社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ソフトウェアECCによるGPUメモリの耐故障性の実現と評価

丸山　直也[†,††]　松岡　聡[†,†††,††]　尾形　康彦[†,††]　額田　彰[†,††]　遠藤　敏夫[†,††]

† 東京工業大学
††† 国立情報学研究所
†† 科学技術推進機構戦略的創造研究推進事業
E-mail: †{naoya,ogata,nukada,endo}@matsulab.is.titech.ac.jp, ††matsu@is.titech.ac.jp

あらまし　高い浮動小数点演算性能により、GPU を HPC 用途に用いる GPGPU が注目されている。しかし、GPU は本来グラフィックス用途に開発されてきたものであり、HPC 用途としては耐故障性に不十分な点が存在する。その一つとして、メモリ誤りの検出、訂正が挙げられる。現状の GPU には ECC を備えたものなく、一般的な HPC 計算ノードと比較して信頼性に劣る。我々は、GPU の信頼性向上のために、ソフトウェアによってメモリ誤りの検出、訂正を行う手法を提案する。本手法では、GPGPU アプリケーション中に ECC を計算、検査するコードを追加することで、グラフィックスメモリ中のビットフリップなどの誤りを検出、訂正する。提案手法を Nvidia による C 言語拡張 CUDA 向けにライブラリとして実装し、行列積と N 体問題アプリケーションに適用した。両アプリケーションを用いて、ECC 計算による性能オーバーヘッドを調査したところ、行列積で最大 300%程度, N 体問題で 15%程度のオーバーヘッドになることを確認し、N 体問題のようにメモリアクセス頻度に対して計算量の多いアプリケーションでは比較的小さなオーバーヘッドで実現可能であることを確認した。
キーワード　GPGPU、耐故障性、ECC

Naoya MARUYAMA[†,††], Satoshi MATSUOKA[†,†††,††], Yasuhiko OGATA[†,††], Akira NUKADA[†,††], and Toshio ENDO[†,††]

† Tokyo Institute of Technology
††† National Institute of Informatics
†† Japan Science and Technology Agency, CREST
E-mail: †{naoya,ogata,nukada,endo}@matsulab.is.titech.ac.jp, ††matsu@is.titech.ac.jp

Abstract　General-Purpose Processing on GPUs (GPGPUs) has rapidly been recoginized as a promissing HPC technology because of GPUs' much higher peak floating-point processing power. However, GPUs have originally been developed for graphics applications, such as 3D games, where reliability is not considered as an important issue as in HPC communities. One notable example is the lack of ECC in graphics memory systems. To improve the reliability of GPUs for HPC applications, we propose a software-based technique to generate and check ECC for graphics memory. Our library-based approache allows for CUDA-based GPGPU applications to be easily extended with ECC-based error checking with little manual intervention. To evaluate the applicability of our approach, we extended two CUDA applications with our ECC libarary: a matrix multiplication and an N-body problem. Our performance studies showed that while matrix multiplication can take up to 300% overhead, the N-body application only incurrs 15% of overhead. These results suggest that software-based ECC would be a promissing approach for computation-intensive applications such as N-body problems.
Key words　GPGPU, dependability, ECC

1. はじめに

Graphics Processing Unit (GPU) を HPC プラットフォームとして用いる GPGPU が注目されている [1]〜[4]。現在の主流 x86 系 CPU と比較して数十倍のピーク計算性能、メモリバンド幅を有し、特に高い並列性を備えたアプリケーションに有

効である。例えば、額田らは Nvidia GeForce 8800 GTX を用いて 3D FFT において約 80GFLOPS の性能を出せることを示しており、CPU に比較して数倍の高速化を達成している [2]。また、GPGPU にはコンシューマ用途として市販されているコモディティな GPU を用いることができ、ClearSpeed [5] などの専用アクセラレータとは異なり、コストパフォーマンスに優れているという特徴を持つ。

上記特徴により HPC 分野において注目されつつある GPGPU であるが、本来グラフィックス用途として設計されてきた GPU を HPC 基盤として用いるには解決されるべき問題が存在する。その 1 つにメモリシステムの耐故障性が挙げられる。通常の HPC 向け計算ノードではメモリシステムの耐故障性のために Error Correcting Code (ECC) を備えたメモリが用いられる。しかし、Nvidia や AMD による現状のグラフィックスカード上のメモリには装備されていない。この原因の一つとして、グラフィックス用途ではビットフリップなどのメモリエラーは深刻な問題として認識されていなかったことが推測できる。実際、ビットフリップが発生したとしてもユーザの画面上のピクセルが一瞬間違った色になるだけであり、3D ゲームなどの従来の GPU の応用では許容範囲であった。しかし、HPC に限らずアプリケーション一般的には 1 ビットのエラーでも許容できない場合が多い。また、一般的にゲームなどのグラフィックス用途に比べて HPC の方がアプリケーションの実行時間が長い。従って、アプリケーション 1 回の実行中に発生した単一のビットフリップが長時間に渡って広範囲な影響を及ぼし、最終的な結果を大きく変えてしまう可能性もある。特にコンシューマ市場向けコモディティグラフィックスカードを用いることで価格性能比の大幅な向上が期待されているが、それらの HPC 実行基盤としての信頼性は明らかではない。

我々は、GPU におけるメモリシステムの耐故障性実現のためにソフトウェアによる ECC を提案する。本手法は GPU からグラフィックスカード上のメモリへのアクセス時に ECC を計算し、1 ビットエラーの修正、2 ビットエラーの検知を実現する。具体的には、まずグラフィックメモリの確保時に別途 ECC 用の領域を確保する。被保護領域 4 バイトもしくは 8 バイト毎に 1 バイトの領域を ECC 用として用いる。被保護領域への書き込み時にはその直後に ECC を計算し、別途確保した ECC 領域へ保存する。ECC としては通常の DRAM 向け ECC と同様にハミング符号を用いる。メモリの読み込み時にも同様に ECC を計算し、それを対応する ECC 領域より読み込んだ ECC と比較し、1 ビットエラーの訂正もしくは 2 ビットエラーの検知を実現する。これにより、ビットフリップなどのソフトエラーに対する耐故障性を実現し、GPGPU においても通常の HPC 向けメモリシステムと同様の信頼性を実現する。

本提案手法を評価するために、ECC による保護を容易にアプリケーションに導入可能にするライブラリを実現した。本ライブラリは Nvidia 社による GPGPU 向け C 言語拡張である CUDA 向けに実装されており [6]、既存 CUDA アプリケーションのソースコードに API 呼び出しを加えることで ECC の計算、エラーの訂正、検知を行う。同ライブラリを行列積と N 体問題に適用し、ECC 計算による性能オーバーヘッドの評価を行った。その結果、行列積で最大 300%程度, N 体問題で 15%程度のオーバーヘッドになることを確認し、N 体問題のようにメモリアクセス頻度に対して計算量の多いアプリケーションでは比較的小さなオーバーヘッドで実現可能であることを確認した。

2. ECC 計算アルゴリズム

通常の DRAM に採用されている誤り検出機構では、SEC-DED ハミング符号を 64 ビット毎に ECC としてデータに付与する。SEC-DED ハミング符号は 1 ビットの誤り訂正かつ 2 ビットの誤りを検出可能な符号である [7]。同符号化方式では、k ビットの被保護データを k 次元ベクトル $\boldsymbol{d} = \{d_0, \ldots, d_{k-1}\}$, $d_i \in 0, 1$ として表す。さらに、データベクトル $\boldsymbol{d}$ に r ビットの符号 $\boldsymbol{c} = \{c_1, \ldots, c_{r-1}\}$ を結合した長さ n のベクトルを $\boldsymbol{v} = \{d_0, \ldots, d_{k-1}, c_0, \ldots, c_{r-1}\}$ と表し、全体として (n, k) SEC-DED ハミング符号と呼ぶ。SEC-DED ハミング符号では、k ビットのデータに対して、最低 $\log_2 n + 1$ ビットの符号が必要であることが知られている [7]。例えば、データ長 64 ビットの場合、最低 8 ビットの符号が必要であり、$(72, 64)$ と表現される。

SEC-DED ハミング符号は、単一誤りの訂正が可能な SEC ハミング符号にパリティビットを付加し、2 ビットの誤り検知を可能にしたものである。以下、SEC ハミング符号について概略を述べる。SEC-DED 符号化については文献 [7] を参照されたい。データベクトル $\boldsymbol{d}$ から符号を含んだベクトル $\boldsymbol{v}$ の生成とその符号との検査には、符号生成行列 $\boldsymbol{G}$ と誤り検査行列 $\boldsymbol{H}$ を用いる。 両行列は、以下の制約を満たすものとして定義される。まず、さ k のデータベクトル $\boldsymbol{d}$ に対する長さ r の符号生成は、$k \times n$ の生成行列 $\boldsymbol{G}$ を用いて以下の通り計算される。

$$\boldsymbol{v} = \boldsymbol{d} \cdot \boldsymbol{G} = \begin{pmatrix} \boldsymbol{I_k} & \boldsymbol{G_r} \end{pmatrix} \tag{1}$$

ここで、$\boldsymbol{I_k}$ は $k \times k$ の単位行列であり、$\boldsymbol{G_r}$ は $k \times r$ の行列であり、各要素は 0 または 1 である。ただし、$\boldsymbol{d}$ と $\boldsymbol{G}$ の行列積では各要素の積の和として排他的論理和を用いる。検査行列 $\boldsymbol{H}$ は、$\boldsymbol{G}$ を用いて以下の通り $r \times n$ の行列として定義される。

$$\boldsymbol{G} \cdot \boldsymbol{H}^T = \boldsymbol{G} \cdot \begin{pmatrix} \boldsymbol{H_r} & \boldsymbol{I_r} \end{pmatrix}^T = \begin{pmatrix} \boldsymbol{h}_0 & \cdots & \boldsymbol{h}_{n-1} \end{pmatrix} = \boldsymbol{0} \tag{2}$$

ここで、$\boldsymbol{h}_i$ は、長さ r のベクトルであり、互いに線形独立なものとして定義される。以上の制約を満たす行列 $\boldsymbol{G}$、$\boldsymbol{H}$ を求め、(n, k) ハミング符号を構成する。例えば、$(7, 4)$ の場合の検査行列 $\boldsymbol{H}_{(7,4)}$ として以下を用いることができる。

$$\boldsymbol{H}_{(7,4)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

$\boldsymbol{G}$ によって生成された $\boldsymbol{v}$ の検査は、検査行列 $\boldsymbol{H}$ を用いてシンドロームベクトル $\boldsymbol{S}$ を求めることで行う。

$$\boldsymbol{S} = \{s_0, \cdots, s_{k-1}\} = \boldsymbol{v} \cdot \boldsymbol{H}^T \tag{3}$$

ハミング符号では、誤りが発生しない限り常に s_i は 0 である。

逆に、i 番目のビットのみが反転した場合は、シンドロームベクトルは h_i と等しくなる。従って誤り検査では、シンドロームベクトルを計算し同ベクトルが 0 ベクトルの場合は誤り無しと判定する。そうでない場合は検査行列においてシンドロームベクトルと等しくなる列ベクトル h_i を求め、i 番目のビットを反転して誤りを訂正する。

3. CUDA 向け実装手法

CUDA とは NVIDIA が提唱する GPGPU 向け C 言語拡張である。本節ではまず CUDA とそれが動作する GPU アーキテクチャの概要、特にメモリモデルについて説明する。次に同メモリシステムの ECC による保護の実現について説明する。

3.1 CUDA の概要

CUDA は C 言語にスレッドレベルの並列実行機能を導入する言語拡張であり、アプリケーションの並列実行可能部分を GPU にオフロードさせることが可能である。ユーザは CPU 上で実行されるホストプログラムと、GPU 上で並列実行されるカーネル関数を作成する。ホストプログラムはカーネル関数の呼び出しを含むプログラムであり、通常の C 言語に変換され、最終的に GCC などの通常の CPU 向けコンパイラによって実行ファイルへと変換される。カーネル関数は CUDA が提供するコンパイラにて CUBIN 形式のバイナリへ変換される。実行時にはホストプログラムから同 CUBIN バイナリをその引数と共に GPU に移送し、GPU 上でマルチスレッドで並列実行される。GPU 上でのスレッドは、スレッドブロックという単位にまとめられ、さらに全スレッドブロックをまとめたものがグリッドと呼ばれる。GPU 上でのスレッドのスケジューリングは、ワープと呼ばれるサイズのスレッド数を単位として行われ、同一ワープ内のスレッドは同一命令を実行する。

CUDA をサポートする GPU アーキテクチャとしては NVIDIA G80 と、それをベースにした G92, G200 などがある。以下では、G92 による GPU の一つである GeForce 8800 GTS (以下 8800 GTS) に基づいて概略を説明する (他のアーキテクチャでもほぼ同様である)。8800 GTS は、Streaming Processor (SP) と呼ばれるプロセッサからなり、それを 8 個ずつまとめた Multi Processor (MP) を合計 16 個、GPU 全体で 128 個の SP を持つ。各 SP は 1.625GHz で動作し、合計で 416 GFLOPS のピーク性能を持つ。8800 GTS では 512MB のグラフィックスメモリを搭載し、256 ビット幅、62GB/s のインターフェイスで GPU チップと接続されている。

CUDA におけるメモリシステムは、オフチップメモリであるグラフィックスメモリとオンチップメモリであるレジスタ、ローカルメモリ、共有メモリから構成される。 グラフィックスメモリは GPU 上の全 SP から共有され、領域の確保の仕方によりグローバルメモリ、コンスタントメモリ、テクスチャメモリとして利用される。レジスタ、ローカルメモリは各 SP に固有な領域であり、共有メモリは MP 内の SP により共有され、それぞれ SP より高速にアクセス可能である。各スレッドは一つの SP 上で動作し、各スレッドブロックが単一の MP にスケジュールされるため、スレッドブロック内のスレッドからは共有メモリを介してデータの高速な共有が可能である。

CUDA アプリケーションからは上記各種メモリはその領域確保手法は異なるが、アクセス方法は通常の CPU 向けプログラムと同様に C 言語の文法にそって行われる。例えば、グラフィックスメモリはグローバルメモリとしてアプリケーションから利用可能であり、CUDA が提供するメモリ管理 API を用いて領域を確保、開放を行う。共有メモリの確保は変数の宣言にキーワードを追加することで実現される。

3.2 CUDA アプリケーションへの ECC 保護の導入

上記メモリシステムにおいて、ECC を用いてメモリエラーからの保護を実現する。今回はオフチップメモリへのアクセスとなるグローバルメモリへのアクセスを保護する。オフチップメモリへのアクセスはグローバルメモリ以外にテクスチャメモリとコンスタントメモリとして利用する場合がある。またオンチップ領域のメモリも今回は対応しない。これらの保護は今後の課題である。

グローバルメモリへのアクセス時に ECC を計算し、2. 節で述べた 1 ビットの誤りの訂正と 2 ビットの誤り検知を行う。3. 1 節で述べたように、CUDA ではグローバルメモリとして確保された領域へのアクセスは常にグローバルメモリへのアクセスとなる。従って、グローバルメモリからの読み込み時にはそのデータの読み込み後に ECC を計算し、誤りの検出訂正を行う。書き込み時には書き込み後に書き込むデータの ECC を計算し、ECC 用の領域に保存する。これらの ECC はグローバルメモリ中に別途確保した領域に保存する。

上記 ECC 計算のアプリケーションへの付加手法として、ソースコードや中間コードの解析による自動変換手法と、ソースコードに明示的に ECC 計算を追加する人手による手法が挙げられる。今回はプロトタイプとして開発の簡便さを優先し、既存アプリケーションへ人手により ECC 計算を追加する手法を選択した。既存アプリケーションの変更箇所を最小化するために、ECC の計算と検査を CUDA 向けライブラリとして実装した。

CUDA ではグローバルメモリへ 4 バイト、8 バイト、16 バイトの単位でアクセスするため、SEC-DED ハミング符号を $k = 32, 64$ について設計し、データサイズに基づいて適切な符号を計算し、誤りの検査を実行する。16 バイトのデータについては、上位 8 バイトの符号と下位 8 バイトの符号で誤り保護を実現する。$k = 32$ の場合は、符号として 7 ビット必要であり、$k = 64$ の場合は 8 ビット必要であるが、両者とも今回の実装では 8 ビットの領域を使用する。実際に 32 ビット ECC と 64 ビットの計算を実装したところ、それぞれ 34 回、144 回の 32 ビットのビット演算が主な演算となった。

以下、具体的な適用例として行列積と N 体問題への適用例を紹介する。

3.3 行列積への適用

実装した ECC ライブラリの CUDA SDK に付属するサンプル行列積プログラムへの適用手法について述べる。

サンプルプログラムでは、行列積 $\boldsymbol{A} \cdot \boldsymbol{B} = \boldsymbol{C}$ において、$c_{i,j}$ を 1 スレッドで計算することで並列化する。MP 内のスレッ

ドはスレッドブロックとして協調して動作し、それぞれの $c_{i,j}$ の計算に必要な $a_{i,\cdot}$, $b_{\cdot,j}$ を共有メモリにロードして計算する。カーネル関数のメインループでは、配列 $\boldsymbol{A}$、$\boldsymbol{B}$ よりそれぞれ float 型の値を 1 つずつロードし、$2B$ 回の単精度浮動小数点計算を行う。ここで B はブロッキングパラメータであり、スレッドブロック内のスレッド数の平方根に等しい。最後に、ループの終了後に配列 $\boldsymbol{C}$ に単一の float を書き込む。図 A·1 に簡略化したコードを示す。

同カーネル関数における ECC の計算は、ループ内の $\boldsymbol{A}$、$\boldsymbol{B}$ からの読み込み時、さらにループ終了後の $\boldsymbol{C}$ への書き込み時に行えばよい。従って、ECC の計算による最も大きな影響はカーネル内の 2 回の ECC 検査である。3.2 節で述べたように、1 回の float 型の ECC の検査には 1 バイトのグローバルメモリからの ECC のロードと、34 回の整数演算が必要であり、ループ内で合計 68 回の演算と 2 回のロード命令を実行する。従って、ループ内の計算量を P と、ECC 付与後の計算量を P_{ecc} とすると、

$$P = 2Bt_f + 2t_m, P_{\mathrm{ecc}} = P + 68t_i + 2t_m$$

である。ここで t_f は浮動小数点演算の実行時間、t_i は 32 ビットのビット演算時間、t_m は 4 バイトのメモリアクセス時間を表す。従って、ECC 計算による実行時間の増加割合 E は、以下の通り表される。

$$E = \frac{P_{\mathrm{ecc}}}{P} = 1 + \frac{32t_i + t_m}{Bt_f + t_m}$$

すなわち、ブロッキングパラメータ B が大きくなるにつれてオーバーヘッドは小さくなる。B はスレッドブロック内のスレッド数の平方根であるため、各 MP で同時実行可能なスレッド数を増やすことでオーバーヘッドを隠蔽可能である。しかし、実際にはレジスタ数、共有メモリなどの制限のため同時実行可能なスレッド数は制限されるため、任意にオーバーヘッドを隠蔽することはできない。定量的な評価を 4 節で示す。

3.4 N 体問題への適用

次に、同じく CUDA SDK に付属するサンプルプログラムの一つである N 体問題への我々のプロトタイプ ECC ライブラリを適用する手法を述べる。同プログラムの詳細については文献 [8] を参照されたい。

同アプリケーションでは、N 体問題の加速度計算部をマルチスレッドで並列実行する。全物体を各スレッドで均等に分割し、それぞれの物体について他のすべての物体との重力の計算、位置、速度の更新を行う。本プログラムでは全点間の重力を計算し、カットオフなどによる計算量削減は行われない。

リスト 1 にカーネルのアルゴリズムを示す。同リスト内で、N は全物体数であり、B はスレッドブロック内のスレッド数である。sub body set が 1 スレッドが担当する物体の集合であり、各担当物体についてその位置をグローバルメモリからロードする。次にその物体と他のすべての物体との距離を計算し、加速度を求める。この際、他の物体の位置をグローバルメモリからロードし、最初にロードした位置との差分より距離を求める。ただし、各スレッドは全物体の位置をそれぞれがロードするのではなく、同一 MP 内でロードした位置を共有し、スレッドあたりのグローバルメモリからの読み込み回数を削減する。すなわち、スレッドブロック内のスレッド数が B の場合、グローバルメモリからの読み込み回数が $1/B$ に削減される。

リスト 1 N 体問題カーネル

```
1: for all b in sub body set do
2:   p = load position of b from global memory
3:   for all i in N/B do
4:     q = load position of body d at index i + tid from global
       memory to shared memory
5:     for all e in B bodies loaded in shared memory do
6:       acc += compute acceleration of b by e
7:     end for
8:   end for
9:   load velocity of b
10:  update position and velocity of b using acc
11:  save new position and velocity in global memory
12: end for
```

カーネル内での計算量は、各物体ペア間での計算に 20 回の単精度浮動小数点演算を要し [8]、それを全ペア間で計算するため $20N^2$ の演算になる。この際、グローバルメモリに対して、$1 + N/B$ 回の物体の位置の読み込み、1 回の速度の読み込み、1 回の位置と速度の書き込みが行われる。位置情報、速度情報をともに 3 次元空間での値を持つために、128 ビットの `float4` 型で表現されている。従って、全体での実行時間 P は、

$$P = t_f(20N^2) + 4t_m(N/B + 3)$$

と表される。 ここで t_f、t_m は行列積の場合と同じく、それぞれ単精度浮動小数点演算に要する時間、4 バイトの値をアクセスする時間とする。

我々のライブラリを用いて上記カーネルに ECC による保護を導入する。同カーネル内の $(N/B+3)$ 回のメモリアクセスについて、それぞれ ECC の付与、検査を行うライブラリコールを追加する。128 ビットの ECC は 64 ビットの ECC2 回分として実現する。従って 128 ビットにつき 16 ビットの ECC を生成する。我々は二つの ECC を同時にグローバルメモリより読み書きする。従ってメモリアクセス回数はカーネル内の元のアクセス回数と等しく $(N/B + 3)$ 回、個々のデータサイズは 2 バイトである。CUDA では 2 バイトのアクセスでも 32 ビット単位で読み書きするため、実行時間は $t_m(N/B + 3)$ と表される。次に、ECC の計算に要する演算回数について考える。$(N/B + 3)$ 個の 128 ビットデータについてそれぞれ 64 ビットずつ計算する。64 ビットの ECC の計算には 144 回のビット演算を必要とする。従って、合計で $(N/B + 3) \times 144 \times 2$ 回、すなわち $288t_i(N/B + 3)$ の実行時間を要する。よって、カーネル全体の実行時間 P_{ecc} は以下の通り表される。

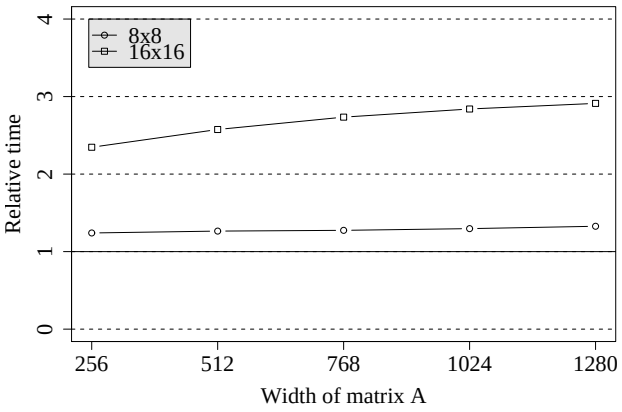

図 1　行列積の ECC による実行時間増加率

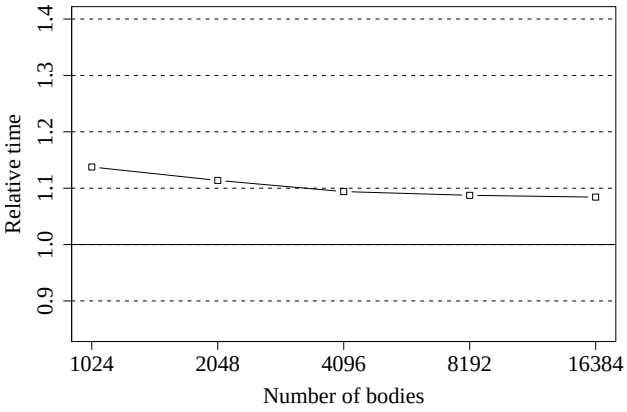

図 2　N 体問題の ECC による実行時間増加率

$$\begin{aligned} P_{\mathrm{ecc}} &= P + t_m(N/B+3) + 288t_i(N/B+3) \\ &= P + (t_m + 288t_i)(N/B+3) \end{aligned} \tag{4}$$

上記モデル化により、ECC 計算追加による実行時間オーバーヘッド E は以下の通り見積もれる。

$$E = \frac{P_{\mathrm{ecc}}}{P} = 1 + \frac{(t_m + 288t_i)(N/B+3)}{t_f(20N^2) + 4t_m(N/B+3)}$$

従って、物体数 N が大きくなるにつれて、オーバーヘッドの割合が小さくなる。しかし、物体数 N はグラフィックスメモリにロード可能な数として制約があるため、行列積の場合と同様に任意に削減可能ではない。定量的な評価を 4 節で示す。

4. 実験評価

ECC の計算、検査によるアプリケーション性能へのオーバーヘッドを評価するために、前述の行列積と N 体問題について ECC 有り無しで実行時間を比較した。評価環境は、AMD Phenom 9850 Quad-Core Processor (2.5GHz), 4GB のメインメモリ、Nvidia GeForce 8800 GTS4 枚を搭載した計算機である。OS として Fedora Core 8 を用い（Linux kernel v2.6.23)、さらに CUDA v1.1 をインストールして評価に用いた。CPU 向けコンパイラとしては gcc v4.1.2 を用いた。

図 1 に行列積における ECC による実行時間の増加率を示す。同グラフの X 軸は行列積 $\boldsymbol{A} \cdot \boldsymbol{B} = \boldsymbol{C}$　の行列 $\boldsymbol{A}$ の列数を表す。行列のその他のサイズはすべて 256 で固定した。Y 軸は元のプログラムに変更を加えず計測した時間に対して、ECC 有りの場合の実行時間の比率を示す。この値は小さければ小さいほどよく、1 の場合にオーバーヘッド 0 であることを意味する。スレッドブロックサイズとして、8×8 の 64 と 16×16 の 256 を設定した。

図より、64 スレッドの場合は速度低下 2 割程度に抑えられているが、256 スレッドの場合は 2 倍以上の低下となったことがわかる。3. 3 節で議論したように、行列積では性能低下率は常に 2 倍程度と予測されたが、実際にはそれと大きく異なる結果が得られた。まず、256 スレッドで予測より性能が低下した理由としては、複数 SP から ECC 読み込みのためのグローバルメモリへのアクセスがコアレッシングされず、シリアライズされてしまっていることが考えられるが詳細は不明である。また、64 スレッドで予測より良い性能が観測された理由も同様に不明であり、これの解析は今後の課題である。

図 2 に N 体問題における ECC による実行時間の増加率を示す。X 軸は物体数であり 1024 から 16384 までについて計測した。Y 軸は ECC 無しの元のプログラムに対する相対実行時間を表す。前述のグラフと同様に、Y 軸の値は小さければ小さいほどよく、1 の場合にオーバーヘッド 0 であることを意味する。スレッドブロックのスレッド数は 256 とした。

図より、オーバーヘッドは物体数によらずたかだか 15%程度であることがわかる。また、3. 4 節で議論したように物体数が増えるに従ってオーバーヘッドの割合が削減され、物体数 16384 の場合は 1 割以下まで削減されることがわかる。N 体問題のようにメモリアクセス量に対して計算量が多いアプリケーションではオーバーヘッドを小さく抑えられることを確認した。

5. 関連研究

Dopson は、通常の PC 向けメモリシステムの誤り検査をソフトウェアによる ECC で実現する手法を提案た [9]。OS に変更を加え、一定間隔毎にメモリをページ単位でチェックサムを計算し、誤りが発生していないか検査する。我々と異なり ECC の計算は一定間隔毎のため、前回の ECC 計算以降に書き込みが発生したページについては誤りを検出できない。また、使用されていないメモリについても一定間隔毎に常に ECC により検査を行うため、オーバーヘッドが大きい。我々は、プログラムのソースコード中のグローバルメモリへのアクセス直後に ECC 検査のための API 呼び出しを埋め込むことで、必要最低限の ECC 検査のみを行う。一方、Dopson による手法ではシステム全体のメモリを OS が検査するため、ユーザアプリケーション側では透過的に ECC による保護を受けられる利点がある。我々は現在の手動による手間を改善するために、ECC 検査コードの自動挿入について検討する予定である。

Sheaffer らは、GPU において冗長実行により耐故障性を実

現するアーキテクチャ拡張を提案した [10]。GPU 内部のフラグメント処理部などについて、計算カーネルを多重に冗長実行させることで誤り検知を行う。提案拡張をシミュレーションによって評価し、2 倍以下のオーバーヘッドで誤り検査付きの実行が可能であることを示している。同手法と異なり我々の提案はソフトウェアのみによるため、汎用性に優れる。また、Sheaffer らの提案は GPU 内部の計算パスを保護するものであり、グラフィクスメモリは保護されない。従って、我々の提案と Sheaffer らの提案は GPGPU における耐故障性実現にむけた相補的な関係にあると言える。

6. おわりに

我々は GPGPU におけるメモリシステムの耐故障性向上のために、ソフトウェア ECC を提案した。現状の GPU は ECC を備えておらず、通常の HPC 向けプラットフォームと比較して耐故障性に劣る。我々は、GPGPU の主流である CUDA について、グラフィックスメモリへのアクセスの際にソフトウェアにより ECC の計算、検査を実行する。これにより 1 ビットの誤り訂正と 2 ビットの誤り検知が可能である。提案方式を CUDA 向けライブラリとして実装し、行列積と N 体問題へ適用した。その結果、行列積では 2 割から 3 倍近くの速度低下が見られ、N 体問題では 2 割弱程度であった。

今後の課題として、得られた実験結果の詳細な解析が挙げられる。現在では行列積において予測と異なる傾向が観測されたが、その原因などについて調査する予定である。また、種々のアプリケーションに提案ライブラリを適用し、性能に関するさらなる知見を得る予定である。さらに、得られた知見を元に ECC 計算の自動挿入について検討を行う。また、ECC の計算部分は十分に最適化されているとは言えず、特に CUDA 環境のメモリアクセス最適化等が不十分である。今後、より詳細な性能解析と共に ECC 計算のチューニングも行う予定である。

謝辞 本研究の一部は Microsoft Technical Computing Initiative, "HPC-GPGPU: Large-Scale Comodity Accelerated Clusters and its application to Advanced Structural Proteomics"、及び科学技術振興機構戦略的創造研究推進事業「Ultra-Low-Powr 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」による。

文　献

[1] D. Blythe: "Rise of the Graphics Processor", Proceedings of the IEEE, **96**, 5, pp. 761–778 (2008).
[2] A. Nukada, Y. Ogata, T. Endo and S. Matsuoka: "Bandwidth intensive 3-d fft kernel for gpus using cuda", SC'08 (2008). To appear.
[3] Y. Ogata, T. Endo, N. Maruyama and S. Matsuoka: "An efficient, model-based cpu-gpu heterogenous fft library", 17th International Heterogeneity in Computing Workshop (HCW'08) (2008).
[4] 小川, 青木："Cuda による定常反復 poisson ベンチマークの高速化", 情報処理学会 HPC 研究報告 2008-HPC-115, pp. 19–23 (2008).
[5] ClearSpeed Technology plc: "ClearSpeed white paper: Csx processor architecture". `http://www.clearspeed.com/`.
[6] NVIDIA Corporation: "NVIDIA CUDA Compute Unified Device Architecture Programming Guide". `http://developer.nvidia.com/object/cuda.html`.
[7] E. Fujiwara: "Code Design for Dependable Systems: Theory and Practical Applications", Wiley Interscience (2006).
[8] L. Nyland, M. Harris and J. Prins: "Fast n-body simulation with cuda", GPU Gems 3 (Ed. by H. Nguyen), Addison-Wesley Professional, pp. 677–695 (2007).
[9] D. Dopson: "Softecc: A system for software memory integrity checking", Master's thesis, Massachusetts Institute of Technology (2005).
[10] J. W. Sheaffer, D. P. Luebke and K. Skadron: "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors", Graphics Hardware 2007 (Eds. by T. Aila and M. Segal), pp. 55–64 (2007).

付　録

図 A·1 にカーネル関数の一部を示す。ここで `check_ecc_float` と `gen_ecc_float` 部分が ECC の計算部分である。それぞれ関数呼び出しとして実現されているが、CUDA では関数呼び出しはすべてインライニングされる。

```
__global__ void
matrixMul_ecc(float* C, float* A, float* B,
              int wA, int wB,
              unsigned *C_ecc, unsigned *A_ecc,
              unsigned *B_ecc)
{
    ...
    for (int a = aBegin, b = bBegin;
         a <= aEnd; a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        int A_index = a + wA * ty + tx;
        AS(ty, tx) = A[index];
        // check loaded value with ECC
        ecc_check_float(AS(ty, tx), A_ecc, A_index);
        int B_index = b + wB * ty + tx;
        BS(ty, tx) = B[B_index];
        // check loaded value with ECC
        ecc_check_float(BS(ty, tx), B_ecc, B_index);
        __syncthreads();
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += AS(ty, k) * BS(k, tx);
        }
        __syncthreads();
    }
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int C_index = ...
    C[C_index] = Csub;
    // Generating ECC for C[C_index]
    ecc_gen_float(Csub, C_ecc, C_index);
}
```

図 A·1　行列積への適用例