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

上野 晃司† 鈴村 豊太郎 †‡

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という新しい計算パラダイムである．変化点検知アルゴリズム SST(Singular Spectrum Transformation)を使った異常検知のストリーム処理では、一辺の長さが 500 以下の行列の演算が必要になる．GPU を使った高速化の既存手法では、小さい行列の計算を高速化することができない．そこで GPU タスク並列により解決した．GPU タスク並列で実装した特異値分解は、行列サイズ 300-500 で４コア CPU に対して４倍程度の高速化、IKA-SST は、ウィンドウサイズ 20-500 で CPU 1 コアに対して 20 倍以上の高速化を達成し、スケーラブルな異常検知機構が実現できることを示した．

Scalable Anomaly Detection on Data Stream Processing with GPU Task Parallelism

KOJI UENO † and TOYOTARO SUZUMURA ‡

Stream computing has emerged as a new processing paradigm that processes incoming data streams in a real-time fashion. On the other hand, many recent efforts have shown the suitability of GPGPU to high performance computing. By bringing two new trends, this paper proposes new innovative method called GPU task parallelism to optimize stream computing with GPGPUs. In this paper we implement the proposed approach over SVD (Singular Value Decomposition) and IKA-SST, a powerful algorithm of change point detection. The experimental results show that the proposed implementation of SVD provides performance gain by around 4 times order against quad-core and the proposed implantation of IKA-SST provides around 20 times order against single-core. This result validates the scalability of our proposed approach.

1. はじめに

近年、センサーデバイス技術やネットワーク技術の発達、IT システムの高度化に伴って、リアルタイムに大量のセンサーデータを取得できる時代が到来している．これに伴い、リアルタイムに入力データを処理するデータストリーム処理が、近年活発に研究され、産業界で利用され始めている．

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．

データストリーム処理のアプリケーションとしてセンサーデータのリアルタイム処理による異常・変化点検知がある．これは、工場の生産ラインの監視やサーバ群のエラー検知のため、センサーから取得したデータをデータストリーム処理によりリアルタイムに解析し、異常を検出するというものである．異常検知においては、多様な入力データを柔軟に扱える変化点検出アルゴリズムが求められ

† 東京工業大学
Tokyo Institute of Technology
‡ IBM 東京基礎研究所
IBM Research - Tokyo

る。例えば、センサーの中には常に一定の値を出し続けるものもあれば、正常な状態でも値が変化し続けるものもある。値の変化が、正常な変化なのか、異常な変化なのかを検出しなければならない場合があるので、単純なアルゴリズムでは対応できない.

SST[1] (Singular Spectrum Transformation)は比較的最近提案された手法で、少ないパラメータ設定で、多様な入力データに対応できるという特徴を持つ優れた変化点検出アルゴリズムである。最初に提案された SST のアルゴリズムは特異値分解(Singular Value Decomposition, SVD)を使って計算するが、特異値分解は計算量が大きく、それほど多くのデータを処理することはできない．そこで、高速に計算可能な近似アルゴリズム IKA-SST[2]が提案されている.

しかし、近似アルゴリズムを使っても計算可能なデータ量には限界がある. 異常検知の応用例である工場の生産ラインの管理などでは、センサー数が数万以上あることも考えられ、より多くのデータを低コストで処理することが求められている.

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（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り、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 を使って SST の性能最適化を行う．我々は先行研究[20]において、特異値分解で使われるハウスホルダー変換による二重対角化の GPU タスク並列による高速化を行い、GPU タスク並列の有効性を示した。本論文では、GPU 部分の並列化に加えて、CPU で計算する部分との連携方法も示し、また、アプリケーションとして、異常検知機構をストリーム処理系上に実装した. 本論文の貢献を以下に示す.

1. SST の計算処理を GPU により高速化する手法を提案する.
2. GPU の利用効率を高める GPU タスク並列の手法を提案する.
3. SST の GPU 処理をデータストリーム処理系上に実装し、実適用可能な状態で性能評価した.

以降、2 章では SST のアルゴリズム、3 章で既存手法の問題点を説明する．4 章でその解決方法である GPU タスク並列を説明し、5 章では SST の GPU タスク並列について説明する．6 章で性能評価、7 章で関連研究、8 章でまとめと今後の展望について説明する.

2. 変化点検知アルゴリズム SST

SST[1]の計算方法を簡単に説明すると次のようになる. SST は各時刻において、過去と現在の部分系列から、それぞれ特異値分解という行列演算により特徴抽出を行い、抽出された特徴ベクトル同士の差異を変化度スコアとする.

SST を計算するには特異値分解が必要である. しかし、特異値分解は非常に計算量の多い演算なので、これを正確に求めず、近似を使って高速化した IKA-SST[2]という手法が提案されている. IKA-SST は特異値分解を計算する単純なアルゴリズムに比べて数十倍～数百倍高速である. 本論文では、特異値分解を正確に計算する SST を IKA-SST と区別して SVD-SST と記述する.

2.1 SVD-SST のアルゴリズム

まず、SVD-SST のアルゴリズムを説明する.

時系列データ $\tau = \{x_t \mid t = (\text{実数全体})\}$ 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. 部分系列を w 本並べた行列として、

$$\mathbf{H}_1 = [s(t-w), \cdots, s(t-2), s(t-1)]$$
$$\mathbf{H}_2 = [s(t-w+\gamma), \cdots, s(t-1+\gamma)]$$

を定義する．ただし、γ は正整数である．$\mathbf{H}_1, \mathbf{H}_2$ はともに一辺の長さ w の正方行列である.

1. $\mathbf{H}_1$ の左特異ベクトルを特異値の大きい順に r $(< w)$ 個求め、$\mathbf{u}^{(1)}, \mathbf{u}^{(2)}, \cdots, \mathbf{u}^{(r)}$ とする.これらは過去側の特徴ベクトルである.
2. $\mathbf{H}_2$ の最大左特異ベクトル $\boldsymbol{\mu}_t$ を求める．$\boldsymbol{\mu}_t$ は現在の特徴ベクトルである.
3. 変化度スコア $z(t)$ は、

$$z(t) = 1 - \sum_{i=1}^{r} (\boldsymbol{\mu}^{\mathbf{T}} \mathbf{u}^{(\mathbf{i})})^2 \qquad (1)$$

で計算される.

ウィンドウサイズ w は検知したい変化の時間スケールによって選択する. パラメータ r は時系列データの性質によって 3~5 程度を選択する. また、変化度スコアを計算するには長さ $2w-1+\gamma$ の時系列データが必要になる.

2.2 IKA-SST のアルゴリズム

次に、IKA-SST のアルゴリズムを説明する．ε をある小さな定数とし、$\mathbf{a}$ を正規化されたランダムベクトル $\mathbf{a}_0$ で初期化しておく．

1. $\mathbf{H}_2^T\mathbf{H}_2$ の最大固有ベクトル $\boldsymbol{\mu}$（$=\mathbf{H}_2$ の最大左特異ベクトル）をべき乗法などの反復法によって求める．このとき初期ベクトルとして $\mathbf{a}$ を用いる．
2. $\mathbf{a}=\boldsymbol{\mu}+\varepsilon\mathbf{a}_0$ とおき、正規化する．$\mathbf{a}$ はフィードバックとして、次の $\boldsymbol{\mu}$ を求めるときに使用する．
3. $\mathbf{r}_0=\boldsymbol{\mu},\ \beta_0=1,\ \mathbf{q}_0=0,\ s=0$ と初期化して、以下の Lanczos 反復を実行し $\alpha_1,\cdots,\alpha_k$ と $\beta_1,\cdots,\beta_{k-1}$ を求める．

$$\begin{aligned}\mathbf{q}_{s+1}&=\frac{\mathbf{r}_s}{\beta_s}\\ s&\leftarrow s+1\\ \alpha_s&=\mathbf{q}_s^T\mathbf{H}_1\mathbf{q}_s\\ \mathbf{r}_s&=\mathbf{H}_1\mathbf{q}_s-\alpha_s\mathbf{q}_s-\beta_{s-1}\mathbf{q}_{s-1}\\ \beta_s&=\sqrt{\mathbf{r}_s^T\mathbf{r}_s}\end{aligned}\qquad(2)$$

4. $\{a_i\}$ を対角要素、$\{\beta_i\}$ を副対角要素とする対称三重対角行列 $\mathbf{T}_k$ の固有ベクトルを固有値の大きい順に r 個求め、$x^{(1)},\cdots,x^{(r)}$ とする．
5. 変化度スコア z は、

$$z=1-\sum_{i=1}^{r}[x^{(i)}]^2\qquad(3)$$

によって得られる．k は、以下のようにする．

$$k=\begin{cases}2r & r\in\text{even}\\ 2r-1 & r\in\text{odd}\end{cases}\qquad(4)$$

アルゴリズム中の Lanczos 反復にて $\mathbf{H}_1$ と $\mathbf{q}_s$ の積があるが、$\mathbf{H}_1$ は 1 本の時系列データからスライディングウィンドウによって取られた行列なので、行列を保持しなくても計算可能である．同様に $\boldsymbol{\mu}$ をべき乗法によって求める場合も、$\mathbf{H}_2$ とベクトルの積は、行列を保持しなくても計算可能である．

3. 既存手法の適用とその問題点

SST を高速化するにあたって、まず、特異値分解を容易に GPU に計算させることができるライブラリ CULA[3]を用いて、SVD-SST の高速化を試みた[5]．CULA は LAPACK の GPU 実装として開発されているライブラリである．図 1 は、特異値分解を CULA に計算させた場合と CPU で計算した場合の SVD-SST の変化度スコア 1 つあたりの計算時間である．実験環境などの詳細は[5]を参照していただきたい．結果、ウィンドウサイズ w が 450 以上では GPU の方が高速だが、400 以下では CPU の方が高速であった．SVD-SST の計算量は特異値分解に依存し、一辺の長さ w の行列の特異値分解が必要になるので、計算量は $O(w^3)$ である．w が大きいと計算量が大きくなり、計算に時間がかかってしまうが、w は 400 以下でも十分な場合が多い．また、より多くのセンサーデータに対して異常検知を行いたい場合、計算量の関係から w をあまり大きくすることはできない．しかし、CULA を利用した場合、$w\le 450$ では SVD-SST を高速化させることができず、これは実用上問題である．

w が小さい場合に CULA で高速化させることができないのは、小さい行列では十分な並列性が得られないためである．GPU を効率よく利用するには、数千個レベルの演算を並列に計算するアルゴリズムが求められる．しかし、w が小さい場合、数千レベルの並列性を達成することはできないので、単一スレッド性能の劣る GPU で計算させると、CPU より遅いという結果になってしまう．CULA では複数の行列を並列に計算することができないので、変化度スコアを複数同時に求めたい場合は、計算時間がリニアに増加するだけで、何の高速化も得られない．

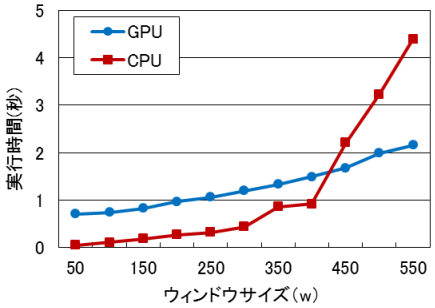

図 1 CULA を利用した場合の SVD-SST の計算時間

4. GPU タスク並列

前章で述べた問題を解決するため、本論文では、GPU タスク並列を提案する．図 2 にあるように、従来の計算手法では、同時に 1 つの行列しか計算しないため、小さい行列だと GPU コアを効率よく利用することができなかった．それに対して、GPU タスク並列は、小さい行列でも複数の行列を同時に計算することで、GPU コアを効率よく利用することができる．同時に 1 つの行列しか計算しない従来の計算手法は、データ並列性のみに依存した並列計算である．それに対して本論文の提案手法は、デー

タ並列性に加えて、複数の行列を同時に計算するというタスク並列性も使った、並列計算手法である．そのため、この提案手法を GPU タスク並列と呼ぶ．

この章では、既存の GPU 実装がある場合、その実装に GPU タスク並列を適用する方法について述べる．ここで述べている手法は、行列計算だけでなく、広範囲に応用可能な手法である．本論文では、NVIDIA の開発した GPGPU の開発環境である CUDA[4]を使用する．CUDA には、処理の連続性を保証する CUDA ストリーム[4]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．

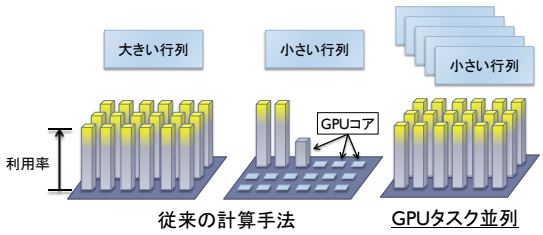

図 2 GPU タスク並列の効果

4.1 GPU プログラム

まず、CUDA を使った GPU プログラミングについて、簡単に説明する．GPU で計算されるコードは GPU カーネルと呼ばれる関数で記述する．CUDA では、C 言語を一部拡張した言語で GPU カーネルを記述することができる．GPU カーネルは CPU で実行されるホストスレッドから起動される．CUDA では、GPU カーネルの起動を関数呼び出しのように記述することができる．ただし、GPU カーネルから見ることができるメモリは、GPU メモリであり、CPU と GPU 間でのデータの通信は明示的に記述する必要がある．

ホストスレッドは CPU で動作する通常のプログラムと変わらない．ホストスレッドのプログラムコードには、GPU カーネルの呼び出しやデータ転送の順番、CPU で計算する部分などが記述される．GPU を使ったプログラムは、ホストスレッドのコード中に GPU カーネルが関数呼び出しのように入っている．

4.2 GPU 実行の並列化

n 個のタスクを並列に計算する方法を考えると、単純にホストスレッドを n 個立ち上げて並列化する方法が思いつくだろう．ホストスレッドを複数立ち上げて実行した場合、CPU で計算される部分は、OS のスケジューリングにより並列化される．また、GPU 部分は、コンカレントカーネルの機能により並列化される．ただし、現バージョンの CUDA では、コンカレントカーネルが適用されるための制約が厳しい．例えば、呼び出し順序などの規則を守る必要があることや最大 16 個までしか並列化されないなどの制約がある。このような制約があるためコンカレントカーネル機能を利用する方法は計算効率が悪い．

本論文では、1 つのカーネルで複数のタスクを計算する方法を採用する．この方法は、CUDA のスレッドブロックの独立性を利用したものである．CUDA では、基本的にすべての計算は SIMD で実行されるが、1 つの命令で処理されるデータ（スレッド）は CUDA で Warp と呼ばれる 32 スレッドの単位である．よって、Warp ごとに異なるコードを実行することが可能である．スレッドブロックはいくつかの Warp の集まりであり、スレッドブロックごとに異なるタスクを割り当て計算することが可能である．

図 3 はスレッドブロックごとにタスクを割り当てるカーネルコード例である．この例では、1 つのタスクを表す構造体を定義し、その構造体の配列をカーネルの引数に渡す．カーネルでは、CUDA のブロック ID を使って、タスクを識別し、複数のタスクを並列に処理する．

```
struct Task {
    int m, n;
    float* a;
};

/* taskArrayはTaskへのポインタの配列 */
__global__ void kernel_func(struct Task* taskArray[]) {
    Task* task = taskArray[blockIdx.y];
    int m = task ->m, n = task ->n;
    float* a = task ->a;
    /* 何かを実行する */
}
```

図 3 GPU タスク並列のカーネルコード例

4.3 CPU 部分の並列化

単一ホストスレッドから GPU カーネルを起動するということを実現するため、タスクの CPU 部分を計算する n 個の計算スレッドと、計算スレッドとは別に 1 つのホストスレッドを導入する．計算スレッドは、GPU カーネル呼び出しをホストスレッドに委譲する．ホストスレッドは複数の計算スレッドから受け取った GPU カーネル呼び出しをタスク並列で実行する．計算スレッドは、ホストスレッドが実行している GPU カーネルの終了を待ち、GPU カーネルが終了したら、CPU 部分の計算を再開する．このような方法で GPU タスク並列は実現可能である．

ただし、この方法ではタスクの数だけ、CPU 側のスレッドが必要になる．OS が提供するスレッド機能は、利用できるスレッド数に限りがあり、また、大量に生成するとパフォーマンスが低下する．SST

による異常検知では、GPU 1 つにつき数百～数千個のタスクの並列化が必要になるので、スレッド数の制限が問題となる．これは、ユーザレベルで実装された効率のよい軽量スレッドライブラリがあれば解決できる．しかし、現在、そのようなライブラリは普及していないので、本論文では、計算スレッドをイベント駆動型に書き換え、スレッドプールを使って計算させることで、この問題を解決した．この解決方法は、既存の GPU プログラムのコードに多くの変更が必要になるので、望ましいものではないが、将来、効率のよい軽量スレッドライブラリが出現すれば、コードの変更は必要なくなる．

5. SST の GPU タスク並列実装

2 章で紹介した 2 つのアルゴリズムを GPU タスク並列で実装した．この章では、2 つのアルゴリズムの実装の詳細を説明する．

5.1 SVD-SST の GPU タスク並列実装

SVD-SST は特異値分解を使って変化度を計算する．そこで、特異値分解を GPU タスク並列で実装した．

$m \times n$ 行列 A の SVD とは、

$$A = U \Sigma V^T \quad (5)$$

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．また、Σ の対角要素を特異値、U を左特異ベクトル、V を右特異ベクトルと呼ぶ．現在主流の特異値分解のアルゴリズムは、前半で二重対角化を行い、後半で特異値、特異ベクトルを計算するという 2 つのステップで計算するものである．二重対角化はハウスホルダー変換により効率よく計算することが可能である．特異値、特異ベクトルの計算に関しては QR 法[6]、分割統治法[7]、MR^3[8,9]、I-SVD[10]など、様々なアルゴリズムが提案されているが、本論文では QR 法を使用した．

ハウスホルダー変換による二重対角化と、QR 法の特異ベクトルの計算部分を GPU で計算する．QR 法の特異値を計算する部分に関しては、並列化ができないので、GPU で計算させると効率が悪い．そこで、この部分は CPU で計算させることにした．QR 法の計算では、まず、CPU で特異値をすべて計算し、次に GPU で特異ベクトルを計算する．このとき必要な計算データを GPU に転送している．図 4 は、CPU, GPU それぞれの計算部分とデータ転送を書いた図である．

1 つの行列に対する特異値分解を 1 タスクとして実装した．SVD-SST では、1 つの変化度スコアを計算するのに特異値分解を 2 回行うので、この 2 回の特異値分解も含めてタスク並列で計算することができる．つまり、n 個の変化度スコアを同時に計算したい場合、2n 個の行列の特異値分解をタスク並列で計算することができる．

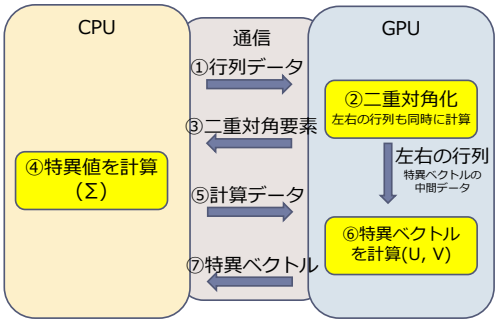

図 4 特異値分解の GPU タスク並列実装

5.2 IKA-SST の GPU タスク並列実装

IKA-SST は 2.2 章のアルゴリズム中 1～3 が計算量の大部分を占める．この部分では行列の圧縮処理を行っており、圧縮前の行列を扱わなければならないからである．アルゴリズム中の 4 で固有ベクトルを求めているが、ここでは一辺の長さ k (=5~10) という小さい行列の計算なので計算量は小さい．また、これほど小さいとタスク並列を使っても高速化は難しい．よって、アルゴリズム中 1～3 を GPU で計算し、固有値を求めて変化度スコアを計算する部分は CPU で行うことにする．また、$\mathbf{\mu}$ はべき乗法で求める．

$\mathbf{H}_1, \mathbf{H}_2$ は $w \times w$ の行列なので、GPU カーネル 1 タスクあたりのスレッド数は、行列サイズと同じ w とした．CUDA では、最大 1024 スレッド（一部 GPU は 512 スレッドまで）まで、1 つの CUDA ブロックとして扱うことができるので、w は最大 1024 まで対応できる．

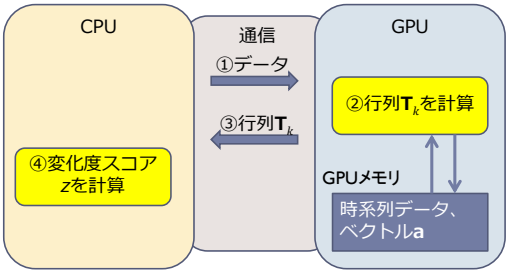

図 5 IKA-SST の GPU タスク並列実装

変化度スコアを計算するには長さ $2w-1+\gamma$ の時系列データが必要になる．時系列データ点が 1 つ追加されるたびに変化度スコアを計算する場合、長

さ$2w-1+\gamma-1$のデータ点は前回の計算で使用した値と同じなので、GPU メモリに記憶しておけば、転送するデータは 1 点のみでよい．また、長さwのベクトル$\mathbf{a}$はフィードバックとして利用するので、次の計算時に必要になる．そいで、これらをタスク用のデータとして GPU メモリに展開しておき、カーネルからアクセスして必要に応じて書き換えるという手法を用いた．

6. 性能評価

この章では、前章で実装した SVD-SST と IKA-SST の GPU タスク並列実装を CPU 実装と比較する．CPU 実装には、ATLAS 3.8.3[11]と LAPACK 3.3.0[12]を使用した．CPU は AMD Phenom X4 9850（4 コア, 2.5GHz）、メモリは 8GB、OS は CentOS 5.4 である．GPU は GeForce 8800GTS 512, Tesla C1060, GeForce GTX460 を使用した．CUDA のバージョンは 3.2 である．

6.1 特異値分解の性能評価

図 6,7,8 は特異値分解の GPU タスク並列実装の性能評価結果である．一辺の長さが 512 以下の行列に対する特異値分解の計算時間を測定した．CPU 実装は LAPACK の SGESVD を使用し、行列は乱数で生成、演算は単精度で行った．CPU 実装もタスクごとに並列化することが可能なので、4 スレッド使って CPU は 4 コア使用している．GPU はデータ転送の時間を含む．また、タスク数 n 個とは n 個の行列の特異値分解を計算することである．図 6 は、特異値分解の計算時間である．すべての特異値、右特異ベクトル、左特異ベクトルを求めている．図 7 は CPU 実装に対する GPU 実装の高速化率である．CPU 実装の性能が不安定なため、なめらかな曲線になっていない．全体的に、行列サイズが大きいほど GPU の高速化率は高く、行列サイズ 416、タスク数 256 で Tesla C1060 の場合 4.14 倍を達成した．図 8 は Tesla C1060 で動かした時の GPU タスク並列実装の実行時間内訳である．GPU 実装は、対角化の CPU 計算と GPU 計算を完全に分けていて、CPU 計算が全て終わってから GPU 計算を開始しているので、もし、CPU 計算と GPU 計算を並列化させることができれば、さらなる高速化が見込める．

6.2 SVD-SST の性能評価

特異値分解の GPU タスク並列実装を使って SVD-SST を実装し、さらに、IBM のデータストリーム処理系 System S[13]を使用して、分散処理可能なリアルタイム異常検知機構を実装した．System S は、SPADE と呼ばれる言語でストリームのデータフローとオペレータを記述する．各オペレータは、入力ストリームに対して、データの選別や変換などの処理を行う．SPADE には多くの組み込みオペレータが用意されているが、C++や Java で記述されたユーザ定義オペレータ(UDOP)を使用することもできる．そこで、SVD-SST を UDOP として実装した．

最大 4 ノードに分散処理させ、GPU を最大 16 台使用して性能評価を行った．図 9 は、その結果である．使用したマシンは 1 ノードにつき、GeForce 8800 GTS 512 を 4 台搭載している．GPU 以外は、6.1 章で使用したマシンと同じ環境である．また、SST のウィンドウサイズは、w=320 とした．パラメータγは過去側と現在側の 2 つの時系列データの間が 8 となるように$\gamma=2w-1+8$とした．γは計算量には影響しない．パラメータrは、3~5 でも広範囲に適用可能な精度だが、近似を使わずに特異値分解を求めているので、より高精度に計算できるよう大きめのr=12 とした．入力データは、波長変動のある正弦波データを使用した．図 9 の横軸は、使用した GPU と CPU コアの数である．例えば、横軸が N の場合、GPU は、N 台の GPU と N 個の CPU コアを使って計算している．CPU は、N 個の CPU コアを使って計算している．N=1~4 は、1 ノードのみ使用して計測した．図 9 から、16 GPU までほぼ線形に性能が向上しているのが分かる．16 GPU でのスループットは、305 スコア/秒であった．これは、もし、センサーデータに対する変化度スコアの計算を 5 秒おきに行うとすると、1525 センサーのデータを同時に処理できる性能である．

6.3 IKA-SST の性能評価

次に、IKA-SST の GPU タスク並列実装の性能を CPU 実装と比較する．CPU 実装は、LAPACK と ATLAS を使用して IKA-SST を計算する．IKA-SST では、$k\times k$ 対称三重対角行列の固有ベクトルを求める演算があるが、k は 10 以下と小さいので、LAPACK の複数ある固有値を求める関数のうち、simple driver である SSTEV を使用した．また、時系列データからスライディングウィンドウによって取られる行列$\mathbf{H}_1, \mathbf{H}_2$と、ベクトルとの積については、行列を保持しないで計算するアルゴリズムを、BLAS を用いて実装することが可能であるが、BLAS には最適な関数が用意されていないので、SIMD を使って最適化した専用ルーチンを実装した．性能評価では、BLAS を使った場合と、専用ルーチンを使用した場合の両方について性能を測定した．なお、CPU 実装は 1 スレッドで実行したので、CPU を 1 コアのみ使用した．IKA-SST のパラメータγは SVD-SST の場合と同じ$\gamma=2w-1+8$とした．パラメータrは時系列データによって 3~5

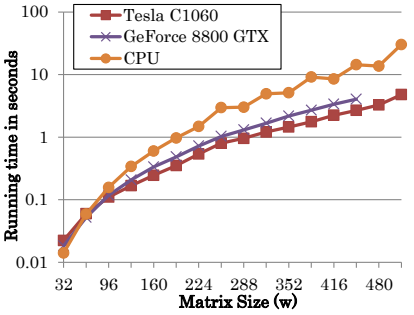

図 6　GPU、CPU の特異値分解の実行時間（秒）（タスク数 64）

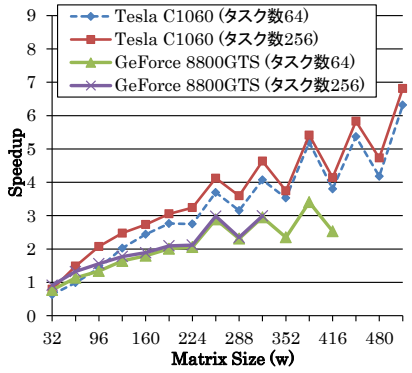

図 7　特異値分解の GPU の CPU に対する高速化率（タスク数 64, 256）

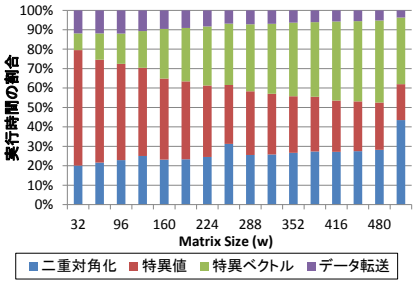

図 8　特異値分解の GPU タスク並列実装の実行時間内訳（Tesla C1060、タスク数 64）

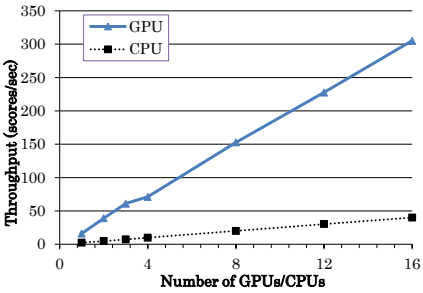

図 9　SVD-SST の System S 実装のスループット

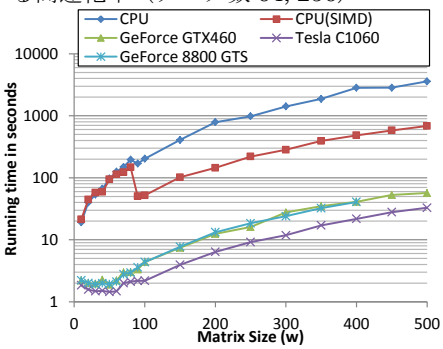

図 10　IKA-SST による変化度スコア 100 万個の計算時間

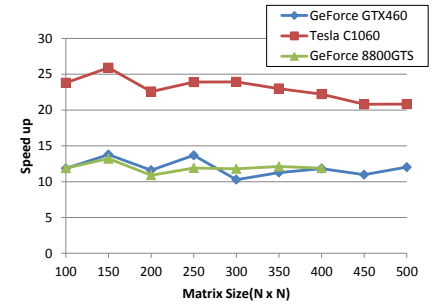

図 11　IKA-SST の GPU の CPU に対する高速化率（CPU は 1 コアで比較）

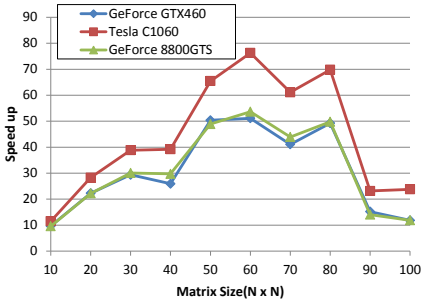

図 12　IKA-SST の GPU の CPU に対する高速化率（CPU は 1 コアで比較）

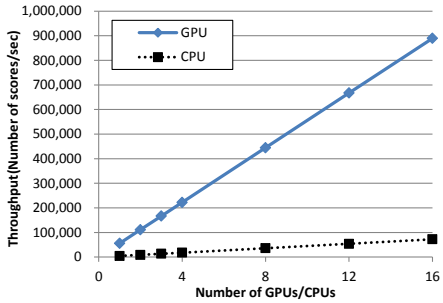

図 13　IKA-SST の System S 実装のスループット

を選択するが、性能評価では代表例として r =3 とした．r は Lanczos 反復の反復回数や行列 $\mathbf{T}_k$ のサイズに影響するが、3~5 と値の取れる範囲は小さいので、他の値を取っても性能の傾向は変わらないと思われる．また、計算精度は単精度、GPU 実装のタスク並列数は 500 とした．

図 10 は変化度スコア 100 万個の計算時間を測定した結果である．GeForce 8800 GTS は、レジスタ数の制限からウィンドウサイズ 400 までしか計測できない．GeForce GTX460 と GeForce 8800 GTS の計算時間は、ほぼ同じだった．それに対して、Tesla C1060 は 2 倍程度高速であった．CPU 実装は、ウィンドウサイズ 80 より 70 の方が、計算時間が長くなっているが、原因は ATLAS が長さ 70 以下のベクトルに対して最適化されていないからである可能性が高い．

図 11 と図 12 は、CPU に対する GPU の高速化率である．CPU 実装は SIMD を使って高速化した専用ルーチンを使用している．また、前述の通り、CPU 実装は 1 コアのみで実行している．ウィンドウサイズ 90 以上での高速化率は、Tesla C1060 で 21-26 倍、8800 GTS と GTX460 で 11-14 倍と、ウィンドウサイズによらずほぼ一定である．ウィンドウサイズ 90 以下では、高速化率は高くなり、Tesla C1060 では最高 76 倍($w=60$ のとき)となった．

IKA-SST も SVD-SST と同じように System S 上に実装し性能評価を行った．図 13 はその結果である．どの GPU 数に対しても GPU 実装は、同じ数の CPU コアを使用する CPU 実装に対して、12 倍以上の高速化を達成した．16GPU を用いた場合のスループットは、88.9 万スコア/秒であった．これは、毎秒 88.9 万タスクを処理していることにな

る．また、100ms 間隔でサンプルされるセンサーデータに対する異常検知を行った場合、8 万以上のセンサーデータをリアルタイム処理できる性能である．

7. 関連研究

GPGPU は近年活発に研究されている[14,15,16]が、データストリーム処理への適用に関する研究はない．GPU におけるタスク並列に関しては、Guevara[17]らによる研究がある．彼らは、コードの変更なしで GPU カーネルを並列化する手法を提案しているが、彼らの手法で並列化できるのは数個程度であり、数百レベルのタスクを並列実行することはできない．

特異値分解の GPU による高速化に関する研究は、[18]や[19]がある．どちらも非常に大きな行列に対しては高速化を達成しているが、小さい行列に対しては高速化できていない．GPU を使った行列計算の高速化に関する研究は、ほとんどが非常に大きな行列を対象としている．小さい行列の並列処理に関する研究はない．

8. まとめと今後の展望

変化点検知アルゴリズム SST を使った異常検知のストリーム処理では、一辺の長さが 500 以下の行列の計算が必要になる．GPU を使った高速化を考えた場合、既存手法では、これを高速化することができなかった．そこで GPU タスク並列により解決した．特異値分解は、行列サイズ 300-500 で 4 コア CPU に対して 4 倍程度の高速化、IKA-SST は、ウィンドウサイズ 20-500 で CPU 1 コアに対して 20 倍以上の高速化を達成した．

本論文で提案した GPU タスク並列の手法は、SST 以外にも適用可能な問題が多くあると思われる．ただし、既存実装の GPU タスク並列化には、カーネルやホストプログラムに変更が必要になるので、手動で行うにはコストがかかる．そこで、既存実装の GPU タスク並列化を自動で実行するフレームワークなどが、今後の課題としてあげられる．

参 考 文 献

[1] Tsuyoshi Ide, Keisuke Inoue, Knowledge Discovery from Heterogeneous Dynamic Systems using Change-Point Correlations, in Proc. 2005 SIAM International Conference on Data Mining (SDM 05), pp.571-576, Newport Beach, CA, USA, April 21-23, 2005.

[2] Tsuyoshi Ide, Koji Tsuda. Change-point detection using Krylov subspace learning. Proceedings of 2007 SIAM International Conference on Data Mining (SDM2007), pp.515-520, Minneapolis, Minnesota, USA, April, 2007.

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

[4] NVIDIA, CUDA C Programming Guide, Version 4.1, 2011.

[5] 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, Jun 2010

[6] G. J. F. Francis, The QR transformation, Parts I and II, Computer Journal, Vol.4, pp.265-271, 332-345, 1961-62.

[7] J. J. M. Cuppen, A divide and conquer method for the symmetric tridiagonal eigenproblem, Numerische Mathematik, Vol.36, pp.177–195, 1981.

[8] 山本有作, 密行列固有値解法の最近の発展(I), 日本応用数理学会論文誌, Vol.15, No.2, pp.181-208, 2005.

[9] I. S. Dhillon, A New O(n2) Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem, Ph.D. thesis, Computer Science Division, University of California, Berkeley, California, May, 1997.

[10] 高田雅美, 木村欣司, 岩崎雅史, 中村佳正, 高速特異値分解のためのライブラリ開発, 情報処理学会論文誌コンピューティングシステム, 47(SIG_7(ACS_14)), pp.81-90, 2006.

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

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

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

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

[15] 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.

[16] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ISCA, pp.451-460, Saint-Malo, France, June, 2010.

[17] 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), pp.69-76, September 2009.

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

[19] 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, Jan 2009.

[20] 上野晃司, 鈴村豊太郎. 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. インターネットコンファレンス 2010.