社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－ARC－182 (5)
2009－HPC－119 (5)
2009／2／26

GPU向けソフトウェアECCの性能評価

丸山直也[†1,†3] 額田彰[†1,†3] 松岡聡[†1,†2,†3]

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している。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、FFT、行列積、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、FFT、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。

Performance Evaluation of Software-Based ECC for GPUs

Naoya Maruyama,[†1,†3] Akira Nukada[†1,†3]
and Satoshi Matsuoka[†1,†2,†3]

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: 3-D FFT, matrix multiplication, and an N-body problem. Our performance studies showed that while FFT and 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.

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),6),8)]。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[4)]。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[2)]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

†1 東京工業大学
†2 国立情報学研究所
†3 JST, CREST

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 を提案し、その予備的な性能を示した[9]。同提案では 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。

本論文では、性能評価対象アプリケーションとして新たに FFT を追加し、従来のプロトタイプ実装を改善した結果を示す。同プロトタイプは 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[5]、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、N 体問題、FFT 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、FFT、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[3]。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[3]。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[3] 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。

$\boldsymbol{G}$ 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、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（現在の GPU では 32）を単位として行われる。

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 512 (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、共有メモリから主に構成される。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、ローカル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。

グローバルメモリは、GPU において通常最も大きい容量を持つ一方、その遅延は他のオンチップメモリと比較して非常に遅く、400 から 600 サイクルと言われている。さらに、連続した 16 スレッド（ハーフワープ）が連続アドレスに 4 バイト、8 バイト、16 バイトにアクセスした場合、それらのメモリアクセスは一括に処理される（コアレッシング）。しかし、上記条件を満たさない場合、例えば各スレッドが 1 バイトずつアクセスする場合などでは、各スレッドのメモリアクセスが逐次的に処理され、性能低下を引き起こす。従って、バンド幅を向上させるためには可能な限りランダムアクセスを避け、連続領域へのバースアクセスにする必要がある。

スレッドブロック内で共有される共有メモリは、グローバルメモリと比較して高速だが小容量である (8800 GTS では MP あたり 16KB)。マルチスレッドプロセッサからのアクセスを並列に処理するため、16 のバンクとして構成され、連続した 4 バイトが連続したバンクにマップされる。共有メモリへのアクセスはハーフワープ単位（16 スレッド）で行われ、ハーフワープ中の複数スレッドから同一バンクにアクセスがあった場合はそれぞれ逐次処理され、性能低下の原因となる。ただし、異なるハーフワープに属するスレッド間では同一バンクへのアクセスであったとしてもアクセスタイミングが異なるためバンクコンフリクトは発生しない。従って、共有メモリへのアクセスを最適化するためには、ハーフワープ内の 16 スレッドが異なるバンクにアクセスするようアルゴリズムを設計する必要がある。

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。現在は GPU において最も大容量なメモリであるグローバルメモリへのアクセスを保護する。その他、共有メモリやテクスチャメモリ等の領域は本手法では対応しない。これらの保護は今後の課題である。

CUDA アプリケーションは一般的に以下の通り構成される。

(1) グローバルメモリにホスト側よりカーネル関数の入力データが転送される。
(2) 一つ又は複数のカーネル関数が転送されたデータを入力として実行される。
(3) 結果データがホスト側へ転送される。

従って、グローバルメモリ上のデータを常に保護するために、カーネル関数からのアクセスとホストとのデータ転送それぞれについて以下の操作を施す。カーネル関数からグローバルメモリへのリードアクセスの際は、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。ホストからのGPUへのデータ転送では、まずホスト上で転送データのECCを生成し、保護対象データと共にGPUへ転送する。実際にGPUカーネルが転送されたデータを用いる際は、同時に転送されたECCを用いて誤り検査を行う。GPUよりホストへデータを転送する際は、保護対象データと共にECCをホストへ転送する。ホスト側では転送されてきたデータをECCによって誤り検査を行う。以上の操作によりグローバルメモリ上データは常にECCがソフトウェア的に付与され、その誤りの検査が可能である。

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。現在のところ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採用している。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では主に4バイトや8バイトの単位でグローバルメモリへアクセスする。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。ECCの計算は主にビットシフトと排他的論理から実装できる。実際に32ビットECCと64ビットECCの計算を実装したところ、それぞれ43回、63回の32ビットのビット演算が主な演算となった。

3.3 最適化

上記手順を基本方針としてグローバルメモリの保護が可能であるが、実際にCUDA上で効率よく実現するためにはCUDA固有のアーキテクチャ、特にワープ単位のスケジューリングやメモリアクセスについて考慮する必要がある。最適化されたCUDAプログラムではグローバルメモリへのアクセスがコアレッシングされるよう設計される。我々は、対象プログラムがグローバルメモリへスレッドブロック内スレッドが1次元、2次元の部分データブロックに一括して順にアクセスする場合に特化することで、ECCによる検査を最適化する。以下、詳細を述べる。

保護対象データが4バイト、8バイトの場合共にECCは1バイトの領域を使う。従って、4バイトデータ型の配列のECCではその長さの4分の1のサイズの領域をECC領域として確保すれば充分である。この場合、この配列の1要素にGPUカーネル関数からリードアクセスがある場合、ECCの読み込み方法として単純に該当ECC1バイトのみにアクセスする方法が可能である。しかし、CUDAはマルチスレッドアーキテクチャであり、ECCの読み込みは同一ワープ内スレッドでも同時に実行される。従って、メモリアクセスのコアレッシングがされるべきだが、1バイトのアクセスではされない[*1]。コアレッシングされるためには各スレッドが最低4バイト単位でアクセスする必要があるが、データアクセス量が4倍になる。

1バイトのECCをマルチスレッドで効率良くアクセスするために、4要素のECCを1つの4バイトデータにパッキングし、4要素分一括してECCを読み書きする。この際、ECCの読み書きにはスレッドブロックの4分の1のみのスレッドを用い、共有メモリを介してスレッド間でECCを分散させる。具体的にはデータ読み込みの検査時にはグローバルメモリ上のECC保持領域より4分の1のスレッドが共有メモリへECCをロードする。ロード完了後、全スレッドにおいて自身の用いる1バイトレジスタへ読み込み、保護対象データより生成したECCと比較し、誤りの検査を行う。データの書き込み時には、まず対象データのECCを全スレッドが並列に生成し、それを共有メモリへ書き出す。書き出し完了後、前半4分の1スレッドが共有メモリより4つの該当ECCを用いてパッキングされたECCを生成し、グローバルメモリのECC保持領域へ書き出す。CUDAでは共有メモリに対する排他命令がサポートされないため、共有メモリへの全スレッドによる書き込み時にはパッキングせずに別個の要素に書き込む。以上のように、スレッドブロックの前半4分の1の連続したスレッドを用いることでECCを保持したグローバルメモリへのアクセスをコアレッシングさせる。

また、共有メモリ上のパッキングされたECCのアクセスにおいてバンクコンフリクトを回避するために、16のストライドでパッキングする要素を選択する。これによりハーフワープ内では別個のバンクからECCを読み込むためコンフリクトは発生しない。

プロトタイプECCライブラリでは、上記ECC生成、検査を1次元ブロック、2次元ブロックでアクセスする場合について実装した。

3.4 適用例

実装したECCライブラリの具体的な適用例として、額田らによる256^3の単精度複素3次元FFT[4]への適用について述べる。

同3次元FFTではx軸、y軸、z軸毎に1次元FFTを行う。その際、x軸方向FFTでは共有メモリにデータをロードし、通常の1次元FFTを行う。グ

*1 CUDA Compute Capability 1.2以降ではされる。

ローバルメモリへのアクセスをコアレッシングさせるために、y 軸、z 軸方向の FFT では 2 回の 16 点 FFT に対し、multirow アルゴリズムを用いる。

x 軸方向 FFT では各スレッドが 4 要素ずつ共有メモリにロードし、FFT を計算したあとグローバルメモリに書き出す動作をループとして構成し、全要素について実行する。1 要素は 8 バイトの複素数であるため、ECC は 1 バイトである。従って、1 度のループボディの実行において、読み込みの検査では各スレッドが 1 バイトの ECC を 4 つパッキングした ECC を共有メモリに読み込み、各スレッドが自身が読み込んだ 4 要素の ECC を生成し、共有メモリ上にある ECC と比較し、検査を行う。書き込み時には同様に 4 要素の ECC を生成し、それらを共有メモリへ書き出す。書き出し完了後、各スレッドが 4 要素分の ECC をパッキングした 4 バイトデータを生成し、グローバルメモリへと書き出す。この際、ストライド 16 でパッキング対象データを選択し、共有メモリ上でのバンクコンフリクトを回避する。また、グローバルメモリへのアクセスも常にコアレスされる。

y 軸、z 軸方向 FFT では、16 点 FFT では、ループボディにおいて各スレッドが 16 要素の読み込みと書き込みを行う。この場合は、16 要素分の ECC16 バイトを一括して共有メモリにロードし、各スレッドで自身の 16 要素を共有メモリ上の ECC を用いて検査する。書き込み時には、4 要素ずつ共有メモリに ECC を書き出し、全スレッドで協調してグローバルメモリへ書き出す。

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、FFT、行列積、N 体問題について ECC 有り無しで実行時間を比較した。行列積は我々の実装による CUDA 向け行列積に ECC による検査を追加した。N 体問題は CUDA SDK に付属のサンプルプログラムを用いた。評価環境は、AMD Phenom 9850 Quad-Core Processor (2.5GHz), 4GB のメインメモリ、Nvidia GeForce 8800 GTS 512 を 4 枚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 v2.1 をインストールして評価に用いた（N 体問題のみ v1.1）。CPU 向けコンパイラとしては gcc v4.1.2 を用いた。

図 1 に FFT を適用した場合の性能比較を示す。比較対象として、オリジナルの FFT (NUFFT)、NVIDIA による CUDA 付属の FFT ライブラリである CUFFT を用いた。NUFFT w/ ECC が ECC 版の性能である。NUFFT は 56GFLOPS であるところ、ECC 版では 19GFLOPS であり、ECC により 3 倍近いオーバーヘッドがかかったことがわかる。一方、オリジナルの FFT は既に高度に最適化されており、ECC 版

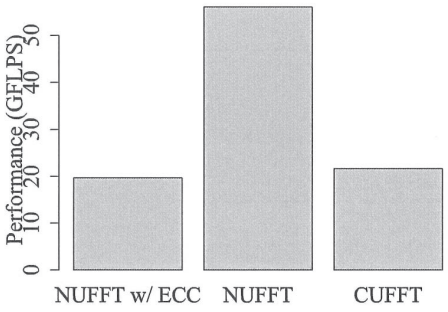

図 1 FFT の ECC による実行時間増加率

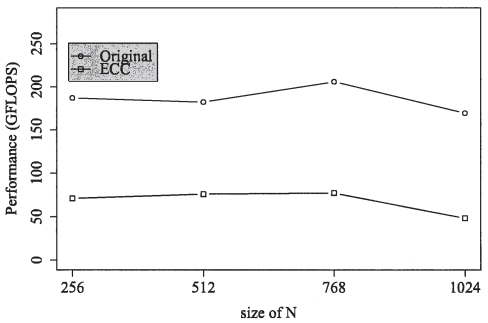

図 2 行列積の ECC による実行時間増加率

でも標準の FFT 実装である CUFFT と同等の性能であった。

図 1 に行列積に適用した場合の性能比較を示す。行列はすべて長さ N の正方行列を用いた。グラフの X 軸は N を表し、Y 軸は性能 (GFLOPS) を表す。図より、

図 3 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 軸は実行時間を表す。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、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。

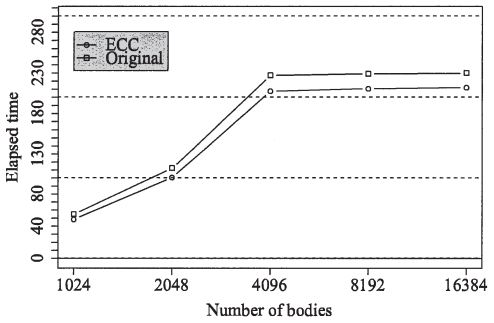

図 3 N 体問題の ECC による実行時間増加率

5. 関連研究

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[7]。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、FFT、行列積、N 体問題へ適用した。その結果、FFT や行列積では 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。

今後の課題としては、ECC によるオーバーヘッドの削減が挙げられる。例えば現在では 32 ビット、もしくは 64 ビット単位で ECC を計算しているが、可能な場合は 128 ビット単位で計算することで ECC の計算コストを削減できる。それによりエラー検知精度が低下するが、性能とのトレードオフを考慮して許容可能な範囲のエラー検査を行うべきである。種々の方式について性能評価を行い、それに基づき ECC のコストモデルを構築し、適切な手法の自動選択についても取り組む予定である。また、現在のところホスト側から GPU へのデータ転送とその際の ECC の生成、検査コストの評価がされていない。これについても取り組む予定である。

謝辞 本研究の一部は Microsoft Technical Computing Initiative, "HPC-GPGPU: Large-Scale Commodity 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。

参考文献

1) Blythe, D.: Rise of the Graphics Processor, *Proceedings of the IEEE*, Vol.96, No.5, pp.761–778 (2008).
2) ClearSpeed Technology plc: ClearSpeed white paper: CSX Processor Architecture. `http://www.clearspeed.com/`.
3) Fujiwara, E.: *Code Design for Dependable Systems: Theory and Practical Applications*, Wiley Interscience (2006).
4) Nukada, A., Ogata, Y., Endo, T. and Matsuoka, S.: Bandwidth Intensive 3-D FFT Kernel for GPUs using CUDA, *SC'08* (2008).
5) NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture Programming Guide. `http://developer.nvidia.com/object/cuda.html`.
6) Ogata, Y., Endo, T., Maruyama, N. and Matsuoka, S.: An Efficient, Model-Based CPU-GPU Heterogenous FFT Library, *17th International Heterogeneity in Computing Workshop (HCW'08)* (2008).
7) Sheaffer, J. W., Luebke, D. P. and Skadron, K.: A Hardware Redundancy and Recovery Mechanism for Reliable GPGPU, *Proceedings of Eurographics/ACM Graphics Hardware 2007 (GH)*, pp.55–64 (2007).
8) 小川 慧, 青木尊之：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).
9) 丸山直也, 松岡 聡, 尾形康彦, 額田 彰, 遠藤敏夫：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, 電子情報通信学会技術研究報告 DC2008-20, pp.9–16 (2008).