GPUにおける耐故障性を考慮した数値計算の電力性能

島田大地[†1,†2] 丸山直也[†1,†2] 額田彰[†1,†2]
遠藤敏夫[†1,†2] 松岡聡[†1,†2,†3]

近年、GPU は画像処理以外に科学技術計算への応用として多くの現場で用いられてきている。しかし、広く用いられてきている反面、GPU に対しての耐故障性は考えられてきていない。耐故障性が考えられていないと一時的な故障が生じた時にプログラムが正しく実行しなくなる可能性があり信頼性が低下する。これまで GPU は主に性能のみについて考えられてきており、信頼性向上手法については十分に議論されていない。そのため、GPU への信頼性の向上と電力性能のコストのトレードオフを検討する必要があるが、十分な評価がされていない。本研究では、CPU に対しての既存の耐過渡故障技術の一つである多重計算を GPU に実装し、評価した。行列積を対象に実装し、評価した結果、元の行列積の 1.88 倍のエネルギーで耐故障性を得ることができることが分かった。

Power-Performance Evaluation of Fault Tolerant Numerics on GPUs

Daichi Shimada,[†1,†2] Naoya Maruyama,[†1,†2]
Akira Nukada,[†1,†2] Toshio Endo[†1,†2]
and Satoshi Matsuoka[†1,†2,†3]

Recently, GPU is becoming a viable commodity for not only graphics processing but also scientific computation requiring enormous amount of calculation. However, fault tolerance has not been considered for the calculation in GPUs. Soft errors such as bit flips can produce wrong results unless any fault-tolerance techniques are employed. To establish the guideline toward performance-power efficient fault tolerant GPU computing, we examine redundant computation in matrix multiplication. We implemented eight different versions of redundant matrix multiplication and examined the impact of the power-performance of each version. Our redundant matrix multiplication implementation achieved higher reliability than that of CUDA SDK matrix multiplication with 1.88x energy increase.

1. はじめに

近年、GPU の発展に伴い GPU を画像処理以外への応用として、科学技術計算等に用いる GPGPU が注目されている[9),13)–15)]。CPU と GPU のピーク性能の差は広がっており、効率よく GPU を使うことによって CPU より高い成果を得ることができると期待されている。

一方、集積回路が微細化、高速化するにつれてビット反転などの過渡故障が発生しやすくなってきている[5)]。科学技術計算や金融機関、宇宙船などのミッションクリティカルな環境では高信頼性が求められ、このような一時的な故障の発生を検出、訂正する必要がある。

この問題に対して、既存の耐過渡故障技術として以下が挙げられる。

- サーバー向けプロセッサ・メモリにおける ECC・パリティなどによるデータの保護[1)]
- 行列演算向けチェックサムによる故障検知[7)]
- 複数スレッドによる多重計算[8)]

一般に、信頼性は電力性能とトレードオフの関係にある[4)]。GPU は従来画像処理等にのみ利用されていたため、信頼性を犠牲にし性能を重視する設計であった。そのため、上記のような耐過渡故障技術は実装されてなく、現在も GPU の信頼性は低いままである。

しかし、GPU が画像処理に限らないより一般的なアプリケーションで用いられるにつれて、その信頼性が必要とされる場合が増えてきている[2)]。例えば、Stone らは GPU により MRI 解析の 13 倍の高速化が可能であることを報告しているが[10)]、このような人命に関わるアプリケーションにおいては誤りの発生は許されない。従って上記のような既存手法やもしくは何らかの新たな手法により信頼性を向上させる必要がある。しかし、これまでのところ GPU では主に性能についてのみ着目されており、その信頼性向上手法については十分な評価・検討がされていない。　すなわち、これらの手法を用いる場合はそれによる信頼性向上効果と付随する性能・電力コストのトレードオフを詳細に検討する必要があるが、我々の知る限り GPU に適用した場合のコストの評価が十分にされていない。

本論文では、GPGPU での過渡故障検出時の性能向上の一歩として、既存の耐過渡故障技術の一つである多重計算を評価した。8 通りの手法を行列積に対して実装し、評価は実行

†1 東京工業大学
†2 科学技術振興機構
†3 国立情報学研究所

時間、電力、エネルギーの観点から行った。多重計算の手法として考えられる 8 通りの方法を実装して評価した結果、行列式の多重計算において単純な行列式と比較して 1.89 倍の実行時間の増加で実行可能であることを示した。また、エネルギーの観点から見ると元の行列式の 1.88 倍のエネルギーで耐故障性を加えた行列式が実行可能であることを示した。これらの結果から、より高効率な多重計算実現に向けた指針を獲得した。

2. 背　　景

この節では、GPU で起こりうる故障について議論する。そのために、CUDA GPU におけるプログラム実行について簡単に説明し、次に、その実行中に起こりうる故障について議論する。

CUDA GPU の実行時の流れを大まかに表すと次のようになる。

(1)　ホストからデバイスへデータ転送
(2)　グリッドとブロックのサイズの設定
(3)　カーネル関数の実行
(4)　デバイスからホストへデータ転送

CUDA では、CPU をホストと言い、GPU をデバイスと言う。1 では、ホストメモリからデバイスメモリへの転送が行われる。4 では逆にデバイスメモリからホストメモリへの転送が行われる。データ転送は異なるデバイス間で行うことができない。したがって、異なるデバイス間で通信を行うときは一度ホストを経由して異なるデバイスへデータを送る必要がある。また、3 のデバイス側で実行される関数をカーネル関数と言う。カーネル関数実行時に行われる複数の要素をスレッドと言い、スレッドはカーネル単位で管理される。一つのカーネルが一つのグリッドを持ち、グリッドはブロックを持ち、ブロックはスレッドを管理する。ブロックとスレッドには番号が割り振られ、プログラム内から番号を参照することができるので、その番号を元に各ブロックやスレッドの実行するデータを異なるものに変えることができる。ブロック数とスレッド数の設定は 2 で行われる。

この実行の流れにおいて、GPU 上の過渡故障は次の 3 箇所で起こりうる。

(1)　ホストからデバイスへデータ転送
(2)　カーネル関数の実行
(3)　デバイスからホストへデータ転送

ここで、1 と 2 では通信時のデータ化け等が起こりうる。3 では、次のようなことによって過渡故障が起こりうる[12)]。

- 正しい順序で命令が実行されない
- デバイスメモリや共有メモリ、レジスタでデータが正しく転送、記憶されない
- ALU が異なる値を示したり、回路が正常に機能しないことによる演算間違え

以下では、これらの故障を対象とする。

3. 提 案 手 法

GPU の過渡故障の起こりうる箇所はカーネル実行とデータ転送の二種類に分けられる。この章ではそれぞれの過渡故障に対する耐故障性を提案する。

3.1 カーネル実行の保護

カーネル実行時の過渡故障に対する耐故障性を実装する手法として多重計算を考える。多重計算は一度のプログラム実行で二回以上の計算をして、その結果を比較することによって計算が正しいかを確認する方法である。多重計算の方法として以下の手法がある。

連 続 計 算

連続計算は連続にデバイス上で二回計算し、時間的冗長性を得る手法である。この方法ではデバイス上で連続に計算をするため、実行時に元の計算よりも 2 倍のエネルギーがかかると予測できる。

並 列 計 算

同じ計算を同時に二回実行する手法である。この手法では、単一の GPU を用い 2 倍のスレッド数を起動し計算する方法と、二枚の GPU を用いて同時に計算を行う方法がある。後者では複数 GPU による空間的冗長性により耐故障性を実現し、実行時間の増加は最小化される。前者では、単一の GPU を用いるため時間的冗長性となるが、ハードウェア資源に 2 倍のスレッドを実行する余裕がある場合は、空間的冗長性となる。一般的にはアプリケーションは大規模メニーコアプロセッサである GPU のハードウェア資源を常に完全に用いるわけではないため、空き資源が存在しうる。そのためこの手法では空間的冗長性と時間的冗長性を両方用いた冗長化となる。

3.2 転送の保護

データ転送時への耐故障性を実装する手法としてデータを複製する手法がある。この手法は、データを二回送る方法や、データを転送するときに転送データを複製し、一度に元のデータと複製データの二つのデータを転送する方法がある。

3.3 結果の比較と復帰

冗長計算終了後に得られる二つの結果を比較することでエラー発生を検知する。違いが生

じた場合は計算を再実行することによりエラー発生より復帰する。

結果の比較には以下の 2 通りが可能である。

デバイス上で比較 計算終了後同一 GPU 上で結果の比較を行う。2 枚の GPU を用いる場合は、一方の GPU 上の結果をホストを介してもう一方の GPU へ転送した上で比較を行う。

ホスト上で比較 計算終了後すべての結果をホストへ転送し、ホスト上で比較を行う。

デバイス上での比較操作はそれ自体が計算であるため過渡故障の可能性があり、具体的には以下のシナリオが発生しうる。

計算が正常に終了した場合 比較操作の過渡故障によって誤ってエラーが発生したと判定され、誤検知 (false positive) が発生する。

計算にエラーが発生した場合 比較操作の過渡故障が計算エラーと異なるデータで発生する場合は、エラー発生数は誤って増加して報告されうるがエラーが発生したことは正しく検知される (true positive)。しかし、計算エラー発生箇所の結果の比較において再度エラーが発生する場合、実際にはエラーが発生したにも関わらず、誤って正常と判定されうる (false negative)。

前者において誤検知が生じた場合、我々の提案方式では計算を再実行するため、その時間的・電力的コストが生ずるが、計算の正当性には影響は与えない。しかし、後者のエラー検知の失敗が生じた場合は誤った結果を返すことになる。実際には、同一箇所に故障が発生する必要があるため、この発生確率は非常に小さいと言えるため、以降ではデバイス上での比較についてもその性能を評価する。

エラー発生からの復帰には同様に多重計算を繰り返すことで対応する。実際にはカーネルへの入力となるデータの退避や、GPU のステートの退避が必要になる。本論文では多重実行と結果の比較のみ評価を行う。復帰に関しては今後の課題である。

4. 実　　装

前章で述べた GPU 上での多重計算でどの手法が優れているかを確認するために実際にプログラムを実装した。数値計算は行列積を対象とし、逐次計算、データを再転送する逐次計算、一枚の GPU を用いた並列計算、二枚の GPU を用いた並列計算の 4 種類の多重計算手法を実装した。行列積の計算はそれぞれ CUDA の SDK 付属の行列積 (MatrixMul) を元にして実装した。比較方法はホストで比較する方法とデバイスで比較する方法をそれぞれの手法に実装したため、合計で 8 種類の手法を実装した。ホスト側で比較する方法では要素を一つずつ排他的論理和を用いて比較し、デバイス側で比較する方法では新しくカーネル関数を作成し、1 スレッド 1 要素を比較するようにした。

表 1 実装手法

名前	手法	比較場所	冗長性	検出可能エラー
並列デバイス	並列	デバイス	時間、空間	カーネル実行
並列ホスト	並列	ホスト	時間、空間	カーネル実行
連続デバイス	連続	デバイス	時間	カーネル実行
連続ホスト	連続	ホスト	時間	カーネル実行
再転デバイス	再転	デバイス	時間、空間	カーネル実行、データ転送
再転ホスト	再転	ホスト	時間、空間	カーネル実行、データ転送
複数デバイス	複数	デバイス	空間	カーネル実行、データ転送
複数ホスト	複数	ホスト	空間	カーネル実行、データ転送

表 2 実験環境

CPU	AMD Phenom(tm) 9850 Quad-Core(2.5GHz)
GPU	GeForce 8800 GTS 512 × 4
Memory	4GB
OS	FedoraCore8 (kernel 2.6.23)
GCC/CUDA	4.3.2/2.2

それぞれの手法についての特徴を表 1 に示す。また、それぞれの手法は以下の通りになっている。

並列 スレッド数を 2 倍にして計算する手法。

連続 カーネル関数を実行した後にもう一度カーネル関数を実行する手法。

再転 一度カーネル関数を実行した後に、入力データを再転送し、もう一度カーネル関数を呼び出す手法。

複数 2 枚の GPU を用いて同時に計算する手法。

5. 評　　価

5.1 実 験 環 境

多重計算を実行するマシンの構成と、電力計を制御するためのマシンと電力計の構成について述べる。

プログラム実行環境は表 2 のようになっている。

電力を測定するために、シナジェテック社製の電力計を使用した。また、この電力計の制御のためにマシンを一台用いた。これは電力計と接続しており、測定プログラムを実行する

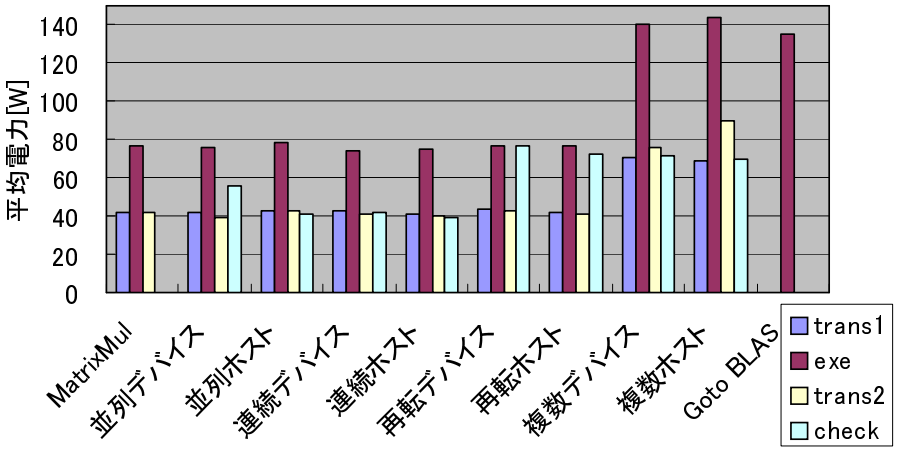

図 1 電力 (MatrixMul)

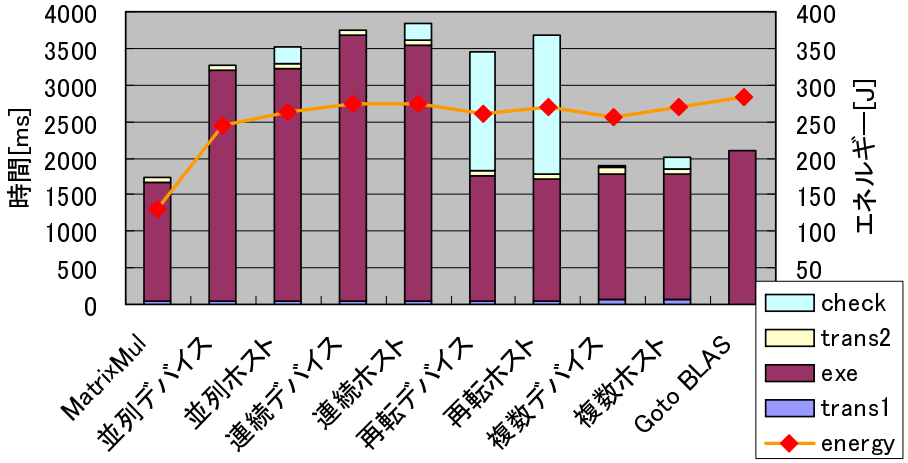

図 2 実行時間とエネルギー (MatrixMul)

ことによって電力計からデータを得ることができる。電力計は直流電流用センサ及び交流電流測定回路を組み合わせ、コンピュータ内部の主要コンポーネント毎の消費電力を測定することを目的として作成されている。GPU の電力は PCI-Express と電源からの二箇所から電力が供給されている。

電源から GPU に直接供給されているケーブルに電力センサを取り付けて計測した。また、PCI-Express からの電源には電流センサを取り付けることができないが、ライザーカードを用いた予備準備により、直接電源から供給されている電力と PCI-Express からの電力の比が 2:1 であることを用いた。電流センサはホール素子を応用した非接触方式となっており、このセンサには測定用の抵抗器などを挿入する必要が無く、分割式となっているので電源を切ってセンサに通さずに測定ができる。

5.2 MatrixMul

MatrixMul の多重計算に関する実験結果を図 1 と図 2 に示す。図 1 では、プログラム実行時の平均 GPU 電力を示しており、図 2 は実行時間と実行時にかかるエネルギーを示している。それぞれの図で、多重計算をしていない MatrixMul、提案手法、CPU 上の高速行列ライブラリである Goto BLAS(計算は一度のみ) を示している。また、それぞれの手法において計測区間を次の四つの区間に分割して計測した。

- trans1 : 入力データの転送
- exe : カーネル関数実行
- trans2 : 出力データの転送
- check : 多重計算の結果比較

ただし、データ再転送法だけは入力データを再転送するところから check として計測している。

5.2.1 実行時間に関する比較

実装した手法で最速であったものは二枚の GPU で行列積を計算した後にデバイスで比較する方法で、多重計算をしない行列計算 (以下 MatrixMul) を実行したときの 1.10 倍の時間で実行することができた。最も時間がかかったのは連続に計算をした後にホストで比較する手法で MatrixMul と比べて 2.21 倍の時間がかかった。提案手法をそれぞれ比較すると、最速の手法は最遅の手法の約 50 %の実行時間となっていた。

デバイスで比較する場合とホストで比較する場合を比べると、どの手法においてもデバイスで比較する場合が速かった。データ再転送法だけは前章で述べたように、データの再転送から check として計測しているため他の手法とは異なり check の値が高くなった。

Goto BLAS と比較してみると、複数デバイスは Goto BLAS の約 90 %の実行時間となっ

ていた。

5.2.2 電力に関する比較

GPU を二枚使う方法を除いて、どの場合も MatrixMul とほとんど変化が無かった。GPU を二枚使う方法では、他の手法に比べておよそ 2 倍の電力となった。電力の場合もデータ再転送法だけはデータの再転送から check として計測しているため、他の GPU を一枚だけ用いる手法と比較すると、check が異なる結果となった。

Goto BLAS の CPU 電力と GPU を用いた行列計算の総合電力を比較してみると、連続に計算した後にホストで比較する方法は Goto BLAS の 1.44 倍の電力がかかった。

5.2.3 エネルギーに関する比較

最もエネルギー効率が良かったのは並列に計算した後にデバイスで比較する方法だった。このとき、MatrixMul と比較して 1.88 倍のエネルギーで実行できた。最もエネルギー効率が悪かったのは連続に計算した後にデバイスで比較する方法であり、MatrixMul の 2.13 倍のエネルギーがかかった。

提案手法別に比較すると、最もエネルギー効率が良い手法は最悪の手法と比較すると約 88 %のエネルギーで実行できている。この結果は速度の観点から見たデータと同じような結果になった。これは、エネルギーの求め方が時間 × 電力であり、電力の値がどの場合もほとんど一致するためエネルギーと時間が比例関係に近いものになったからである。

また、Goto BLAS と比較してみると、並列に計算した後にデバイスで比較する方法は Goto BLAS の 86 %のエネルギーとなった。

6. 関 連 研 究

信頼性を高めることは GPGPU 以外にも考えられている。Plank ら[6] はディスクを用いないチェックポイントに基づいた手法を用いている。Cholesky,LU,QR,PCG に対して、提案手法を用い、2 種類のクラスタ上で実行した。結果として、適度なインターバルでのチェックポイントを取ることで低いオーバーヘッドになった。

1 章で述べたように GPU は画像処理向けに作られて来た。ゲームなどではエラーがあってもすぐに次の処理計算が実行され、表示されるのでエラーは隠されることになる。Dimitrov ら[3] は、GPGPU に対する耐故障性を実装する手法を示し、アプリケーションレベルで実装している。それぞれの手法では数値計算に冗長性を持たせ、多重計算を行っている。この研究で行われている多重計算は、以下の 3 通りで、それぞれ異なる分野で良く用いられる 6 つのアプリケーションに実装している。

(1) GPU 計算を 2 回実行する
(2) 使用されていない命令レベルでの並列計算
(3) スレッドレベルでの並列計算

1 では冗長性を加えない計算のおよそ半分のスループットとなった。2 と 3 はいくつかのケースで有用であったが、他の場合では、オーバーヘッドや複雑さの増加などが問題で有用でなくなっている。これを解決するために、ソフトウェア保護手法を適用する前にアプリケーションの性質やハードウェアプラットホームについて理解しておく必要があるが、解析をすることはアプリケーション開発者の重荷になりうる。

ソフトウェアでの解決法ではプログラマに CPU 側で比較するための冗長なコードを書かせることになったり、コンパイラに二回計算して結果を比較するようなコードを生成させなければならない。さらに、計算回数やメモリバンド幅のオーバーヘッドが 2 倍以上になることもある。Sheaffer ら[11] は、ハードウェアに冗長な部分を加えることにより耐故障性を実装している。実装はシェーダコアのレベルで行われていて、それぞれの入力要素が二回扱われるようにデータが複製される。このようにすることにより新しく論理回路を扱う必要が無く冗長な実行が実装できる。また、データの複製は変更する箇所を少なくするためにラスタライザからの出力を複製している。エラーを発見した場合は、再計算をするために計算の最初のステップまで入力を戻す必要がある。これは、シェーダモデルの統合によって行われている。この研究では、実際に実装してメモリ読み込み時のキャッシュ性能や電力を計測している。測定の結果、1.5 倍以下のオーバーヘッドで実行できることを示している。

7. 終 わ り に

7.1 ま と め

本研究では GPU 上で科学技術計算をするときの過渡故障について考察し、行列積に多重計算を実装、評価した。実装は 4 通りの多重計算を用いて行い、それぞれホストで比較するか、デバイスで比較するかの計 8 通りの手法を用いた。評価の結果、多重計算による信頼性を獲得したが、代わりに消費エネルギーがおよそ 2 倍になることが分かった。

実験の結果、行列積では並列に計算してデバイスで比較する手法が最もエネルギー効率のいい手法となった。このとき、多重計算をしない行列積と比較して 1.88 倍のエネルギーがかかった。また、それぞれの手法別に見ると連続に計算してデバイスで比較する方法は並列に計算してデバイスで比較する方法の 1.13 倍のエネルギーを必要とした。

7.2 今後の課題

高速フーリエ変換などの別のプログラムに対しても多重計算を実装し検証する。計算結果が異なることが検出できた場合に自動でエラーを直して実行を再開できるように実装し、実行した場合の性能も考慮する。自動にエラーを見つけて実行を再計算する方法の例として、もう一度多重計算をして間違いが無いことを確認して進む方法、計算を一度のみ実行して多数決法で修正する方法などがある。また、今回は単精度の冗長計算のみを対象としたが、倍精度の計算もあるため、倍精度冗長計算をした後にチェックをすることも検証していきたい。さらに、ランダムに fault injection をして、速度向上に対する検出率の低下を計測する手法についても研究していく。

謝　　辞

本研究の一部は科学技術振興機構戦略的創造研究推進事業『ULP-HPC: 次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング』、及び Microsoft Technical Computing Initiative “ HPC-GPGPU: Large-Scale Commodity Accelerated Clusters and its Application to Advanced Structural Proteomics ” によるものである。

参 考 文 献

1) Ando, H., Kan, R., Tosaka, Y., Takahisa, K. and Hatanaka, K.: Validation of hardware error recovery mechanisms for the SPARC64 V microprocessor, *Dependable Systems and Networks With FTCS and DCC, 2008.*, pp.62–69 (2008).
2) Blythe, D.: Rise of the Graphics Processors, *Proceedings of the IEEE*, Vol.95, No.5, pp.761–778 (2008).
3) Dimitrov, M., Manto, M., Zhou, H., Orlando and Orlando: Understanding Software Approaches for GPGPU Reliability, *ACM International Conference Proceeding Series*, Vol.383, pp.94–104 (2009).
4) Gonzalez, A., Mahlke, S., Mukherjee, S., Sendag, R., Chiou, D. and Yi, J.J.: Reliability: Fallacy or Reality?, *Micro, IEEE*, Vol.27, No.6, pp.36–45 (2007).
5) Mukherjee, S.S., Emer, J. and Reinhardt, S.K.: The soft error problem: an architectural perspective, *11th International Symposium on High-Performance Computer Architecture*, pp.243–247 (2005).
6) Plank, J.S., Kim, Y. and J.Dongarra, J.: Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations, *25th Annual International Symposium on Fault-Tolerant Computing*, pp.351–360 (1995).
7) Prata, P. and Silva, J.G.: Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations, *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pp.4–11 (1999).
8) Reinhardt, S.K. and Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading, *Proceedings of the 27th annual international symposium on Computer architecture*, Vol.28, No.2, pp.25–36 (2000).
9) Schatz, M., Trapnell, C., Delcher, A. and Varshney, A.: High-throughput sequence alignment using Graphics Processing Units, *BMC Bioinformatics*, Vol. 8, No. 1 (2007).
10) Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.-M.W., Liang, Z.-P. and Sutton, B.P.: Accelerating advanced mri reconstructions on gpus, *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pp.261–272 (2008).
11) W.Sheaffer, J., P.Luebke, D. and kevin Skadron: A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors, *Graphics Hardware*, pp.55–64 (2007).
12) 井上弘士：高信頼性マイクロプロセッサ・アーキテクチャ，日本信頼性学会誌，Vol.30, No.1, pp.27–35 (2008).
13) 尾形泰彦，遠藤敏夫，丸山直也，松岡　聡：性能モデルに基づく CPU 及び GPU を併用する効率的な FFT ライブラリ，情報処理学会論文誌コンピューティングシステム，Vol.1, No.1, pp.40–50 (2008).
14) 大久保宏樹，藤本典幸，荻原兼一：GPU 向け汎用計算機環境 CUDA を用いた k-means 法の高速化，*Symposium on Advanced Computing Systems and Infrastructures*, Vol.2008, No.5, pp.97–104 (2008).
15) 遠藤敏夫：東京工業大学 TSUBAME におけるアクセラレータ活用事例，情報処理，Vol.50, No.2, pp.100–106 (2009).