データストリーム処理における GPU タスク並列を用いたスケーラブルな異常検知機構の実現

上野 晃司† 鈴村 豊太郎†‡

†東京工業大学 〒152-8550 東京都目黒区大岡山 2-12-1
‡IBM 東京基礎研究所 〒242-8502 神奈川県大和市下鶴間 1623-14
E-mail: †ueno.k.ac@m.titech.ac.jp, ‡suzumura@cs.titech.ac.jp

あらまし データストリーム処理とは、データを蓄積することなくリアルタイムに処理することで、本論文では、データストリーム処理アプリケーションの 1 つである変化点検知の GPU による高速化を扱う．変化点検知では、大量のデータに対して小規模な計算を繰り返す場合が多く、これまでの GPGPU による手法では高速化できなかった．そこで、GPU タスク並列による高速化を提案する．変化点検知アルゴリズムの 1 つである特異スペクトル変換は、計算量の多くを特異値分解が占めているが、特異値分解の計算のうち二重対角化を GPU タスク並列で実装し、行列サイズ 320、タスク数 256 のとき、CPU 1 コアに対して、17.22 倍の高速化を達成した．

キーワード データストリーム処理、異常検知、SST、GPGPU、GPU タスク並列、特異値分解

Scalable Anomaly Detection on Data Stream Processing with GPU Task Parallelism

Koji UENO† and Toyotaro SUZUMURA†‡

†Tokyo Institute of Technology 2-12-1 Ookayama, Meguro-ku, Tokyo, 152-8550 Japan
‡IBM Research - Tokyo 1623-14 Shimotsuruma, Yamato-shi, Kanagawa, 242-8502 Japan
E-mail: †ueno.k.ac@m.titech.ac.jp, ‡suzumura@cs.titech.ac.jp

Abstract Recently data stream processing has been extensively studied as the new computing paradigm for processing a massive amount of streaming data in real time without storing them on the secondary storage. In this paper we propose a new task parallelism method on GPGPU to improve the performance and scalability of a real-time anomaly detection algorithm called SST (Singular Spectrum Transformation) for a massive amount of sensor data. Since the main computationally dominant part of SST is a singular value decomposition, we successfully implemented the bidiagonalization with the proposed task parallelism. Our experimental result demonstrates that our optimization achieves 17.22 times performance gain on GPU against 1 core CPU when the number of tasks is 256 and the matrix size is 320.

Keyword Data Stream Processing, Anomaly Detection, SST, GPGPU, GPU Task Parallelism, SVD

1. はじめに

近年、デジタルデータの用途多様化に伴い、大量のデータが容易に入手できるようになり、そこからの知識発見が重要な課題となっている．知識発見の１つに、リアルタイム異常・変化点検知がある．これは、インターネットトラフィックのモニタリングによるネットワーク異常の検知や、サーバのオンラインデータ観測によるボットネットなどのサイバーテロ監視、株取引や工場の生産ラインの監視など広範囲に応用でき、需要は高まっている．これに対し、データストリーム処理は、このようなリアルタイム処理を可能とする計算パラダイムであり、昨今活発に研究されている．

また、近年、一般向け PC に搭載されるグラフィック処理ユニット（Graphics Processing Unit, GPU）の高い計算性能が注目され、GPU をグラフィック以外の汎用計算で利用するための技術（General-Purpose computing on Graphics Processing Units, GPGPU）の研究が盛んに行われている。CPU と比較して、GPU はシンプルなコアを多数集積したプロセッサと高速なメモリが特徴である。これらの特徴を活かすことができれば CPU の数倍の性能を発揮する。

本論文では、GPU を用いた変化点検知の高速化を考える．以下の章では、2 章にデータストリーム処理と変化点検知アルゴリズム特異スペクトル変換の概要、3 章に変化点検知に GPGPU を適用する際の問題と GPU タスク並列による解決方法、4 章に特異値分解の GPU タスク並列実装の詳細、5 章で性能評価、6 章に関連研究、

7 章でまとめと今後の展望について述べる．

本論文による貢献を以下に示す．

1. 多数の時系列データを同時に処理するため、GPU タスク並列を提案した．
2. GPU タスク並列のアプリケーションとして、変化点検知アルゴリズム特異スペクトル変換を扱い、特異値分解の計算のうち二重対角化を実装した．これまでの GPGPU による手法では高速化できなかった行列サイズ 320 の場合でも、タスク数 256 のとき、CPU 1 コアに対して、17.22 倍の高速化に成功した。これにより、GPU タスク並列の有用性を示した．
3. GPU タスク並列により変化点検知を高速化する場合に問題となる点や、一般的な GPGPU と異なる点について、特異値分解の GPU タスク並列実装を例に取り、示した．

2. データストリーム処理と変化点検知

この章では、データストリーム処理の概要、データストリーム処理アプリケーションの１つである変化点検知、及び変化点検知アルゴリズムの１つである特異スペクトル変換[5]について述べる．

2.1 データストリーム処理

データストリーム処理とは、止め処なく生成される情報の流れをストリームと呼び、このストリームを蓄積することなく逐次処理していくという新しい計算パラダイムである．バッチ処理と呼ばれる計算対象を全てストレージに蓄積してから計算する従来の手法と違い、リアルタイムの応答が要求される場合や、時系列で前後する僅かなデータのみを参照すればよい計算や、全データの蓄積が物理的に困難な処理に適している．このような手法は音声や動画のストリーミングなど一部の処理では利用されていたが、データストリーム処理はこれを抽象・汎用化し、幅広い処理に対して適用できるよう洗練された処理系としてまとめられている点が従来とは異なっている．

分散環境上で実行可能なデータストリーム処理系として、M.I.T.の Borealis[15]や IBM Research の System S[16]などが存在する。

なお、後述する GPGPU の開発環境 CUDA には、処理の連続性を保証するストリームという概念があるが、それはデータストリーム処理におけるストリームとは異なる．

2.2 変化点検知アルゴリズム SST

時系列データの変化点とは、データ生成機構にある変化が生じた時点と定義される．

データストリーム処理で扱うべき様々なアプリケーションの中でも異常・変化点検知は最もレイテンシが重視されるアプリケーションの１つである．変化点検知アルゴリズムは何種類も存在し、計算処理が軽い単純なものや計算量が多く複雑なアルゴリズムで変化点を検出するものまで多様であるが、どのアルゴリズムが最適かはアプリケーション依存である．本論文では特に、ヘテロ環境のシステムで動的な異常・障害検知を行うことができるが、計算量が多いことで知られる変化点検知アルゴリズム、特異スペクトル変換[5]（Singular Spectrum Transformation, SST）を扱う．

SST は、他の手法とは異なり特定の確率モデルを仮定しない為、入力時系列の多様性に比較的頑強であり局所解の心配がない．実用例として自動車のセンサーによる監視やサーバ群のエラー検知などコンピュータシステムや機械システムなどで用いられている．

アルゴリズムを以下に示す．

1. 時系列データ$\tau = \{x_t | t = (\text{実数全体})\}$を考える．ウィンドウサイズを$w$とおき、長さ$w$の部分系列を列ベクトルとして$s(t) = (x_{t-w+1}, \cdots, x_{t-1}, x_t)^T$とおく．
2. 部分系列をn本並べた行列として、
$$H(t) = [s(t-n), \cdots, s(t-2), s(t-1)]$$
$$G(t) = [s(t-n+\gamma), \cdots, s(t-1+\gamma)]$$
を定義する．ただし、γは正整数である．
3. $H(t)$を特異値分解(Singular value decomposition, SVD)して、特異値の大きい順に左特異ベクトルを$r(< w, n)$個求め、$u_t^{(1)}, u_t^{(2)}, \cdots, u_t^{(r)}$とし、過去の代表パターンとする．
4. $G(t)$を特異値分解して、最大左特異ベクトルμ_tを現在の代表パターンとする．
5. 変化度スコアは、
$$z(t) = 1 - \frac{\mu_t^T U_t \mu_t}{\|\mathrm{U}_t^T \mu_t\|} \quad (1)$$
で計算される．ただし、$\mathrm{U}_t = \left[u_t^{(1)}, u_t^{(2)}, \cdots, u_t^{(r)}\right]$である．

SST では SVD を使うが、SVD は非常に計算量の多い演算であり、素直に SVD を計算する実装では、計算が間に合わずリアルタイム処理ができない．そこで、何らか高速化手法を適用する必要がある．

また、例えばセンサーが複数あり、各センサーから一定間隔でデータが取得できる場合のように、複数の時系列データを扱う場合がある．SST ではそれぞれの時系列データを独立に扱い計算することになる．変化点検知においては多数の時系列データを扱うことが多く高速化が望まれている．

3. GPU タスク並列による変化点検知の高速化

変化点検知は、大量のデータに対して同じ処理を繰り返すことが多く GPU を使った高速化が期待できる．

3.1 SST 高速化における GPGPU 適用の問題

SST の計算において、SVD の高速化が重要となる．

GPU を用いた SVD の高速化については、これまでに Sheetal Lahabar らによる研究[3]や、深谷らによる研究[6]

がある. また、数値計算ライブラリ LAPACK を CUDA[2]上に移植した CULA[7]には、SVD を計算する関数も用意されている．また、森田らは SVD の計算に CULA を使うことで SST 計算の高速化[1]を試みている．

しかし、これらの研究やライブラリをリアルタイム変化点検知に適用するには問題がある．SST の場合、大量のデータに対して SVD を繰り返すことになるので、リアルタイム処理するためには、計算量の関係から行列サイズを数十～数百に抑えなければならない．しかし、これまでの GPU による高速化の研究は、非常に大きな行列を対象としている．したがって、 SST の高速化には適用できない.

実際、森田ら[1]は、CULA を用いて GPU による高速化を研究したが、ウィンドウサイズ 400 以下では、GPU は CPU よりも遅いという結果だった．また、同様に森田らの研究によれば、ウィンドウサイズ 450 以上では、GPU は CPU よりも速いという結果だったが、ウィンドウサイズ 450 のとき、GPU での SST の計算時間は、1.68 秒であった．これでは時間がかかりすぎて、リアルタイム処理ができない. 行列サイズを大きくすれば、GPU の優位性はより大きくなるが、SST の計算時間が増加することは自明である．かつ、センサーの数は高々１で実験しており、複数のセンサーに対する性能向上は見られていない.

小さな行列で高速化できない理由は、行列サイズが小さいと、GPU の性能を引き出すための十分な並列性を確保できないからである.

SVD の高速化に関する他の研究も同様である．Sheetal Lahabar らによる研究[3]や深谷らによる研究[6]は、ともに、数値計算ライブラリ BLAS の CUDA 実装である CUBLAS[8]を使用している．CUBLAS は非常に大きな行列を対象にしているので、中・少規模の行列では高速化できない.

3.2 タスク並列による解決

前述の通り、これまでの手法では変化点検知の高速化に GPU を適用するには問題がある．そこで、タスク並列による解決を提案する．GPU において、中・少規模の問題を高速化するには、並列性の確保が課題となる．そこで、複数の時系列データを並列計算するタスク並列を導入する．このタスク並列により、並列性を高めることができ、GPU の性能を十分に引き出せるようになる.

3.3 タスク並列の実現方法

CUDA[2]は NVIDIA の開発した GPGPU の開発環境であるが、本論文では CUDA を使用した．NVIDIA のほとんどの GPU では、カーネル関数は同時に１つしか実行できないため、タスク並列を実現するには、工夫が必要となる.

GPU でのタスク並列については、Marisabel Guevara らによる研究[9]がある．彼らは、CUDA のプログラムが CUDA ブロックごとに独立して動作することに着目し、タスク並列に成功している．この原理を理解するには、NVIDIA GPU のアーキテクチャについて理解する必要がある.

```
__global__ void kernel_func(float* A, int bda, … ){
    /*ブロック ID を使ってデータを識別*/
    float* bA = &A[bda*blockIdx.y];
    /* 何かを実行する */
}
```

図 1 GPU タスク並列カーネル関数コード例

NVIDIA GPU は、多数の Streaming Multiprocessor (SM)と呼ばれる、SIMD ユニットから構成されており、各 SM は独立した命令発行ユニットを持っている．１つの SM は、最大 24~32 個の warp を同時に実行することができる．warp は 32 スレッドで、これは SIMD ユニットの実行単位となる．つまり、同じ SM に割り当てられた warp でも、各 warp は独立に命令ポインタを持ち、擬似的に並列実行されている.

CUDA には、ブロックという概念があり、1 ブロックあたり最大 512 個のスレッドを実行することができる．ブロックごとの同期ポイントを作成することができる．これは、このポイントを通過するとブロック内の全スレッドで同期が取られ、そのポイント以前に実行したメモリ操作命令の結果がブロック内の他の全スレッドから見えるようになる、というものである.

通常、各 SM には複数のブロックが割り当てられる．１つの warp が複数のブロックにまたがることはないので、各ブロックは独立して動作することになる．したがって、ブロックごとに条件分岐で異なる動作をさせるようにすれば、単一 GPU 内で異なるコードを並列実行することができる.

我々は、カーネル関数に引数で渡されたポインタから、ブロック ID を使って各ブロックが異なるオフセットにあるデータを処理することによって、タスク並列を実現した．各ブロックは独立して動作することから、タスク間で異なる動作をさせることも可能である．また、各タスクはコードを共有しているので、タスク数によらず条件分岐や引数の数を一定にすることができる．変化点検知のように大量のデータに対して同じ処理を繰り返す場合、この手法は特に有効である.

図 1 は CUDA のコード例である. 実装にあたっては、プログラムの見やすさから、ブロック ID のy次元をタスク ID として統一して使用した．ブロック ID のx, y次元はそれぞれ最大 65,536 なので、ブロック ID の 1 つの次元だけで、最大 65,536 個のタスクを処理できることになる.

NVIDIA の開発した最新の GPU(Fermi)では、複数カーネルを同時実行できるが、同時実行可能なカーネル数は 16 までである．変化点検知では、数十～数百レベルの時系列データを同時に処理する. Fermi コアの GPU

では、我々の提案手法なら最大 128 タスク（1SM あたり最大 8 ブロックまで並列処理可能で、SM が 16 個あるので）を並列処理できるので、カーネルの同時実行機能に頼るより、我々の手法を使った方がより効率よく計算できる．

また、タスク並列を実現するにあたっては、カーネル関数を変更する必要があるため、コードの公開されていない CUBLAS[8]や CULA[7]は利用できず、すべてのカーネル関数を新たに実装する必要があった．

4. SVD の GPU タスク並列実装

GPUタスク並列のアプリケーションとして､SST を扱い、SVD 計算のうち二重対角化を実装した．この章では、SVD の GPU タスク並列実装の詳細を述べる．

4.1 SVD のアルゴリズム

SST では、密行列の SVD が必要になる．

$m \times n$行列Aの SVD とは、

$$A = U\Sigma V^T \tag{2}$$

と分解することである．ただし､Uは$m \times m$の直交行列、Vは$n \times n$の直交行列、Σは$m \times n$の対角行列である．

SVD の GPU への実装は、Sheetal Lahabar らによる研究[3]を参考にした．彼らは、LAPACK で使われている Golub-Reinsch のアルゴリズムを使って、GPU を用いた高速化に成功している．このアルゴリズムは 2 つのステップから構成される．まず、行列をハウスホルダー変換によって二重対角化する．次に、二重対角行列を shifted QR iteration によって対角化する．

本論文では､2 つのステップのうち二重対角化をタスク並列で GPU に実装した．また、行列は正方行列で、サイズは 512 までの比較的小規模なものに対応させた．計算はすべて単精度(float)で行っている．

二重対角化のステップでは、ハウスホルダー変換を繰り返して、行列Aを、

$$A = QBP^T \tag{3}$$

と分解する．ただし、Bは二重対角行列、QとPは直交行列である．

ここでは、$n \times n$の正方行列Aの二重対角化について説明する．

まず、後述する計算方法により、$A(1{:}n,1)$からベクトル$\boldsymbol{v}^{(1)}$と係数$\tau_v^{(1)}$を求める．また、$A(1,2{:}n)$からベクトル$\boldsymbol{u}^{(1)}$と係数$\tau_u^{(1)}$を求める．すると、

$$\begin{aligned} A^{(1)} &= \left(I - \tau_v^{(1)}\boldsymbol{v}^{(1)}\boldsymbol{v}^{(1)^T}\right) A \left(I - \tau_u^{(1)}\boldsymbol{u}^{(1)}\boldsymbol{u}^{(1)^T}\right) \\ &= \begin{bmatrix} \alpha & \beta & 0 & \cdots & 0 \\ 0 & x & x & \cdots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \cdots & & x \end{bmatrix} \end{aligned} \tag{4}$$

となる．これは、最初の行と列の消去である．

次に$A^{(1)}(2{:}m,2)$,$A^{(1)}(2,3{:}n)$に着目して同様の変換をすると、2 番目の行と列が消去される．これを繰り返すと最終的に二重対角行列Bが計算される．すべての行と列が消去されると、Bは、

$$B = Q^T AP \tag{5}$$

となる．ただし、

$$Q = \prod_{i=1}^{n} H_i, P = \prod_{i=1}^{n-1} G_i \tag{6}$$

H_i ,G_iはそれぞれ、i行,i列の消去に使用されるハウスホルダー行列であり、

$$\begin{aligned} H_i &= I - \tau_v^{(i)}\boldsymbol{v}^{(i)}\boldsymbol{v}^{(i)^T} \\ G_i &= I - \tau_u^{(i)}\boldsymbol{u}^{(i)}\boldsymbol{u}^{(i)^T} \end{aligned} \tag{7}$$

である．

ベクトル$\boldsymbol{y} = [y_1, \cdots, y_l]$に対応する、ベクトル$\boldsymbol{r}$と係数$\tau$は次のようにして求める．

$$\begin{aligned} \alpha &= -\mathrm{sign}(y_1)\left||\boldsymbol{y}\right|| \\ a &= \mathrm{sign}(y_1)\left||\boldsymbol{y}\right|| \\ \tau &= \frac{y_1 + a}{a} \\ \boldsymbol{r} &= \frac{\boldsymbol{y} + [a, 0, \cdots, 0]^T}{y_1 + a} \end{aligned} \tag{8}$$

このようにすると、$(I - \tau\boldsymbol{r}\boldsymbol{r}^T)\boldsymbol{y} = [\alpha, 0, \cdots, 0]^T$となる．

4.2 二重対角化の実装

ハウスホルダー変換による二重対角化では、一般的に高速化のためブロック化[4]という手法が用いられる．局所的なメモリアクセスの方が高速なアーキテクチャの場合、行列ベクトル積より行列積の方が高速になる．ブロック化は行列ベクトル積を減らし、行列積に置き換えることによって高速化する、というものである．

しかし、ブロック化は計算量を減らすものではなく、また、アルゴリズムは複雑化する．非常に大きな行列に対しては有効だが、中・小規模の行列に対しても有効かどうかが分からない．タスク並列を実現するためには、すべてのカーネル関数を新たに実装する必要があるため、アルゴリズムが複雑化すると実装コストが増大する．

以上のような理由により、まずブロック化しないアルゴリズムで実装した．

i番目の行・列の消去は、

$$\begin{aligned} A^{(i)} &= \left(I - \tau_v^{(i)}\boldsymbol{v}^{(i)}\boldsymbol{v}^{(i)^T}\right) A^{(i-1)} \left(I - \tau_u^{(i)}\boldsymbol{u}^{(i)}\boldsymbol{u}^{(i)^T}\right) \\ &= A^{(i-1)} - \boldsymbol{v}^{(i)}\boldsymbol{y}^{(i)^T} - \boldsymbol{x}^{(i)}\boldsymbol{u}^{(i)^T} \end{aligned} \tag{9}$$

と表せる．ただし、

$$\begin{aligned} \boldsymbol{y}^{(i)} &= \tau_v^{(i)} A^{(i-1)^T}\boldsymbol{v}^{(i)} \\ \boldsymbol{z}^{(i)} &= \tau_u^{(i)} A^{(i-1)}\boldsymbol{u}^{(i)} \\ \boldsymbol{x}^{(i)} &= \boldsymbol{z}^{(i)} - \tau_u^{(i)}\boldsymbol{v}^{(i)}\boldsymbol{y}^{(i)^{\mathrm{T}}}u^{(i)} \end{aligned} \tag{10}$$

である．

よって、アルゴリズム 1 のようになる.

アルゴリズム 1 二重対角化

1: for $i = 1$ to $n - 1$ do
2: $\quad \tau_v^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)}$を計算
3: $\quad$ 変換を適用した後の$A(i, i+1{:}n)$を計算
4: $\quad \tau_u^{(i)}, \boldsymbol{u}^{(i)}, \boldsymbol{x}^{(i)}$を計算
5: $\quad A(i+1{:}n, i+1, n)$を更新
6: end for

4.3 実装の詳細

1 つの行列の二重対角化を 1 タスクとして、これを並列化した.

リアルタイム変化点検知で使用するため、レイテンシを小さくしなければならない. GPU-CPU 間のメモリ転送はレイテンシを増加させるため、二重対角化はすべての演算を GPU で実装した. GPU-CPU 間のデータ転送は、行列Aの GPU への転送と、結果を取得するための二重対角行列の帯要素の CPU への転送のみである.

また、カーネル関数の切り替えにかかるオーバーヘッドは、行列サイズが小さくなると無視できなくなる。しかし、レジスタや共有メモリはカーネル関数ごとに静的に確保されるため、1 つのカーネル関数を大きくしすぎると、無駄が発生しやすくなる. つまり、カーネル関数切り替え回数を減らしつつ、レジスタや共有メモリの無駄が少なくなるようにするのが望ましい. そこで、なるべく 1 つのカーネル関数を大きくしたが、使用レジスタ数や共有メモリ量が大きく変化するポイントでカーネル関数を分けた. 結果、カーネル関数の呼び出し回数はアルゴリズム 1 の 1 ループあたり 3 回となった.

4.3.1 行列ベクトル積

行列は列メジャーでメモリ上に展開しているため、行列が転置でない場合、素直にスレッドごとに内積を求めることで、メモリアクセスが Coalesced アクセスになり、効率の良い行列ベクトル積となる.

しかし、行列が転置の場合、この手法ではメモリアクセスが Coalesced でなくなり、効率が悪くなる. 行列が転置の場合、次のような手法でメモリアクセスが Coalesced アクセスになるにした.

アルゴリズム 2 $\boldsymbol{m} \times \boldsymbol{n}$行列$\boldsymbol{A}$の転置とベクトル$\boldsymbol{v}$の積

1: $k = \frac{m}{16} + 1$ { 16 は Coalesced アクセスの単位}
2: for $i = 1$ to k do
3: $\quad A^T(1{:}n, 16i{:}16i+16)$を共有メモリにコピー
4: $\quad \boldsymbol{v}(16i{:}16i+16)$を共有メモリにコピー
5: $\quad A^T(1{:}n, 16i{:}16i+16)$ と $\boldsymbol{v}(16i{:}16i+16)$ の積を計算してそれまでの結果に加える
6: end for

実装にあたっては、Tesla C1060 を使用したため、この GPU が念頭に置かれているが、他の GPU に対してもパラメータを微調整するだけで最適化できる.

共有メモリへのアクセスでバンクコンフリクトが発生しないように、共有メモリは、$(n'+1) \times 16$ 行列（n'はnを 16 の倍数まで切り上げた数）と同じレイアウトで確保している.

この手法では、共有メモリの制限から行列サイズ 256 以上の場合 1 つのブロックだけでは計算できない. そのため、行列サイズが大きい場合、複数のブロックを使用して、計算することにした.

また、この手法では Partition camping[10]が高い頻度で発生し、速度が低下することが、後に判明した.

Partition camping とは、メモリアクセスが、ある Partition に集中した結果、メモリ読み書きの効率が低下することである. 共有メモリが 32bit 幅で 16 個（Fermi では 32 個）のバンクに分かれているのと同じように、グローバルメモリは、64byte 幅でいくつかの Partition に分かれている. ある Partition にアクセスが集中すると、そこがボトルネックとなってしまう.

行列ベクトル積に関しては、この手法だとプロファイルの結果、Tesla C1060 上では、最悪メモリバンド幅の 1/8 の速度しか出せないことがあった.

4.3.2 ノルム

ノルムの計算には和を取る演算がある. この演算の実装にあたっては、Mark Harris の Parallel Reduction[11]を参考にした.

4.4 ブロック化

ブロック化しない方法で実装したプログラムをプロファイルした結果、メモリバンド幅がボトルネックになっていた. そこで、ブロック化した方が、グローバルメモリの読み書きを減らせることから、高速化できると考え、ブロック化したものを実装した. アルゴリズムは[4]で提案されているものである. ブロックサイズは GPU の特性から 32 とした.

アルゴリズム 3 二重対角化 – ブロック化

1: $kMax \leftarrow \frac{n}{L}$ {L はブロックサイズ}
2: for $i = 1$ to $kMax$ do
3: $\quad t \leftarrow L(i-1) + 1$
4: $\quad A(t{:}n, t{:}n)$の転置を作成
5: $\quad \tau_v^{(t)}, \boldsymbol{v}^{(t)}, \boldsymbol{y}^{(t)}$を計算
6: $\quad$ 変換を適用した後の$A(t, t+1{:}n)$を計算
7: $\quad \tau_u^{(i)}, \boldsymbol{u}^{(i)}, \boldsymbol{x}^{(i)}$を計算
8: $\quad$ for $k = 2$ to L do
9: $\quad\quad t \leftarrow L(i-1) + k$
10: $\quad\quad$ 最新の$A(t{:}n, t), A(t, t+1{:}n)$を、$k-1$組のベクトル$\boldsymbol{v}, \boldsymbol{u}, \boldsymbol{x}, \boldsymbol{y}$を使って計算
11: $\quad\quad \tau_v^{(t)}, \boldsymbol{v}^{(t)}, \boldsymbol{y}^{(t)}$を計算
12: $\quad\quad$ 変換を適用した後の$A(t, t+1{:}n)$を計算
13: $\quad\quad \tau_u^{(i)}, \boldsymbol{u}^{(i)}, \boldsymbol{x}^{(i)}$を計算
14: $\quad$ end for
15: $\quad A(iL+1{:}n, iL+1{:}n)$を更新
16: end for

4.5 ブロック化した実装の詳細

ブロック化しない場合と同様に、すべての演算をGPU で実装した. GPU-CPU 間のデータ転送は、行列AのGPU への転送と、結果を取得するための二重対角行列の帯要素の CPU への転送のみである.

また、タスク数が少ない場合にも性能を引き出すため、アルゴリズム 3 の 10: $A(t{:}n,t)$と$A(t,t+1{:}n)$の計算を、条件分岐を使ったタスク並列で並列化し、並列度を高めた.

カーネル関数の呼び出し回数はアルゴリズム 3 の内側のループ、1 ループあたり 7 回に抑えた.

行列ベクトル積において、行列転置との積を計算する場合、どうしても、行列が転置でない場合より遅くなってしまう. そこで、ブロック化したアルゴリズムでは、ブロックごとのループの先頭で、行列Aの転置をメモリ上に展開しておくことで、高速化した.

4.5.1 行列ベクトル積

行列が転置でない場合は、ブロック化しない場合と同様の手法を使ったが、ループの展開（unroll）を使って、実行命令数を減らし、高速化を図った. ただ、行列ベクトル積はメモリの読み書きがボトルネックとなるので、実行命令数を減らしても、有意な速度の変化は見られなかった.

行列が転置の場合、Partition camping の発生を抑えるため、新たな手法を考案した. 実装の詳細は割愛するが、この手法により、行列が転置でない場合とほぼ同等か、少なくとも 8 割以上の速度が達成された.

4.5.2 行列転置

行列転置は、入力行列の転置を出力行列に書き込む操作である. 必要な計算は少ないので、メモリバンド幅がボトルネックなる. その場合、Partition camping を減らすことが重要となるが、[10]に行列転置において、Partition camping を減らす方法が示されている.

Partition camping を減らすには、ブロックの実行順序を制御する必要がある. CUDA では、以下の式で定義される 1 次元ブロック ID を使ってこの値の小さいブロックから SM に割り当てられる.

$$\mathrm{bid} = \mathrm{blockIdx.x} + \mathrm{gridDim.x} * \mathrm{blockIdx.y} \quad (11)$$

ただし、blockIdx と gridDim はそれぞれブロック ID とグリッドの次元を表す CUDA の記法であり、blockIdx.x、blockIdx.y はそれぞれブロック ID のx,yの値、gridDim.x はカーネル関数起動に使用されたブロックのx方向の数である. 列メジャーで行列が展開されている場合、複数のブロックが同じ行にアクセスすると、Partition camping が発生してしまう. 行列転置の場合、入力行列・出力行列双方で、同じ行への複数ブロック同時アクセスを防がなければならない. これを実現するには 1 次元ブロック ID の順番が行列の対角方向に振られるようにすればよい. そこで、[10]ではブロック ID から、実際にそのブロックで処理するブロックの座標を算出するのに以下の式を用いている.

$$\mathrm{by} = \mathrm{blockIdx.x}$$
$$\mathrm{bx} = (\mathrm{blockIdx.x} + \mathrm{blockIdx.y})\%\mathrm{gridDim.x}$$

bx,by は実際に処理するブロックの座標である.

我々はブロック ID のy次元をタスク並列のために使用しているので、この手法をそのままでは使用できない. そこで、bx,by を共に初期値 blockIdx.x から開始し、bx はカーネル関数内でインクリメントしながらループさせることにした.

タスク並列では、複数の行列が同時に計算されるため、タスク間の実行順序も考慮しなければならない. Partition camping は、メモリアクセスにおけるブロック間干渉と言うことができるが、式(11)からブロック ID のy次元が同じブロックでの干渉が大きく影響することが分かる. 我々はブロック ID のy次元をタスク ID として使用し、x次元を 1 タスク内の並列処理のために使用したが、この方法は、1 タスク内のブロック間干渉を減らすことで全体として干渉を減らせるという点で優れている. もし、x,yを逆にしてしまうと、複雑なタスク間干渉を考えなければならなくなる.

4.5.3 行列積

行列積は CUBLAS を使えば、十分な大きさを持つ行列の場合、Tesla C1060 上で 330GFLOPS の速度が出せるが、NVIDIA の Programming Guide[2]にある手法では、200GFLOPS 弱しか出せなかった.

Yi Yang らは、GPGPU の最適化コンパイラの研究[12]を行っている. 彼らは、GPU での最適化を意識していないコードからのコンパイルで、最適化によって CUBLAS と同等かそれ以上の速度で動作するコードの生成に成功している. 行列積の最適化後のコードが例として示されていたので、それを利用した. 彼らのコードを使えば、メモリアクセスを Programming Guide[2]にある手法の数分の 1 程度に減らすことができ、CUBLAS と同等の性能が出せる. 彼らの提案している最適化手法の 1 つにスレッドマージがある. 行列積の例として示されていたものは、32 スレッドをマージしたものであった. 我々は、32 スレッドをマージしたものと、16 スレッドをマージしたものの 2 つを実装・比較したが、16 スレッドをマージしたものの方が高速だったので、そちらを採用した.

5. 性能評価

評価環境として、CPU は AMD Phenom X4 9850 (4 コア, 2.5GHz)、メモリ 8GB、OS は CentOS 5.2、GPU は Tesla C1060 を用いた. LAPACK[17](Version 3.2.1)、ATLAS[18](Version 3.8.3)を用いて、1 コアで実行したものを CPU の速度とした.

前章の通り、1 つの行列の二重対角化を 1 タスクとする. CPU は各タスクの行列を LAPACK の二重対角関

数である SGEBRD で二重対角化し、全タスクの行列の二重対角化が終了するまでの時間を計測した．GPU は CPU-GPU 間のデータ転送も含めた時間を計測した．また、計測はそれぞれ 5 回実行し、その平均を取っている．行列は、ランダムな値で生成したが、CPU と GPU で計算する行列は同じ単精度浮動小数点型のデータを使用している．

図 2 はタスク数を 256 としたときの GPU,CPU の実行時間である．X 軸は行列サイズ、Y 軸は実行時間で対数目盛になっている．図中 Simple はブロック化していない GPU 実装、Block はブロック化した GPU 実装を示している．他の図も同様である．図 3 は GPU の CPU に対する高速化率である．X 軸は行列サイズ、Y 軸は GPU の CPU に対する高速化率である．タスク数は 64 の場合と、256 の場合を計測した．

ほとんどの場合、ブロック化した実装の方がブロック化しない場合より速いという結果だった．行列サイズ 320、タスク数 256 の場合、ブロック化した実装では 0.584 秒、CPU 1 コアでの実行時間は 10.06 秒と 17.22 倍の高速化を達成した．

計測した全ての場合において、行列サイズが 256 と 512 の場合に速度が低下している．これは、Partition camping（4.3.1 章）が原因と思われる．計測に使用した GPU デバイス Tesla C1060 の場合、Partition が 8 個なので行列サイズが 256 や 512 の場合に影響が出やすい．ブロック化しない実装では、Partition camping の影響が小さいように見えるが、これはすべての行列サイズで Partition camping の影響を受けて速度低下しているので、行列サイズが 256 と 512 の場合の速度低下が相対的に小さく見えるだけである．ブロック化した実装では Partition camping を減らす努力をしたので、ある程度改善したものの、行列サイズが 256 と 512 では Partition camping の発生を抑えることができず、速度低下が発生してしまった．

図 4 は行列サイズ 448 固定でタスク数を変えたときの、GPU の CPU に対する高速化率である．X 軸はタスク数の対数目盛になっている．GPU 実装は、1 タスクを 1 ブロックに割り当てて計算している部分が多い．Tesla C1060 は SM 数が 30 であるため、タスク数が 30 以下だと十分に性能を引き出すことができない．しかし、タスク数 32 以上では良好な結果が得られた．

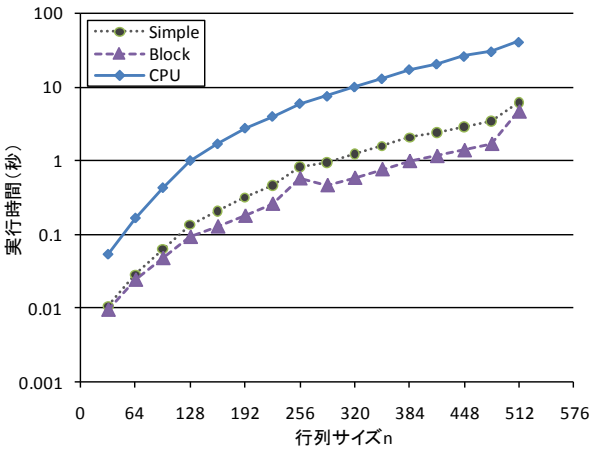

図 2 タスク数 256 の場合の GPU、CPU の二重対角化の実行時間（秒）

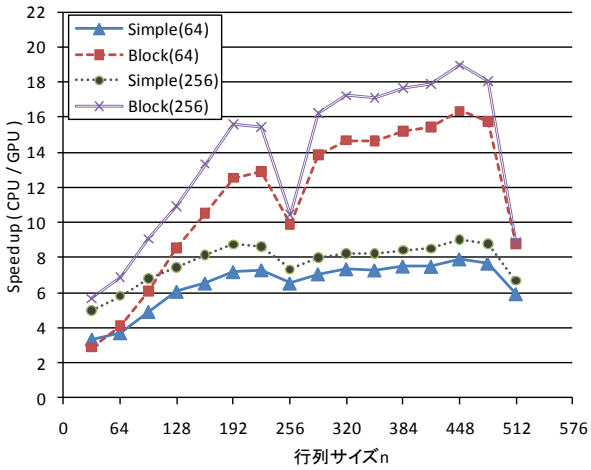

図 3 タスク数 64 の場合と 256 の場合の GPU の CPU に対する高速化率

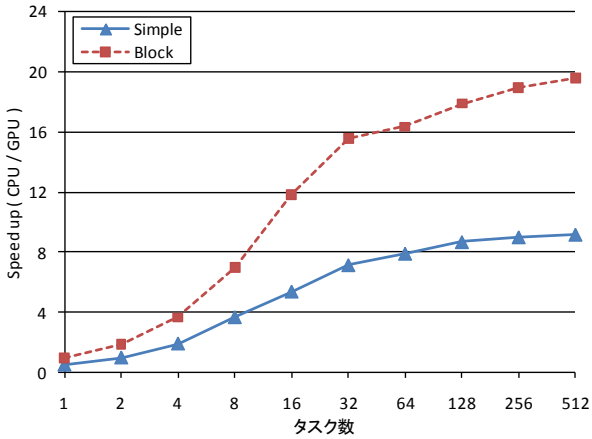

図 4 行列サイズ 448 固定でタスク数を変えたときの GPU の CPU に対する高速化率

6. 関連研究

6.1 SST 高速化に関する研究

特異スペクトル変換 SST (Singular Spectrum Transformation)は、その発案者である井手氏による高速化の研究[14]がなされている．彼の FELIX-SST という手法は次のようにして高速化している．SST アルゴリズム（2.2 章）では、過去と現在、2 つの行列の SVD を求めていた．過去側は左特異ベクトルを大きい方からr個使用し、現在側は最大左特異ベクトルのみ使用する．最大左特異ベクトルは、変分方程式の反復法による解法を使って効率よく求めることができる．過去側につ

いては、左特異ベクトルを求めず、Implicit kernel 近似を使って効率よく変化度スコアを求める．この手法によって、ウィンドウサイズ 250 において、130 倍の高速化に成功している．

6.2 GPU タスク並列に関する研究

GPU におけるタスク並列に関しては、Marisabel Guevara らによる研究[9]がある．コード修正なしで、タスク並列を実現する手法を提案している．しかし、彼らの手法では、タスク数に比例して、条件分岐や引数の数が増えるという問題がある．タスク数が少なければ有効だが、タスク数が多いとオーバーヘッドが大きくなる．変化点検知の場合、各タスクはそれほど大きくなく、多数の時系列データを処理する場合があるので、彼らの手法はそのままでは、変化点検知の高速化に適用できない．

6.3 GPU を使った行列計算の高速化に関する研究

SVD の高速化については､Sheetal Lahabar,らによる研究[3]や、深谷らによる研究[6]がある．しかし、彼らの手法は、非常に大きな行列に対しては有効だが、小さい行列を高速化することはできない．

GPU を使った行列計算の高速化に関する研究は非常に多い．しかし、ほとんどの研究は非常に大きな行列を対象としていて、小さい行列に対しての高速化に関する研究はない．

7. まとめと今後の展望

本論文では、大量のデータに対する変化点検知に GPU を適用させる場合の手法について述べた．これまでの GPGPU の研究は､非常に大きな行列などの大きな問題を対象としていたので、変化点検知に適用させることができない．そこで、GPU タスク並列による高速化手法を提案した．

変化点検知アルゴリズム SST では、SVD を使うが、SVD の計算のうち二重対角化を GPU タスク並列で実装した．性能評価の結果、行列サイズ 320、タスク数 256 のとき、CPU 1 コアに対して、17.22 倍高速化した．これにより、GPU タスク並列によって、変化点検知などの中・小規模の計算を繰り返すアプリケーションに GPU による高速化が適用できることを示した．

本論文では、SVD の 2 つのステップのうち二重対角化のみを実装したが、今後、SST 全体を GPU 実装し、GPU タスク並列による変化点検知の高速化を実証する必要がある．

謝辞：

本研究の一部は科学研究費補助金・挑戦的萌芽研究（課題番号:22650017) の助成によって行われた．

参 考 文 献

[1] 森田康介,鈴村豊太郎. 「データストリーム処理を用いた変化点検知の実装と GPU による性能最適化」. 電子情報通信学会 データ工学研究会, 2010 年 6 月

[2] NVIDIA CUDA Programming Guide, Version 3.0, 2010.

[3] Sheetal Lahabar, P. J. Narayanan. Singular value decomposition on GPU using CUDA. IEEE International Symposium on Parallel & Distributed Processing Symposium. 2009.

[4] Jaeyoung Choi, Jack J. Dongrra, David W. Walker. The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal and bidiagonal form. Numerical Algorithms, Volume 10, Number 2, 379-399. 1995.

[5] Tsuyoshi Ide, et al, Knowledge Discovery from Time-series Data using Nonlinear Transformations, The 4th Data Mining Workshop of JSSST 2004

[6] 深谷猛, 山本有作, 畝山多加志, 中村佳正. 正方行列向け特異値分解の CUDA による高速化. HPCS, 2009.

[7] CULA. http://www.culatools.com/.

[8] NVIDIA Corporation. NVIDIA CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf

[9] Marisabel Guevara, Chris Gregg, Kim Hazelwood, Kevin Skadron. Enabling Task Parallelism in the CUDA Scheduler. Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA). September 2009, pages 69-76.

[10] G. Ruetsch and P. Micikevicius. Optimizing Matrix Transpose in CUDA. NVIDIA, 2009.

[11] Mark Harris. Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf, 2008.

[12] Yi Yang, Ping Xiang, Jingfei Kong, Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. ACM SIGPLAN Conference on Programming Language Design and Implementation, 2010.

[13] N. Fujimoto. Faster matrix-vector multiplication on GeForce 8800GTX. IEEE International Parallel & Distributed Processing Symposium, 2008.

[14] T. Ide. Speeding up Change-Point Detection using Matrix Compression. IBIS, 2006.

[15] Daniel J. Abadi, etc., The Design of the Borealis Stream Processing Engine, 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05), Asilomar, CA, January 2005

[16] J. L. Wolf, N. Bansal, et al, SODA : An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems, Middleware 2008.

[17] LAPACK. http://www.netlib.org/lapack/.

[18] ATLAS. http://math-atlas.sourceforge.net/.