Vol.2009-ARC-186 No.8
Vol.2009-HPC-123 No.8
2009/12/1

GPU 向け耐メモリエラーソフトウェアフレームワーク

丸 山 直 也[†1] 額 田 彰[†1] 松 岡 聡[†1,†2]

我々はコモディティGPU を対象とした耐 DRAM ソフトエラーを実現するソフトウェアフレームを提案する。同フレームワークは符号化によるビットフリップ等の DRAM ソフトエラーを検知する。エラーが検知された場合、ホスト側に取得済みのチェックポイントから GPU カーネルを再実行することでエラーからの復旧を実現する。同フレームワークを CUDA GPU 上で実装した場合の性能を評価し、エラーチェックによるオーバーヘッドは、行列積のような計算負荷の大きいカーネルでは 10 パーセント以下、3D FFT のようなメモリアクセス負荷の大きいカーネルにおいて 35%程度で抑えられることを示す。

Software Framework for GPU Memory Errors

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

We present a high-performance software framework to enhance commodity off-the-shelf GPUs with DRAM fault tolerance. It combines data coding for detecting bit-flip errors and checkpointing for recovering computations when such errors are detected. We analyze performance of data coding in GPUs and present optimizations geared toward memory-intensive GPU applications. We present performance studies of the prototype implementation of the framework and show that the proposed framework can be realized with very low overheads in compute intensive applications such as matrix multiplication, and as low as 35% in a highly-efficient memory intensive 3-D FFT kernel.

†1 東京工業大学
Tokyo Institute of Technology
†2 国立情報学研究所
National Institute of Informatics

1. はじめに

GPU による HPC アプリケーションの高速化が注目されている一方で[5),6),8)]、本来グラフィックス用途として設計されてきた GPU を HPC アプリケーションに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。NVIDIA は HPC 向け次世代 GPU に ECC を搭載することをアナウンスしているが、実際の製品としては未だ存在しない。また、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いため、コンシューマ向けコモディティGPU である GeForce シリーズへの ECC の搭載される可能性は低い。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、ゲームなどのグラフィックス用途に比べて HPC アプリケーションは実行時間が一般的に長い。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がある。

この問題を解決するために、我々は GPU DRAM におけるビットフリップエラーを対象とした耐故障ソフトウェアフレームワークを提案する。本フレームワークはまず GPU アプリケーションを実行する前に入力データのコピーをホスト側に保存する。アプリケーションを GPU で実行中に DRAM データの正しさをデータ符号化により検査し、エラーが検出された場合はホスト側に存在する入力データのコピーを用いて再実行する。

これによりビットフリップエラーの検出とエラー発生じのリカバリを実現可能であるが、その性能について検証が必要である。実際に我々はハミング符号を用いた (72, 64) Single-Error-Correction Double-Error-Detection (SEC-DED) ECC のソフトウェア実装ではアプリケーション性能において最大 3 倍程度の大幅なオーバーヘッドが生じることを報告している[11)]。本論文では典型的な GPU アプリケーションに特化することで、(72, 64) SEC-DED ECC ハミング符号と同程度の信頼性を実現しつつ、大幅にオーバーヘッドを削減する方式を示す。典型的な GPU アプリケーションではメモリアクセスは非常に多数のスレッドによりバースト的に行われることに着目し、128 バイトのデータブロックに対してクロスパリティ符号を計算する[2)]。同符号化では 32 個の 4 バイトデータより 32 個のパリティビットと同データをローテートしたデータより 32 個のパリティビットを生成する。これにより通常

© 2009 Information Processing Society of Japan

のハミング符号と比較して計算量を大幅に削減しつつ、4 バイト中の 1 ビットエラーと 128 バイト中の 2 ビットエラーの検出が可能である。

これらの符号化によるエラーチェックは GPU カーネルのメモリアクセス時に実行される。従って、メモリアクセス比率の大きいカーネルではチェックによるオーバーヘッドが比較的大きい。しかしそのようなメモリアクセス比率の大きいカーネルでは計算コアがアイドルである時間が多くなると言え、符号化によるオーバーヘッドをそのアイドルコアを活用することで削減可能である。また、計算負荷の大きいカーネルではアイドルコアの活用によるオーバーヘッドの削減は期待できないが、そもそもそのようなカーネルではメモリアクセスの割合が小さいためカーネル実行時間全体に対するオーバーヘッドは限定的と言える。実際に提案フレームワークを CUDA GPU アプリケーションについて実装し、その性能を評価した。その結果、行列積のような計算負荷の大きいプログラムでは数パーセント以下、3D FFT のようなメモリアクセス負荷の大きいプログラムにおいて 35%程度で抑えられることがわかった。

2. GPU 向け高効率耐 DRAM エラー手法

我々は、通常のサーバー向け DRAM における ECC、すなわち 8 バイトデータに対する SEC-DED ECC と同等の信頼性の実現を目標とし、チェックポイントとエラー検知を組み合わせた手法を提案する。

多くの CUDA アプリケーションにおける GPU の利用は、実行プログラムをカーネル関数として記述し、その入力データの GPU への転送、カーネルの実行、出力データのホストへの転送の 3 つのステップからモデル化できる。すなわち、GPU への初めの入力データの転送時にそのコピーをホスト側に維持することでカーネル関数の再実行が可能である。我々はこの特徴を利用した CUDA 向け簡易チェックポイントライブラリを実装した。本ライブラリでは通常の CUDA API をラッピングし、GPU メモリ割り当て、コピー等の際にそのデータをホスト側にコピーが存在することを保証し、GPU メモリの復旧のための API を提供する。本ライブラリは、ホスト上に GPU データを保持するためホスト自体の故障等には対応しないが、ホストも含めたシステム全体を GPU コンテキストと共にチェックポイントをとる手法も提案されており[9]、我々のフレームワークでも同様の技術を用いることが可能である。

チェックポイントを用いることで、SEC-DED と同等の信頼性を実現するためにはエラー検知のみ実現すれば十分であり、我々はクロスパリティコードを応用した符号化を用いる[3].。

表 1 符号化によるエラーチェックのスループット

Device	Original	Cross Parity	ECC
GTX 285	148.15 GB/s	124.27 GB/s	34.28 GB/s
S1070	87.54 GB/s	75.92 GB/s	33.03 GB/s

同符号化では、GPU アプリケーションでは一般的に比較的大きなブロック単位でデータにアクセスする場合が多く、GPU メモリアーキテクチャ自体もそのようなデータアクセスパターンに最適化されていることから、ブロック単位で符号化を行うことで計算量を削減する。通常のクロスパリティでは 2 次元データブロックに対してパリティビットを 2 種類生成する。ここで、ブロックを N 個の連続 N ビットワードからなる領域とする。1 つ目のパリティは個々のワードから 1 ビットのパリティビットを生成するものであり、水平符号化と呼ばれる。2 つ目のパリティは、N 個のワードから同一ビットオフセットのビット集合よりパリティビットを生成するものであり、垂直符号化と呼ばれる。水平符号と同等の信頼性を持ちよりソフトウェアで実装効率の良い手法として対角線上のビット列のパリティを用いる対角符号があり[7]、我々はこれと垂直符号からなるクロスパリティ符号を用いる。

我々はクロスパリティをスレッドブロックがアクセスする 128 バイト毎のデータブロックに対して適用する。データはスレッドブロック内の各スレッドからアクセスされるため、符号化対象のデータは各スレッドに分散して保持されている。我々はスレッドブロック内のスレッドについて、リダクション操作を繰り返すことで協調的にクロスパリティを計算させる。

GTX 285 と S1070 において上記符号化によりエラーチェックを行った場合のメモリ読み込みスループット、通常のスループット、ソフトウェア ECC によるチェックを行った場合のスループットを表 1 に示す。テストに用いたカーネルは 256 スレッドからなる 480 のスレッドブロックからなり、全体で 128MB のデータをグローバルメモリからレジスタへ読み込む。クロスパリティでは両デバイスにおいておよそ 84%と 86%の性能を達成できることを確認した。

通常の SEC-DED ECC では常に 64 ビットワードあたり 2 ビットエラーまで検知可能だが、クロスパリティでは検知が保証されるのは 32 ビットワード内における単一ビットエラーのみである。2 ビットエラーはクロスパリティを適用する 128 バイトのブロックにおいては常に検知されるが、64 ビットワード内では検知されるとは限らない。例えば、128 バイト内の複数の 64 ビット領域で 2 ビットエラーが発生した場合は検知を保証しない。一方で一

© 2009 Information Processing Society of Japan

般的に 2 ビットエラーの頻度は 1 ビットエラーと比較して非常に小さいことが知られている。例えば、Schroeder らによる DRAM エラー頻度を統計的に調査した結果では、2 ビットエラーの頻度は 1 ビットエラーの 3%程度であった。また上記スループットの 4 倍近くの向上によりアプリケーション実行時間も削減されるため、結果として故障の可能性も削減される。従って、ワード内 2 ビット検知を保証することもソフトウェア実装のため容易に変更可能だが、多くのアプリケーションではクロスパリティによる信頼性と性能のトレードオフが適していると言える。

3. CUDA アプリケーションにおける符号化による耐エラーソフトウェアフレームワーク

クロスパリティ符号によるエラーチェックとチェックポインティングを実際に CUDA アプリケーションに適用する、ソフトウェアフレームワークを提案する。同フレームワークでは故障モデルとして GPU DRAM (CUDA global memory) におけるビットフリップを想定し、ホスト側も含めたその他のコンポーネントの故障は発生しないものとする。アプリケーションの実行を 1) 入力データ転送、2) カーネル実行、3) 出力データ転送の 3 つのステップに分割し、それぞれのステップについて以下に説明する。

3.1 入力データ転送

CUDA ではカーネル実行の前に入力データをホスト側メモリより GPU 側メモリへ CUDA API 用いて転送する。GPU DRAM におけるエラーはデータ転送直後に発生しうるため、そのエラーチェック符号をホスト側で生成し、入力データと共に転送する必要がある。従って、アプリケーション全体の性能に対して、CPU を用いた符号化の時間と符号の GPU への転送の時間がオーバーヘッドとなる。この際、転送コストに関してはそのサイズがたかだか元のデータの 16 分の 1 と小さいため、それによるオーバーヘッドも限定的と言える。CPU 上での符号化のコストはデータのホスト DRAM からの読み込みとそれに対するクロスパリティの計算からなるが、クロスパリティの計算は単純な論理演算のみであるため主にホストメモリへのスループットによって決まる。一般的に、ホストメモリのスループットは GPU メモリのそれに比較して数パーセントから 10 パーセント程度と大幅に劣るため、GPU カーネルの規模によっては CPU 上での符号化のコストがアプリケーション全体に対して大きなオーバーヘッドとなりうる。

我々は CPU 上での符号化のコストを削減するために、符号化と元のデータの GPU メモリへの転送をオーバーラップさせ、符号化のコストを隠蔽する。G PU への転送速度はたかだか 8GB/s 程度であり、一方ホストのメモリアクセス速度は DDR3 を用いることで同等の性能を達成可能である。従って、両者を同時に実行させることにより、符号化のコストを大幅に隠蔽可能である。実際に入力データ転送時のオーバーヘッドを低く抑えられることを 4 節で示す。

3.2 カーネル実行

カーネル実行時には各スレッドがアクセスする global memory の値についてエラーチェックを行うために、読み込み時には符号化による検査、書き込み時には符号ビットの生成と書き込みを行う。我々のフレームワークでは現在のところこれらの処理を元のカーネルソースへの変更によって実現する。いくつかの典型的なデータアクセスパターン向けの符号化ルーチンを提供し、対象カーネルからは適切なルーチンを呼び出すことで実現する。

読み込み時のエラーチェックについては、以下の二つの手法を選択的に用いる。一つ目はインラインチェックであり、これはスレッドがデータをレジスタに読み込んだ直後に毎回検査する手法である。もう一つはオフラインチェックであり、これはカーネル実行終了後に global memory 中のデータの正当性をチェックする。読み込みのみのデータに関しては後者のオフラインチェックが適用可能である。前者と比較して後者はそのチェックを別カーネルとして実現でき、元のプログラムへの変更が必要ない等、比較的容易に実現可能である。一方、前者では元のプログラム中でレジスタ等のオンチップメモリに読み込まれたデータをチェックするのに対して、後者ではチェックの際も global memory のデータを再度読み込む必要があり、データアクセスのコストが比較的大きい。FF T、流体計算のようなメモリアクセス時間が主な計算では前者のインラインチェックの方が性能的には優れている。

3.3 出力データ転送

最後のステップではカーネル実行の結果をホスト側メモリへ転送する。この転送時にも GPU DRAM 内においてエラーが起こりうる。我々はこのエラーを検知するため、ホスト側へ転送終にデータのエラーチェックを GPU 側で行う。実際には PCIe の転送路においてもエラーは起こりうるがそのようなエラーは PCIe が備える CRC により検出される。

4. ケーススタディ

提案フレームワークの性能への影響を調べるために、2 つの CUDA アプリケーションへ適用しその性能を評価する。1 つ目は行列積であり、計算が全体の主な時間を占めるアプリケーションの例として選択した。2 つ目は 3 次元 FF T であり、メモリアクセス時間が主要な時間を占める。すべての評価は、プロセッサとして AMD Phenom II X4 955(Quad-core)、

© 2009 Information Processing Society of Japan

チップセットとして AMD 790FX、さらに 8GB の DDR3-1600 メモリを用いた。OS としては 64 ビット Fedora 10 を用い、NVIDIA GPU Driver 190.18、GCC 4.3.2、CUDA Toolkit 2.3 を用いた。

4.1 行 列 積

行列積は科学技術計算において最も重要な計算の一つであり、$O(n^3)$ の計算と $O(n^2)$ のデータアクセスを行う (n は入力を正方行列と仮定した場合の一辺の長さ)。提案フレームワークの実装には基となる行列積ソースコードが必要なため、我々は単精度浮動小数正方行列の積を計算するカーネルを実装した。同カーネルは、スレッドブロック単位で結果の行列の部分ブロックについて内積を計算し、その際に共有メモリを用いて DRAM アクセスのコストを最適化するものであり、その性能は CUBLAS と同等である。

我々は同カーネルに対して以下の通りフレームワークを適用した。まず、入力データについては読み込みのみであり、かつ行列積は計算時間が主なためオフラインチェックを用いる。これは初めにホスト側でクロスパリティ符号を生成し、それの GPU への転送、さらに行列積カーネルの実行終了後に入力行列のチェックを行う別のカーネルの実行によって実現される。結果の行列に対しては、基の行列積カーネルに対してその行列への内積の書き込み直後にクロスパリティをスレッドブロック毎に生成し global memory へ保存する。さらに結果のホストへの転送後に global memory 上の同データの正当性をクロスパリティによってチェックするカーネルを実行する。以上により行列積実行時データをすべて検査する。

エラーから復帰するためのチェックポイントとして、入力となる 2 つの行列についてはその元データがホスト側にカーネル実行中存在するように保証する。これは我々の簡易チェックポイントライブラリを用いて実現する。同チェックポイントライブラリでは指定されたデータについて、カーネル実行中にホスト側で変更される場合のみその元のデータを別領域にコピーし、常にカーネルを再実行可能なものとする。

4.2 3 次元 FFT

3 次元 FFT における提案フレームワークの性能を評価するために、額田らによる 256^3 単精度複素 3 次元 FFT のカーネルに適用した[4)]。同カーネルは各次元に対して順方向の変換後、逆方向の変換を行う。同カーネルはメモリアクセス時間が主なため、インラインチェックを用いた。インラインチェックでは、元の 256 スレッドから構成されるスレッドブロックが常に 2KB の連続領域にアクセスするため、16 の 128KB ブロック毎に各スレッドブロックがクロスパリティを生成する。これにより読み込み時のチェックと書き込み時の符号の付与を行う。入力と出力の転送時については行列積と同様な変更を施した。

4.3 結　　果

図 1 と図 2 は、それぞれ行列積と FFT について基のバージョンと比較したグラフである。また、参考までに CPU 上での同等の計算の時間として GotoBLAS v1.26 と FFTW v3.2.2 の結果、さらに CUDA 付属の FFT ライブラリである CUFFT v2.3 の結果を示した。GPU プログラムについてはそれぞれカーネルのみの性能と入出力データ転送の時間を含んだ全体の性能を示した。

結果より、行列積ではフレームワーク適用によるオーバーヘッドはたかだか基のカーネルの 10%に抑えられることがわかった。さらに、このオーバーヘッドは行列サイズが大きくなるにつれて小さくなることがわかる。これは行列積のコストはそのサイズが大きくなるにつれて計算時間が主になるためであり、メモリアクセス時のコストが相対的に小さくなるためである。また、入出力を含めた全体の性能もその傾向はカーネルのみの性能と同様であることがわかり、ホスト側での符号の計算等のコストは無視できるほど小さいことがわかる。

一方で、FFT では 35%程度と比較的大きなオーバーヘッドが観測された。これは FFT は一般的にはメモリアクセスが主要な時間を占めるが、元もとのカーネルは高度にチューニングが施されており計算ユニットがアイドルである割合が符号化のコストを隠蔽するには十分でなかったことが推定できる。

5. 関 連 研 究

現状市販されている中で最新の NVIDIA GPU である Tesla S1070 や GTX 285 などにはエラー検知、修正などの機能は我々の知る限り一切含まれていない。NVIDIA は次期 GPU である Fermi において外部 DRAM に限らずレジスタ、オンチップキャッシュ等も含めた ECC による保護を提供することを発表済みだが、実際の製品としては未だ存在せずその性能等は不明である。また、ハードウェアによる実装は Tesla 等の HPC 専用 GPU に限られ、より安価な GPU である GeForce には搭載されない可能性が高い。我々の手法はソフトウェアのみによるものであるため、任意の NVIDIA GPU に対して適用可能である。また、AMD は同社の GPU への ECC 等のサポートを提供していないが、我々の手法を応用可能である。

同様に GPU の信頼性向上の研究として、Dimitrov ら[1)] や島田ら[10)] による多重実行が挙げられる。Dimitrov らの手法では AMD GPU において、空き VLIW スロットを活用することで多重実行のオーバーヘッドを削減可能であることが示されている。我々の手法においてもメモリアクセス時に空き計算コアを活用することでコストの隠蔽を図っており、コス

© 2009 Information Processing Society of Japan

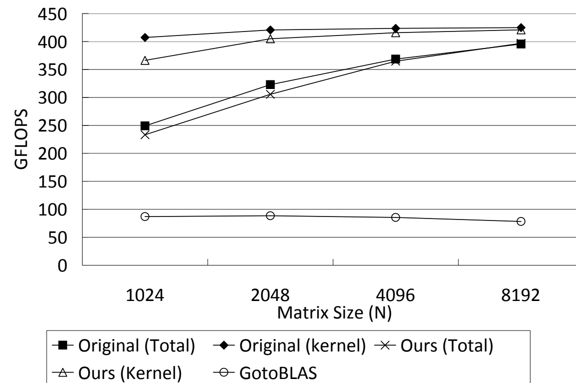

図 1　N^2 行列積の性能比較.

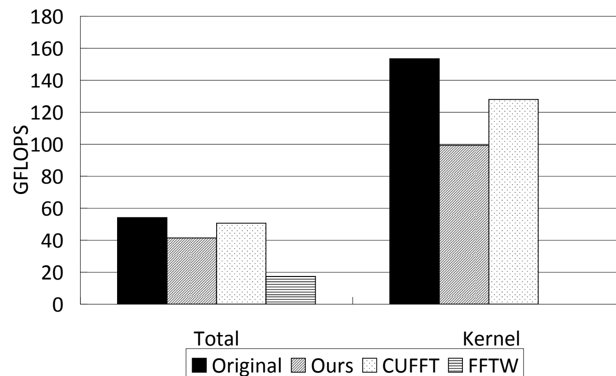

図 2　3 次元 256^3FFT の性能比較

ト削減手法の観点より両手法は本質的に関連していると言える。

Shirvani らはホスト DRAM のスクラビング手法を提案しており[7]、我々のクロスパリティ符号も Shirvani らの手法を基に CUDA GPU 向けに拡張したものである。また、Shirvani らの手法では符号化の計算コストが非常に大きいためアイドル時のスクラビングのみ行われる。我々は GPU が大規模マルチスレッドプロセッサである特徴を利用し、エラーチェックと元のメモリアクセス・計算をスレッド間でオーバーラップさせることで、計算実行時にチェックを行った場合でも高々35%程度のオーバーヘッドで抑えられることを示した。

6. おわりに

我々は GPU DRAM におけるビットフリップ等のソフトエラーの検知、回復を実現するソフトウェアフレームワークを提案した。同フレームワークは、エラー検知符号とチェックポイントを組み合わせることで、4 バイトワード内の単一ビット、128 バイトブロック内の 2 ビットエラーまで対応可能である。実際に行列積と FFT について提案フレームワークを適用したところ、行列積で 10%程度、FFT で 35%程度のオーバーヘッドで実現できることがわかった。これはメモリエラーに限定されるものの、多重実行等による 2 倍以上のオーバーヘッドを要する手法に比べて大幅に軽量な手法と言える。

今後の課題としては、フレームワークのアプリケーションへの適用の自動化、チェックポイントによるハード故障への対応などが挙げられる。

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。

参　考　文　献

1) Dimitrov, M., Mantor, M. and Zhou, H.: Understanding software approaches for GPGPU reliability, *2nd Workshop on GPGPU*, pp.94–104 (2009).
2) Fujiwara, E.: *Code Design for Dependable Systems: Theory and Practical Applications*, Wiley Interscience (2006).
3) Mano, T., Yamada, J., Inoue, J. and Nakajima, S.: Circuit techniques for a VLSI memory, *IEEE Journal of Solid-State Circuits*, Vol.18, No.5, pp.463–470 (1983).
4) Nukada, A., Ogata, Y., Endo, T. and Matsuoka, S.: Bandwidth Intensive 3-D FFT

© 2009 Information Processing Society of Japan

Vol.2009-ARC-186 No.8
Vol.2009-HPC-123 No.8
2009/12/1

Kernel for GPUs using CUDA, *SC'08* (2008).
5) Nyland, L., Harris, M. and Prins, J.: Fast N-Body Simulation with CUDA, *GPU Gems 3* (Nguyen, H., ed.), Addison Wesley Professional, chapter31 (2007).
6) Schatz, M., Trapnell, C., Delcher, A. and Varshney, A.: High-throughput sequence alignment using Graphics Processing Units, *BMC Bioinformatics*, Vol.8, No.1, pp. 474+ (2007).
7) Shirvani, P.P., Saxena, N.R. and McCluskey, E.J.: Software-implemented EDAC protection against SEUs, *Reliability, IEEE Transactions on*, Vol.49, No.3, pp.273–284 (2000).
8) Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G. and Schulten, K.: Accelerating molecular modeling applications with graphics processors, *Journal of Computational Chemistry*, Vol.28, No.16, pp.2618–2640 (2007).
9) 滝沢寛之，佐藤功人，小松一彦，小林広明：CUDA アプリケーション向けチェックポイント・リスタート機能の実装と評価，情報処理学会研究報告 HPC 2009-HPC-122 (2009).
10) 島田大地，丸山直也，額田　彰，遠藤敏夫，松岡　聡：GPU における耐故障性を考慮した数値計算の電力性能，情報処理学会研究報告 2009-HPC-121 (SWoPP 2009) (2009).
11) 丸山直也，額田　彰，松岡　聡：GPU 向けソフトウェア ECC の性能評価，情報処理学会研究報告 HPC 2009-HPC-119，pp.25–30 (2009).

© 2009 Information Processing Society of Japan