU for scaling up mining software repositories,” *Proc. Int’l Conf. on Software Engineering (ICSE)*, Poster Session, pp.1435–1436 (2012).

[8] S. A. Manavski: “CUDA compatible GPU as an efficient hardware accelerator for AES encryption,” *Proc. Int’l Conf. on Signal Processing and Communications (ICSPC)*, pp.24–27 (2007).

[9] C. Trapnell and M. C. Schatz: “Optimizing data intensive GPGPU computations for DNA sequence alignment,” *Parallel Computing, Vol.35, No.8-9*, pp.429–440 (2009).

[10] H. H. Kagdi, M. L. Collard, and J. I. Maletic: “A survey and taxonomy of approaches for mining software repositories in the context of software evolution,” *Journal of Software Maintenance, Vol.19, No.2*, pp.77–131 (2007).

[11] The R Foundation. R. `http://www.r-project.org/`.

[12] Machine Learning Group at University of Waikato. Weka. `http://www.cs.waikato.ac.nz/~ml/weka/`.

[13] 岡田賢治: “CUDA 高速プログラミング入門,” 株式会社秀和システム (2010).

[14] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan: “Revisiting common bug prediction findings using effort aware models,” *Proc. Int’l Conf. on Software Maintenance (ICSM’ 10)*, pp. 1–10, (2010).

[15] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy: “Predicting fault incidence using software change history,” *IEEE Transactions on Software Engineering, Vol.26, No.7*, pp. 653–661, (2000).

[16] N. Nagappan and T. Ball: “Use of relative code churn measures to predict system defect density,” *Proc. Int’l Conf. on Software Engineering (ICSE’ 06)*, pp. 284–292, (2005).

[17] R. Moser, W. Pedrycz, and G. Succi: “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” *Proc. Int’l Conf. on Software Engineering (ICSE’ 08)*, pp. 181–190, (2008).

[18] O. Mizuno and T. Kikuno: “Prediction of fault-prone software modules using a generic text discriminator”, *IEICE Transactions on Information and Systems, Vol.E91-D, No.4*, pp.888-896, (2008).

[19] S. Kim, E. J. Whitehead, Jr., and Y. Zhang: “Classifying software changes: Clean or buggy?,” *IEEE Trans. Softw. Eng., 34(2)*, 181–196, (2008).

[20] T. Zimmermann and P. Weisgerber: “Preprocessing cvs data for fine-grained analysis,” *Proc. Int’ l Workshop on Mining Software Repositories (MSR’ 04)*, pp.2–6, (2004).

[21] 深谷敏邦，権藤克彦: “GPU を利用したポインタ解析の実装と評価,” ソフトウェア工学の基礎 XVIII，近代科学社, pp.259–260 (2011).