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
東京農工大学, 産業技術総合研究所

背景

- サーバマシンに搭載されるメモリ容量が増加
  - ➢ 消費電力も増加（メモリが占める割合も増加）
- Approximate Computing
  - 計算精度を落とし**一意性を保証しない**が、消費電力は削減できる ex) DRAMのリフレッシュレートを下げる
  - 繰り返し計算し結果が収束する性質から、物理シミュレーションや人工知能アプリケーションへの適用が期待される
- Approximate Computingの弊害
  - メモリの中身が化ける可能性が高まる
  - **化け方によってはプロセスがクラッシュする**
  - ➢ 先行研究では、レジスタの値が化けて有り得ないアドレスを指してしまった場合での対処について議論されている

目標：数値アプリケーションに対してApproximate Computingを適用した際に、プロセスの異常終了を避けつつ、許容できる誤差範囲で計算を終えられる実行環境

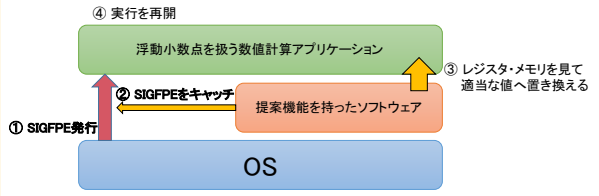
基本アイディア

- DRAMにApproximateComputingを適用し、メモリが化けると想定する
- メモリが化けて浮動小数点がNaNになると、その後の計算結果が数値であることと正常に実行を継続できる保証はできない
  - NaNを含む行列同士の演算はNaNが計算結果へ伝播

$$\begin{pmatrix} 0.6 & 0.1 & 0.8 \\ 0.7 & 0.5 & 0.3 \\ 0.2 & 0.9 & 0.4 \end{pmatrix} \begin{pmatrix} 4 & 9 & 2 \\ 3 & \mathrm{NaN} & 7 \\ 4 & 9 & 2 \end{pmatrix} = \begin{pmatrix} 6 & \mathrm{NaN} & 8 \\ 7 & \mathrm{NaN} & 3 \\ 2 & \mathrm{NaN} & 4 \end{pmatrix}$$

$$\begin{vmatrix} 0.5 & \mathrm{NaN} \\ 2.4 & 1.6 \end{vmatrix} = \mathrm{NaN}$$

- signaling NaNを対象とした演算は例外が発生しOSからSIGFPEが発行される
  - ➢ **SIGFPEをハンドルして、NaNを0もしく適切な値に書き換えれば透過的に実行の継続が可能**
  - ➢ どのような値に書き換えると、誤差を抑えられるかは議論・検討が必要
- 書き換えの対象は2つ
  - レジスタ
    - 例外の直接的な原因は、SSEレジスタ(xmm0やymm1)に存在するsNaNであるため、これを適当な値に置き換える。しかし、一時的な対処である。
  - メモリ
    - メモリが化けることを想定しているため、根本的な原因であるNaNはメモリに存在する。メモリ中にあるNaNを適当な値に置き換えることで、更なる例外の発生を抑える。



実装

- シグナルをキャッチでき、レジスタやメモリを操作できることからgdbを採用した。NaNの検索やメモリのダンプには、gdb pythonを用いた。
  - *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*
  - *gdbによる時間的オーバーヘッドは、シグナルをキャッチした時にのみ発生する*
- gdbの機能で実行アプリケーションに変更を加えずに透過的に実現
  - handle SIGFPE stop print nopassとcatch signalで、アプリにシグナルを渡さずにgdbで割り込むことができる

① SIFGPEハンドル時のコンテキスト

```
   0x5555555549ff <calculate()+79>:     movsd  xmm0,QWORD PTR [r10+rsi*8]
   0x555555554a05 <calculate()+85>:     add    edx,edi
   0x555555554a07 <calculate()+87>:     cmp    eax,r8d
=> 0x555555554a0a <calculate()+90>:     mulsd  xmm0,QWORD PTR [r9+rcx*8]
   0x555555554a10 <calculate()+96>:     addsd  xmm1,xmm0
```

- *レジスタの書き換え*
  1. *gdbシグナルをキャッチした時、命令レジスタは例外を引き起こした命令を指している*
  2. *命令を見てオペランドとなっているSSEレジスタを特定する*
  3. *SSEレジスタの中にあるNaNを探す*
  4. *NaNを0に置き換える*
- *メモリの書き換え*
  1. SIFFPEを引き起こしたNaNのビット列を特定する（レジスタ時と同様）
  2. heapなどの計算用データが配置されていると予想されるメモリ領域をダンプする
  3. ダンプに対してNaNのビット列で検索かけて、位置を特定する
  4. 特定した位置のメモリを0に書き換える

② SIGFPE時のSSEレジスタの中身

```
gdb-peda$ p $xmm0.v2_double
$2 = {nan(0x464544434241), 0}
```

```
gdb-peda$ p $xmm0.v2_int64
$3 = {0x7ff0464544434241, 0x0}
```

評価と将来の展望

- 提案する方法でNaNが混入しても継続して実行が可能なことを確認した（環境は表1の通り）
  - ✓ NaNを仕込んだ正方行列の行列計算でテストした
  - ✓ 実行自体は正常に終了し、計算結果もNaNによる影響を抑えることができた
- 実行時間のオーベーヘッドを計測した（結果は図1）
  - normal: NaNを仕込まない通常の行列計算
  - register: SIGFPE時にレジスタのNaNのみを書き換える
  - memory: レジスタだけでなくメモリにあるNaNも合わせて書き換える
- 今後の展開
  - 実際の数値計算アプリに対してApproximateComputingを適用して、提案手法のテストする
    - ➢ 誤差はどの程度になるか？
    - ➢ 計算時間はどれほど延びるのか？
  - NaNがあるアドレスの特定方法の改善
    - 現状の方法では、メモリのダンプが必要で巨大なデータの場合では資源を無駄遣いする
    - SIGFPEの発生時から、実行命令を逆に辿ることでアドレスを一意に求めたい。多くの場合では、汎用レジスタを用いてmovされるため、レジスタの値を知る必要がある
      - 現在は数命令分だけ遡って、NaNのあるアドレスが特定できる
      - 分岐やループ、関数の引数によって渡された場合の対応は今後の課題である

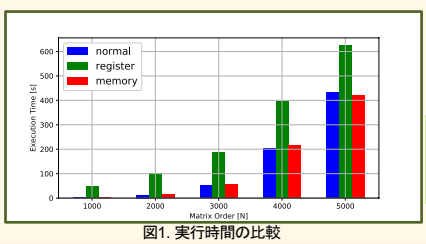


図1. 実行時間の比較

表1. 実行環境

CPU	Intel Xeon CPU E5-2699 v3
OS	Debian 4.9.30-2 GNU/Linux
gdb	Debian 7.12-6

メモリを書き換えることで例外を抑制し、時間的オーバーヘッドを減らすことができる