Approximate Computing での NaN の除去による継続的な実行

濱田槙亮[†]　穐山空道[††]　並木美太郎[†]

1. はじめに

近年、サーバマシンに搭載されるメモリ容量が増加しており、ビッグデータ解析や機械学習、インメモリ KVS(Key Value Store) など大容量のメモリが要求される。また大容量メモリの搭載によって、消費電力のうちメモリサブシステムが占める割合が増加している。そこでメモリシステムの消費電力を削減することで、全体の消費電力の改善が期待できる。

メモリの消費電力を削減する手法として Approximate Computing が存在する。Approximate Computing は本来決められたハードウェアの処理にかかる時間や比率を増加/削減することによって消費電力の削減を図る手法である。例えば DRAM のリフレッシュレートを低下させることによってメモリセルのチャージで発生する消費電力を削減できる。しかし好ましくない影響もある。リフレッシュレートを下げることによって DRAM の消費電力は削減できるが、メモリの中身が一意であることが保証されない。Approximate Computing によってメモリの中身が化けて期待しない値となり、値の化け方によってはアプリケーションの実行が不可能になってしまい計算途中で処理が異常終了となる。先行研究[1] では、ポインタが化けた場合の対処方法について述べられている。

本研究では、浮動小数点に致命的なエラーが発生しても継続的に実行できる方法について扱う。

2. Approximate Computing による浮動小数点演算への影響

実行中のアプリケーションには、値が化けてもいいデータと化けてはいけないデータが存在する。化けてはいけない大事なデータにはテキスト領域やカーネル領域などが挙げられ、これらは化けてもいいデータとは分けてメモリに配置される[2] 。化けてもいいデータは計算用のデータであり、物理シミュレーションや機械学習などでは、ある程度化けても最終的な結果は収束していくため無視できる。

しかし、浮動小数点では化けた結果が NaN になる可能性がある。ビッグデータ解析や人工知能アプリケーションでは小数点を含む数値を取り扱うことがほとんどであり、NaN による計算結果への影響は大きくなる。例えば行列に NaN が含まれる場合、行列演算のたびに NaN が伝搬する。行列同士の乗算では演算元の 1 つの NaN が演算結果に 1 行もしくは 1 列の NaN となり、NaN を含む行列の行列式は NaN となる。行列を扱うアプリケーションにおいて行列内に NaN が発生することは致命的であり、計算結果が全く使えなくなる。そのため実行中に NaN が発生しても処理を継続でき、計算結果へは無視できる程度の影響に留める環境が必要となる。

3. 提案手法

3.1 基本アイディア

NaN を対象とした浮動小数点演算は例外として定義されており、どのように対応するかは OS・処理系によって決定される。Linux では例外をハンドルし、例外を起こしたアプリケーションへ SIGFPE シグナルを発行する。

例外の原因となった NaN を 0 もしくはランダムな値に書き換えることで、アプリケーションの実行を継続させ、収束していく計算結果への影響を抑えられる。これを OS から発行される SIGFPE シグナルをキャッチし、例外で停止しているアプリケーションの NaN を書き換えることで実現する。

書き換える対象は、レジスタとメモリの 2 つが挙げられる。例外の直接的な原因となる NaN はレジスタに存在し、これを浮動小数点として適切な値に置き換えて実行を再開すればアプリケーションは何事もなかったように処理を続けられる。しかし、本研究ではメモリに対する Approximate Computing を想定しているので、レジスタに存在する NaN はメモリから

† 東京農工大学
Tokyo University of Agriculture and Technology
†† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology

ロードされたものである。そのため再度メモリからレジスタへロードされ演算で使用されると例外を引き起こす。そこでメモリに存在する NaN も置き換えることで、例外の発生を抑制できる。

3.2 実装方針

シグナルのキャッチ及びレジスタやメモリの値の取得と書き換えには gdb を使用した。実行アプリケーションは gdb にアタッチされた状態で実行されることとなる。gdb による時間的オーバーヘッドは、シグナルキャッチ時に発生する操作がほとんどである。今回は書き換える値に 0(全てのビットが 0) を採用した。

実行の流れは以下の通りとなる。

(1) gdb がアタッチしてアプリケーションの実行を開始
(2) 計算途中で NaN による SIGFPE が発生し gdb でキャッチする
(3) gdb でレジスタまたはメモリを操作しアプリケーションの実行を再開する (NaN が再度混入した場合は (2) に戻る)
(4) アプリケーションの実行を終了する

3.3 レジスタの書き換え

gdb が SIGFPE をキャッチした時、命令レジスタは例外を引き起こした命令を指している。そのため命令レジスタから、例外を起こした命令とレジスタを特定できる。これを利用して、レジスタの中身を探り NaN になっている箇所を特定する。gdb の機能でレジスタの値を書き換えてから実行再開させる。あらかじめ gdb でシグナルをキャッチしアプリケーションに通さないようにしておくことで、アプリケーションは例外を起こしたことを知らずに実行を再開できる。

3.4 メモリの書き換え

シグナルをキャッチした状態では、どのレジスタに NaN があるかまでしか把握できない。しかし NaN がどのようなビット列であるかは知ることができる。そこで、計算用データが配置されているであるメモリ領域をダンプし、ビット列で検索をかけて NaN がいる位置を特定しメモリを直接書き換える。しかしこの手法ではメモリのダンプにオーバーヘッドの多くが占めるため、実行命令を逆方向に辿ることで一意に NaN のアドレスを特定できることが理想である。

4. 検証結果

Intel Xeon CPU E5-2699 v3 で動作する Linux で動作を検証した。NaN を意図的に仕込んだ演算をレジスタとメモリの書き換えによって継続的に実行可能なことを確認した。

N 次の正方行列同士の掛け算において、NaN が含まれない通常の状態で実行した場合と、NaN を行列内に仕込んで gdb でアタッチし提案手法を用いた場合で、実行時間を計測した。提案手法では、レジスタのみの書き換えとレジスタとメモリの両方の書き換えの 2 種類で計測した。結果は図 1 に示す。横軸は行列の次数を示し、縦軸はかかった実行時間を示す。*normal* は通常の実行で、*register* はレジスタのみの書き換え、*memory* はレジスタとメモリの両方を書き換えた場合の実行時間である。

レジスタのみを書き換える場合ではメモリに *NaN* が残るので、参照されるたびに例外となり *gdb* が割り込むため大きな時間的オーバーヘッドとなる。対して、メモリも合わせて書き換える場合では最初の *1* 回の例外で抑えることができるため通常の実行時間と同じ程度となった。しかしメモリを一度ダンプしシーケンシャルに検索しているため、行列が大きくなるとオーバーヘッドも増大すると考えられる。

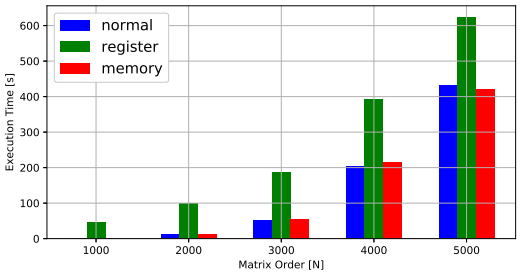

図 1 実行時間比較

5. 今後の課題

今後の課題として、巨大なメモリのダンプや検索にかかる時間を削減するために実行命令を逆方向に辿ることで一意に *NaN* のアドレスを特定と、実際に計算アプリケーションを動作させて誤差の評価を行う。

参考文献

1) *Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman, and Matei Ripeanu. Letgo: A lightweight continuous framework for hpc applications under failures. In* Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, *HPDC '17, pp. 117–130, New York, NY, USA, 2017. ACM.*

2) 穐山空道, 広渕崇宏. 性能カウンタを用いた近似実行の高速なエミュレーション. 情報処理学会研究報告 *(2017-OS-141), pp. 1–9, 2017.*