Vol.2021-ARC-245 No.14
2021/7/21

Approximate Memory におけるエラー混入対象データの重要度の事前推定に関する検討

穐山 空道[1,a)]　塩谷 亮太[1]

概要：CPU 性能の増加に伴う DRAM アクセスレイテンシの相対的な増大が問題となっており、そのためデータに低確率でエラーが入ることを許す代わりに低レイテンシでのアクセスが可能な Approximate Memory が注目されている。Approximate Memory を実用するためには、エラーにより引き起こされるアプリケーションの計算結果の誤差を見積もり、高速化とのトレードオフを制御することが望ましい。デバイスレベルのエラー発生と遅延の関係は広く研究されているものの、エラー発生に伴う最終的なアプリケーションレベルの計算誤差と高速化の関係は未知である。そこで本研究では、Approximate Memory 上の入力データに混入するエラーが計算結果に与える影響を事前推定することを提案する。プログラムを入力データを受け取り計算結果を返す数学的な関数と捉え、自動微分技術を用いて入力データの各チャンクに対し、チャンク内のデータが微少量ずれた場合の計算結果のずれを表す「重要度」を計算する。このための初期検討として、複数のアプリケーションの代表的な入力データの重要度を計算し、アプリケーションごとの重要度のデータ依存性を議論する。実験の結果、多くのアプリケーションでは重要度のデータ依存性は小さく、事前計算した重要度を未知の入力データにも利用できる可能性が高いことを示した。

1. 序論

CPU 性能の増加に伴う DRAM アクセスレイテンシの相対的な増加が問題となっている。アプリケーションから見えるメモリアクセスレイテンシはキャッシュミスであることが確定するまでの時間とその後 DRAM にアクセスする時間からなる。前者はキャッシュの動作周波数に律速されるため CPU コアの性能と類似する割合で改善している。一方後者は製造プロセスが小さくなっても変わらず 20 年以上ほぼ一定である [1, 2]。従ってアプリケーションの性能に与える DRAM アクセスレイテンシの重要性は年々増大している。

Approximate Memory は DRAM の読み出し時に低確率でエラーが入ることを許す代わりにレイテンシを短縮する新たな技術である。Approximate Memory では仕様で定められた読み出し時間よりも短い時間で読み出し操作を行う [2–4]。DRAM の仕様では正しい動作を保証するために大きなマージンが設けられているため、低確率のエラー混入を許せば大幅に短いレイテンシで読み出し可能である。この Approximate Memory は DRAM のアクセスレイテンシを直接に短縮することができるため、レイテンシがアプリケーション性能に与える影響が増大するに従いその重要性が高まっている。

Approximate Memory を実用するには、アプリケーション実行結果の誤差をユーザの望む範囲に抑える技術が望まれる。この技術により、実行結果の誤差を与えられた範囲内に抑えかつ最大限の高速化を得るようにシステムが自動的に Approximate Memory を制御可能になる。例えば画像処理アプリケーションにおいてユーザが許容できるソースノイズ比を指定するとその範囲で最大限エラーを許し高速化するなどが考えられる。

しかし、Approximate Memory のレイテンシをある値に設定した時にその上で実行されるアプリケーションの実行結果の誤差どの程度になるか知るのは難しい。デバイスレベルの研究ではレイテンシとエラー率の相関が調べられているが、これは個々のメモリアクセスに関する知見である。一方でアプリケーションの実行結果の誤差は発生したエラーがデータフローにより伝播し現れる。既存研究では実際にアプリケーションを Approximate Memory 上で実行し結果がある程度に収まっていることを評価するが、実用のためには事前にエラー率と実行結果の誤差の関係を知る必要がある。

そこで本研究では、Approximate Memory 上の入力デー

[1] 東京大学 情報理工学系研究科 創造情報学専攻
Department of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo
[a)] akiyama@ci.i.u-tokyo.ac.jp

タに混入するエラーが計算結果にどの程度影響するかを事前に推定することを提案する。プログラムを入力データを受け取り計算結果を返す数学的な関数と捉え、自動微分技術を用いて入力データの各チャンクの重要度を計算する。入力データの「チャンク」とは Approximate Memory のエラー率を制御できる対象単位の区切り（典型的には 512 バイトから数キロバイト）であり、その「重要度」とはチャンク内のデータが微少量ずれた場合に実行結果に表れる誤差の目安である。このための初期検討として、複数のアプリケーションを対象に代表的な入力データの重要度を計算し、アプリケーションの違いによる重要度のデータ依存性を議論する。本稿では画像処理アプリケーションに注目し、ガウシアンフィルタ、ソーベルフィルタ、バイラテラルフィルタ、離散コサイン変換、ニューラルネットワークによる推論を対象とする。実験の結果、多くのアプリケーションでは重要度のデータ依存性は小さく、事前に計算した重要度を未知の入力データにも利用できる可能性が高いことを示した。

2. Approximate Memory の利用上の課題

2.1 Approximate Memory の概要

Approximate Memory は保持するデータにビット化けが低確率で起こることを許す代わり低レイテンシでのアクセスを実現する。高速アクセスは DRAM 内の電気的動作に定められた時間パラメータを、JEDEC による仕様を満たさない値に設定することで実現される。例えば文献 [2] では、DDR3-1600 の仕様で 12.5 ns に定められた `tRCD` という値を 7.5 ns に削減したときのエラー発生率が特定の DIMM では 10^{-9} 未満であることが報告されている[*1]。

Approximate Memory のエラー発生率は 512 バイトから数キロバイトごとの大きな単位でしか設定できないため、本稿ではこれを念頭にアプリケーションの入力データをチャンクに分割する。DRAM の時間パラメータは物理的構造の制約により DRAM 内の row と呼ばれる構造ごとに制御される。row の大きさは一般的な DRAM では 512 バイトから数キロバイトであるため、このような制約がある。なお row の大きさを 1 バイトなどに小さくすることは転送帯域や実装面積の観点から現実的ではない。時間パラメータの row ごとの制御の詳細やそれにより発生する課題については我々の先行研究 [5] およびその拡張版（`arXiv:2101.10605`）を参照のこと。

2.2 エラーと計算誤差

本稿では Approximate Memory 上のデータに発生するビット化けを「エラー」と呼び、その結果生じるアプリケーションの出力に生じるずれを「計算誤差」と呼ぶ。

[*1] エラー発生確率は DRAM のベンダーごとや製造年ごとに大きく異なる。

エラーの発生確率は使用する DIMM のベンダー、製造年、製造ばらつき、時間パラメータの値等によって決まり、アプリケーションの内のデータフローとは独立である。一方「計算誤差」とはエラーの発生によってアプリケーションの最終的な計算結果（出力変数）に生じるずれを指す。計算誤差は Approximate Memory 上に置かれたデータに発生したエラーが出力変数に伝播することで生じる。計算誤差の大きさはプログラムのデータフローおよびコントロールフローに依存する。例えば 2 つの異なるプログラム $y = 2 * x$ と $y = x * x$ の入力データの同じ位置にエラー発生し x が 0.1 だけずれても、計算誤差は前者では 0.2 、後者では $0.1 * x + 0.01$ と異なる。

Approximate Memory は動画像処理のように結果を人間が目で見たり、数値の絶対値よりもその大小関係が重要なアプリケーションで特に有用である。これは前述の「エラー」と「計算誤差」の関係による。動画像処理では主なデータである画素値にエラーが入ってもアプリケーションのコントロールフローには影響せず、計算誤差は生じるものの異常終了せず実行可能なことが多い。また数値の大小関係が重要なアプリケーションでは数値にエラーが入っても大小関係が入れ替わらなければ計算誤差は生じない。例えば文献 [6] では x264 動画エンコーダで使用するデータを Approximate Memory 上に配置しても結果が大きく劣化しないことが報告され、文献 [7] ではディープラーニングの学習を Approximate Memory 上で行う場合に推定精度の劣化を抑える手法を提案している。また我々の先行研究 [8] では SPEC CPU ベンチマーク 2006 に含まれる離散最適化アプリケーションである mcf の探索対象データの一部を Approximate Memory 上に置き、パラメータによっては大半の実行で正しい結果を返すことを示した。

2.3 課題：計算誤差のバウンド

Approximate Memory を実用するには、アプリケーションの計算誤差を与えられた範囲に収めるようにシステムがエラー率を自動制御することが望ましい。Approximate Memory において直接制御できるのは DRAM の時間パラメータおよびそれと強く相関するエラー発生率であるが、ユーザの関心は計算誤差と高速化の関係である。しかしエラーと計算誤差の関係は具体的なアプリケーションごとに異なるため、ユーザは望む計算誤差から設定すべき DRAM の時間パラメータを容易に推測できない。例えば動画エンコーダを例にとると、ユーザはエンコード結果の SN 比に望む値を指定することはできるが、具体的にその SN 比を実現する DRAM の時間パラメータを指定することは非常に困難である。そこでシステムが自動的に時間パラメータを制御できれば、ユーザは本来の関心事である計算誤差のみを指定すればよくなる。

計算誤差を与えられた範囲に収めるように Approximate

Memory を制御することは我々の知る限り未達成である。文献 [7] では深層学習を Approximate Memory 上で学習しても推論精度を大きく低下させない手法を提案する。本文献ではエラーの事前分布を推定し、その分布を仮定した再学習を行うことで推論精度を高く保つ。本手法は達成される推論精度の高さ観点では有用だが、精度がユーザの指定範囲に収まるよう Approximate Memory を制御することはできない。また文献 [9] では仮想マシンを Approximate Memory を含むサーバで実行する際に可用性を指定された値以上に保証する。可用性は平均故障間隔や平均復旧時間で計測され、これを一定以上にするには現在までの履歴に応じより積極的に Approximate Memory を使うかを決めればよい。これに基づき本文献では Approximate Memory の制御はは簡単なヒューリスティック[*2]で行う。しかしアプリケーションの計算誤差は途中結果から外挿できないことが一般的であり、計算誤差の保証にはこのようなヒューリスティックは利用できない。

3. 計算結果の誤差の事前推定

3.1 基本アイデア

本研究ではエラーが計算結果の誤差に与える影響を事前推定し Approximate Memory の適切な制御を目指す。このためにプログラムを入力データを受け取り計算結果を返す数学的な関数と捉え、計算結果の入力データに関する微分値を利用する。この微分値は入力データが微小な値だけ変わったときの出力のずれであり、入力データに混入するエラーと出力変数の計算誤差を結びつけることできる。

しかし Approximate Memory では第 2.1 章に示したようにエラー率の制御は大きな単位でしか行えず、各変数の微分値が分かってもそれに合わせたエラー率設定はできない。例えば入力データが `double` 型の配列であるとき、`double` 型のサイズは 64 バイトであるため Approximate Memory のビットエラー率制御は配列の 8 ($=\frac{512}{64}$) 要素ごとにしか行えない。そこで本稿では入力データは数値の配列と仮定し、これを Approximate Memory の制御可能な粒度よりも大きいサイズのチャンクに分割する。分割されたチャンクごとに微分値を統合することで Approximate Memory 制御に微分値の利用を可能とする。

プログラムを数学的な関数として微分する技術は自動微分と呼ばれる。プログラム $y = f(x)$ の入力と出力が配列 $x = \{x_i \mid i = 0, 1, ..., N\}$、$y = \{y_j \mid j = 0, 1, ..., M\}$ のとき、自動微分は与えられた具体的な x に関し $\frac{\partial y_j}{\partial x_i}$ を計算する。これは基本的には計算グラフ上の各オペレータに微分規則を適用すればよい。例えばプログラム上のある式 $c = a * b$ は積の微分規則から $\frac{\partial c}{\partial x} = \frac{\partial a}{\partial x} * b + a * \frac{\partial b}{\partial x}$ と変形でき、これを繰り返して $\frac{\partial y}{\partial x}$ を得る。計算グラフを入力からたどるか出力からたどるかによって forward-mode と reverse-mode が存在するが、本研究に関わる部分に違いはない。またプログラム f を変形し $\frac{\partial y}{\partial x}$ を直接計算するプログラム f' を出力する手法も存在する [10, 11]。

[*2] 文献中の式 (4)

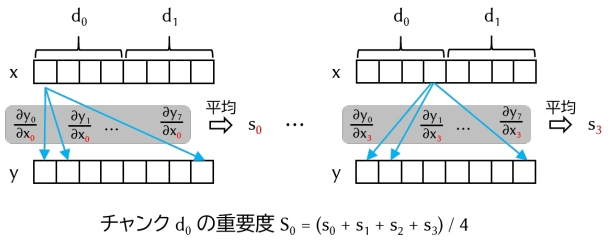

図 1 入力データの重要度の計算例

3.2 入力データの重要度

微分値を Approximate Memory の制御に利用するため、本稿では入力データを数値の配列と仮定し、Approximate Memory の制御可能な粒度よりも大きいサイズのチャンクに分割する。各チャンクごとにチャンク内のデータに関する微分値を統合したものをそのチャンクの**重要度**と呼ぶ。重要度はチャンク内のデータが微少量ずれた場合に実行結果に表れる誤差の目安である。

図 1 に重要度の具体的な計算方法を示す。まず数値の配列である入力データ x を、それぞれのサイズが N のチャンク $d_k = \{x_{kN}, x_{kN+1}, ..., x_{(k+1)N-1}\}$ に分割する。ただし以降では簡単のため x の添字は各チャンクごとに 0 からはじまるとし、$d_k = \{x_0, x_1, ..., x_{N-1}\}$ と書く。また出力変数 y を M 要素の配列と仮定し、$y = \{y_0, y_1, ..., y_{M-1}\}$ と書く。図では x, y ともに 8 要素の配列であり、x をサイズ 4 のチャンクに二分割する。すなわち $N = 4$、$M = 8$ である。チャンク d_k の重要度は以下のように計算される。

(1) 出力 y の各要素 y_i に対し、d_k の各要素 x_j に関する微分値 $\frac{\partial y_i}{\partial x_j}$ を計算する。これは自動微分の枠組みそのものであり、プログラムを一度評価すれば全ての (i, j) の組に関して微分値が求まる。

(2) 各 x_j に関する微分値 $\frac{\partial y_i}{\partial x_j}$ の絶対値の、i に関する平均を s_j とする。すなわち

$$s_j = \frac{1}{M} \times \sum_{i=0}^{M-1} \left| \frac{\partial y_i}{\partial x_j} \right|$$

である。

(3) 各 d_k に対し、その全ての要素 x_j に関する s_j の平均を重要度 S_k とする。すなわち

$$S_k = \frac{1}{N} \times \sum_{j=0}^{N-1} s_j$$

である。

4. 入力データの重要度のデータ依存性

4.1 実行時の重要度計算

入力データの重要度を用いて Approximate Memory を制御するには実行時に与えられる具体的な入力データに対する重要度が必要である。これはプログラムの入力と出力の関係は一定ではなく重要度にデータ依存性がある場合があるからである。例えばプログラムが $y = 2 * x$ のとき $\frac{\partial y}{\partial x} = 2$ であり x の重要度はその具体的な値に依存しない。しかし出力が入力に関して線形でない場合は入力データの重要度にデータ依存性がある。例えば

$$y_0 = 2 * x_1$$
$$y_1 = x_0 * x_1$$

のとき、

$$\frac{\partial y_0}{\partial x_0} = 0, \quad \frac{\partial y_0}{\partial x_1} = 2$$
$$\frac{\partial y_1}{\partial x_0} = x_1, \quad \frac{\partial y_1}{\partial x_1} = x_0$$

となり入力データの重要度はその具体的な値に依存する。

ある特定の入力データの重要度をプログラムの実行時に求めることは計算時間の観点から困難である。forward-mode と reverse-mode のいずれを用いても、$\frac{\partial y}{\partial x}$ の計算には y の計算と同じ時間計算量がかかる。例えば forward-mode では入力 x の微分値を 1 としプログラムの計算順に微分値を求め、これにはプログラム全体の評価が必要である。従って Approximate Memory による高速化のために微分値を求める間に計算結果を直接求めることができ、これは高速化の観点では無意味である。

4.2 データ依存性の大きさ

ある特定の入力データに関する重要度が正確に分からなくとも、重要度のデータ依存性が低ければサンプルデータに対して計算した重要度を使い回せる可能性がある。まず出力が入力に関して線形の場合は入力データの重要度のデータ依存性は 0 であり、常に同じ重要度を利用できる。また出力が入力に関して線形でない場合でも、重要度のデータ依存性が小ければ事前に特定の x で計算した重要度と実行時の入力データの重要度がほぼ同じであると仮定し Approximate Memory を制御できる可能性がある。

そこで本稿では、入力データの重要度のデータ依存性を複数のアプリケーションに関して計算し、重要度のデータ依存性がどの程度あるかを調査する。調査によって微分値を用いた Approximate Memory 制御のために有用な知見を導くことを目的とする。Approximate Memory を微分値を用いて制御する試み自体が我々の知る限り初めてであるため、どのようなアプリケーションで微分値のデータ依存性が大きい・小さいのかも現状は明らかではない。本稿ではこれを明らかにすることで、Approximate Memory を実際に制御する手法の初期検討とする。

4.3 対象アプリケーション

本稿では画像処理分野のアプリケーションを対象に入力データの重要度のデータ依存性を計測する。画像処理アプリケーションでは結果を人間が目で見たり、例えば画像に写っているものを判別するなど結果から大局的な情報のみを取得することが多い。このようなケースでは計算結果に多少の誤差があっても問題なく、Approximate Memory 上にデータを置くことが適している。具体的に対象とするアプリケーションはガウシアンフィルタ、ソーベルフィルタ、バイラテラルフィルタ、離散コサイン変換、簡単なニューラルネットワークによる推論、の 5 種類である。

4.3.1 ガウシアンフィルタ

画像を平滑化することでノイズを除去する処理である。入力画像に対してサイズ $k \times k$ のフィルタ K を 2 次元畳み込みした結果を出力画像とする。フィルタの各要素の値はガウシアンフィルタではガウシアン分布に従う。本稿では $k = 3$ とし、フィルタ K は以下を用いる。

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

ガウシアンフィルタでは出力画素値は入力画素値に関し線形であり、入力データの重要度のデータ依存性は 0 であると予想される。

4.3.2 ソーベルフィルタ

画像内の変化が大きい部分のみを強調することでエッジを抽出する処理である。入力画像に対してサイズ $k \times k$ のフィルタ K を 2 次元畳み込みした結果を出力画像とする。フィルタの各要素の値は事前に決まっており、縦方向のエッジを抽出するフィルタと横方向のエッジを抽出するフィルタが定義されている。本稿では $k = 3$ とし、縦方向のエッジを抽出するフィルタ K として以下を用いる。

$$K = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

ソーベルフィルタでは畳み込み結果は入力画素値に関して線形だが、出力画素値は入力画素値に関して必ずしも線形とは限らない。これは畳み込み結果が画素値としてあり得る範囲を越える場合にクロップする処理が必要だからである。例えば K は負の値を含むため畳み込み結果は負になりえるが、この場合は出力画素値は 0 にする必要がある。

4.3.3 バイラテラルフィルタ

画像を平滑化することでノイズを除去しつつ、画像内のエッジは保持する処理である。フィルタサイズを $k \times k$、入力画像の第 (i,j) 画素値が $x_{i,j}$、出力画像の第 (i,j) 画素値が $y_{i,j}$ のときの入出力の関係は以下の通りである。

$$y_{i,j} = \frac{\sum_{l=-k}^{k}\sum_{m=-k}^{k}(x_{i+m,j+l} \times w(i,j,l,m))}{\sum_{l=-k}^{k}\sum_{m=-k}^{k} w(i,j,l,m)},$$

$$w(i,j,l,m) = \exp\left(-\frac{m^2+l^2}{2\sigma_1^2}\right) \times \exp\left(-\frac{(x_{i,j}-x_{i+m,j+l})^2}{2\sigma_2^2}\right)$$

ただし σ_1, σ_2 はフィルタの感度を決定する定数である。

バイラテラルフィルタでは入力画素値同士の積が表れるため、入力画素値と出力画素値の関係が線形ではなく入力データの重要度にはデータ依存性がある。

4.3.4 離散コサイン変換

画素値をコサイン関数の線形和として表すことで画像を周波数領域に変換する処理である。画像処理では画像を $N \times N$ のブロックに区切り、各ブロック内で離散コサイン変換を適用することが一般的である。入力画像の大きさが $h \times w$ のとき出力は $h \times w$ の行列である。各ブロックの左上を $(0,0)$ として、出力の第 (i,j) 要素 $y_{i,j}$ と入力画像 x の関係は以下で与えられる。

$$y_{i,j} = \alpha(i,j) \times \sum_{k=0}^{N-1}\sum_{l=0}^{N-1} x_{k,l}\cos\left(\frac{\pi(2k+1)i}{2N}\right)\cos\left(\frac{\pi(2l+1)j}{2N}\right)$$

ただし $\alpha(i,j)$ は i、j のみに依存する係数である。

離散コサイン変換では出力が入力の線形和で表されるため、入力データの重要度のデータ依存性は 0 と予想される。

4.3.5 ニューラルネットワークによる推論

あるニューラルネットワークに関して学習済みの重みとテストデータを読み込み推論する処理である。本稿では簡単な例として Fashion_MNIST データセット [12] を用い、ネットワーク構成は以下の 2 層とする。なお第 1 層の入力次元数 784 はデータが縦 28 ピクセル、横 28 ピクセルのグレースケール画像であることから、第 2 層の出力次元数 10 は分類するべきクラス数が 10 であることから決まる。

(1) 入力次元数 784、出力次元数 128、活性化関数 relu の全結合層
(2) 入力次元数 128、出力次元数 10、活性化関数 softmax の全結合層

一般にニューラルネットワークが全結合層のみを持ちかつ活性化関数が恒等関数のみの場合、そのネットワークの推論処理では入力データの重要度のデータ依存性は 0 と予想される。活性化関数が恒等関数のとき、i 番目の全結合層の入力、重み、バイアス、出力をそれぞれ x_i, A_i, b_i, y_i（ただし x_i, b_i, y_i はベクトル、A_i は行列）と書くと $y_i = A_i x_i + b_i$ である。ネットワーク全体の入力 x_1 と最終段の出力 y_n の関係は、これを繰り返し適用して、

$$\begin{aligned} y_n &= A_n x_n + b_n \\ &= A_n y_{n-1} + b_n \\ &= A_n(A_{n-1}x_{n-1} + b_{n-1}) + b_n \\ &= A_n A_{n-1} x_{n-1} + (A_n b_{n-1} + b_n) \\ &= \cdots \\ &= \left(\prod_{i=1}^{n} A_i\right) x_1 + \sum_{k=1}^{n}\left(\prod_{i=k+1}^{n} A_i\right) b_k \end{aligned}$$

となる。x_1 の係数と第二項はともに x_1 によらないため、結局 y_n は x_1 に関して線形である。従って入力データの重要度のデータ依存性は 0 と予想される。

一方で活性化関数が恒等関数ではない場合は入力データの重要度のデータ依存性は 0 とは限らない。例えば今回使用する relu 関数では入力が 0 より小さい場合は出力は常に 0 である。従って出力が入力に関して線形でないため微分値は定数ではない、すなわち重要度にはデータ依存性があると予想される。

5. 実験

5.1 自動微分の実装

自動微分には kv ライブラリ[*3]に含まれる `autodif.hpp` を用いる。本ライブラリは通常の値と微分値を同時に持つ型（`kv::autodif<T>`、T は `double` のような数値型）を用い通常の値の計算と同時に微分値を計算する。実験ではまず各アプリケーションを `double` 型で計算を行うように実装し、それを `kv::autodif<double>` に置き換える。`double` 型を用いた場合の計算結果と `kv::autodif<double>` 型を用いた場合の計算結果の通常の値（微分値ではない値）が一致することを確認した。

一般に条件分岐を含む関数は微分不能なため kv ライブラリではサポートされないが、ソーベルフィルタにおけるクロップ処理やニューラルネットワークの推論における relu 関数の実現には条件分岐が必要である。そこで本稿では `autodif.hpp` を改変しこれに対処する。具体的には入力が微少量変化してもコントロールフローは変化しないと仮定し微分値を近似する。これを実現するため、T と `kv:autodif<T>` を比較する際には微分値を無視し両者の値の比較結果を返すよう改変する。この近似はニューラルネットワークにおいて微分不能な関数に back-propagation を適用可能にする手法と本質的に同じである。

[*3] `http://verifiedby.me/kv/`

5.2 実験設定

第 4.3 章にあげたアプリケーションを cpp で実装し実験を行う。全てのアプリケーションにおいて、重要度を計算する単位であるチャンクサイズは画像の 1 行と同じする。画像の 1 行は Approximate Memory のエラー率制御が可能な最低サイズ（典型的には 512 バイトから 4 キロバイト）より小さいが、今回は簡単のためこのように設定する。

それぞれのアプリケーションについて個別の実験設定を以下に示す。

ガウシアンフィルタ、ソーベルフィルタ、離散コサイン変換
対象画像は cifar10[*4] データセットの全 10 クラスからそれぞれファイル名順に 64 枚を取り 640 枚とする。画像は 32×32 ピクセルであり、カラー画像を 255 段階グレースケール画像に変換し用いる。

バイラテラルフィルタ 対象画像は ImageNet[*5] データセットの小規模版である TinyImageNet の全 200 クラスからそれぞれファイル名順に 3 枚を取り 600 枚とする。画像は 64×64 ピクセルであり、カラー画像を 255 段階グレースケール画像に変換し用いる。ガウシアンフィルタ、ソーベルフィルタ、離散コサインと異なる画像を用いる理由は、バイラテラルフィルタに一般的なフィルタサイズである 5×5 や 9×9 では 32×32 の画像では小さすぎるからである。重要度を計算する単位であるチャンクサイズは画像の 1 行と同じとする。またフィルタサイズは 5×5、パラメータは $\sigma_1 = 3, \sigma_2 = 0.3$ とする。フィルタの適用結果を目視しフィルタの効果がある程度現れている値を選んだ。

ニューラルネットワークの推論処理 ネットワークの学習は Fashion_MNIST の学習データである 6 万画像、入力データの重要度の計算には Fashion_MNIST のテストデータである 1 万画像を用いる。画像は 28×28 のグレースケール画像である。学習は TensorFlow で行い、エポック数は 5、学習アルゴリズムは Adam、ハイパーパラメータは全て Adam のデフォルト値を用いる。テストデータの正答率は 84.47 % であった。cpp で実装した推論プログラムは TensorFlow で学習した重みと入力画像を読み込み推論し、推論結果は TensorFlow の推論結果と一致することを確認した。

5.3 実験結果

実験結果では入力データの重要度のデータ依存性として、入力データを変えたときの各チャンクの重要度およびその変動係数を測定する。変動係数は標準偏差を平均値で割ったものである[*6]。値の集合が 2 つあるとき、それぞれの標準偏差は元の値の大小によって大きさが変わり直接比較できないため、平均値で割り正規化する。

5.4 ガウシアンフィルタ

表 1 ガウシアンフィルタの入力データの重要度とその変動係数

チャンク番号	0, 31	1, 30	2 – 29
重要度	2.289E-4	6.866E-4	9.155E-4
変動係数	0	0	0

表 1 はガウシアンフィルタプログラムの入力データの各チャンクの重要度とその変動係数である。値は実験設定で 640 枚の画像についての平均である。

重要度の変動係数は全てのチャンクで 0、すなわち重要度の入力データ依存性はなかった。4.3.1 で示したように、ガウシアンフィルタでは出力画素値は入力画素値に関して線形であるため、出力の入力に関する微分値は一定である。従ってガウシアンフィルタでは事前にサンプルデータで計算した重要度をサンプルと異なるデータに対しても利用できると考えられる。

重要度はチャンク 0 と 31、チャンク 1 と 30、またチャンク 2 から 29 が同じ値であった。これは画像の端では畳み込み演算が一部できずにスキップされることが理由である。今回のフィルタは大きさが 3×3 であるため上から 3 行目と下から 3 行目の間（すなわちチャンク 2 から 29）ではフィルタの全ての値と畳込み演算が可能である。一方それより画像の端に近い部分ではフィルタの一部と畳込み演算ができないため、出力に与える影響が小さくなり重要度も小さくなる。

5.5 ソーベルフィルタ

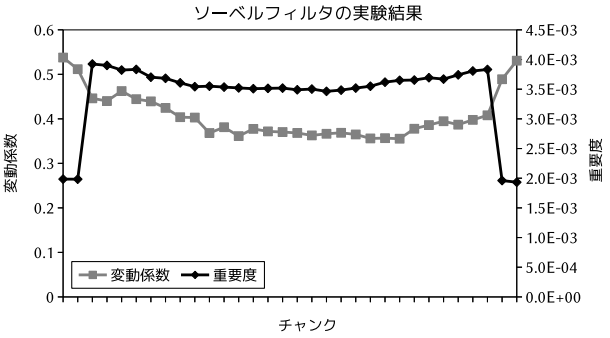

図 2 ソーベルフィルタの入力データの重要度とその変動係数

図 2 はソーベルフィルタプログラムの入力データの各チャンクの重要度とその変動係数である。横軸はチャンクの番号、縦軸は第 1 軸（左）が変動係数、第 2 軸（右）が重要度であり、縦軸の値は実験設定で 640 枚の画像についての平均である。結果は画像の上下の 2 行に対応するチャ

[*4] https://www.cs.toronto.edu/~kriz/cifar.html
[*5] https://image-net.org/
[*6] https://en.wikipedia.org/wiki/Coefficient_of_variation

ンク（チャンク 0、1、30、31）はその他の行に対応するチャンク（チャンク 2 から 29）と異なる傾向がある。これはガウシアンフィルタと同様に画像の上下 2 行は畳込み演算が一部スキップされるためである。

重要度の変動係数はデータ依存性があり、その値は非常に大きい。変動係数は全体として 0.5 程度で、これは標準偏差が平均値の半分程度あるということである。従ってソーベルフィルタでは事前にサンプルデータで計算した重要度をサンプルと異なるデータに対して利用することは難しいと考えられる。

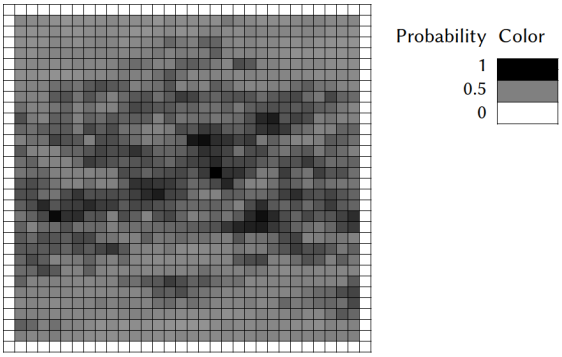

図 3 画素値が 0 にクロップされる確率。図中の四角 1 つが 1 画素に対応し、色の濃さが確率を表す。画像の中央に近いほど確率が高い。

重要度はチャンク 2 から 29 では画像の中央に近いチャンクほど低くなっている。これは画像の中央に近いほど畳み込み演算の結果が負の値になりやすく、その結果微分値が 0 になりやすいことが理由である。畳み込み演算の結果が負のとき出力画素値は入力画素値に値に関わらず 0 であり、入力画素値がわずかに変化したときの出力画素値の変化である微分値も 0 である。**図 3** に出力の画素値が 0 にクロップされる確率をヒートマップで表す。図のそれぞれの四角が画素に対応し、その色が 0 にクロップされる確率である。確率は全画像データ 640 枚についての平均である。画像の中央に近づくほど確率が高くなっている。

5.6 バイラテラルフィルタ

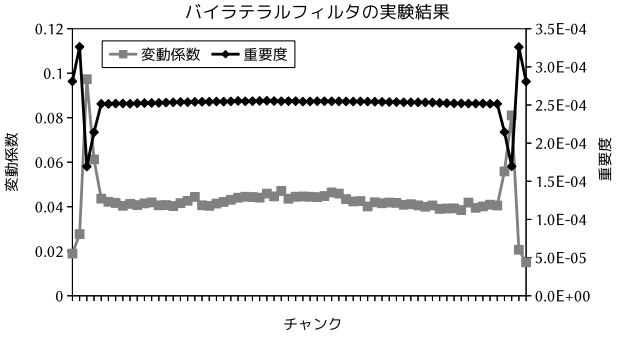

図 4 バイラテラルフィルタの入力データの重要度とその変動係数

図 4 はバイラテラルフィルタプログラムの入力データの各チャンクの重要度とその変動係数である。横軸はチャンクの番号、縦軸は第 1 軸（左）が変動係数、第 2 軸（右）が重要度であり、縦軸の値は実験設定で 600 枚の画像についての平均である。結果は画像の上下の 4 行に対応するチャンク（チャンク 0 から 3、チャンク 60 から 63）はその他の行に対応するチャンク（チャンク 4 から 59）と異なる傾向がある。これはガウシアンフィルタ・ソーベルフィルタと同様の理由で、フィルタサイズが 5×5 なので画像の上下 4 チャンクは一部の計算をスキップするためである。

重要度の変動係数は 0 ではなく重要度にはデータ依存性があるが、その変動は小さい。変動係数の最大値は 0.0973（チャンク 2）、最小値は 0.0150（チャンク 63）、全チャンクにわたる平均は 0.0428 であった。従ってバイラテラルフィルタでは事前にサンプルデータで計算した重要度をサンプルと異なるデータに対しても利用できる可能性が高いと考えられる。

5.7 離散コサイン変換

表 2 離散コサイン変換の入力データの重要度とその変動係数

チャンク番号	0 - 31
重要度	6.816E-3
変動係数	0

表 2 は離散コサイン変換プログラムの入力データの各チャンクの重要度とその変動係数である。重要度は全てのチャンクで同じで、変動係数は全てのチャンクで 0 であった。

重要度の変動係数は、4.3.4 に示したように入力と出力の関係が線形であることから 0 になる。従って離散コサイン変換では事前にサンプルデータで計算した重要度をサンプルと異なるデータに対しても利用できると考えられる。

重要度が全てのチャンクで同じ理由は以下である。第 (m,n) 番目の入力画素値に関する第 (i,j) 番目の出力値の微分 $\frac{\partial y_{i,j}}{\partial x_{m,n}}$ は、4.3.4 の式の $\sum$ 演算において $k=m, l=n$ の部分のみ残るので、

$$\frac{\partial y_{i,j}}{\partial x_{m,n}} = \alpha(i,j) \times x_{m,n} \times \cos\left(\frac{\pi(2m+1)i}{2N}\right) \cos\left(\frac{\pi(2n+1)j}{2N}\right)$$

となる。重要度の計算でこの式から (i,j) および (m,n) への依存性がなくなる理由を説明する。まず (i,j) については全ての (i,j) に関して平均されるため依存性はなくなる。次に (m,n) に関する依存性は、m を画像の縦方向、n を横方向とすると、

(1) n に関する依存性はチャンク内で平均されることでなくなる。チャンクサイズ（画像の 1 行と同じ）は 32 で

これはブロックの一辺の長さ 8 より大きいため、チャンクには全ての n に関する微分値が入っている。

(2) m に関する依存性はチャンクサイズがブロックの一辺の長さの定数倍であることでなくなる。m は 0 から N までコサインの半周期分を動くが、チャンクサイズがこの定数倍なので結局微分値は画像の全ての行で同じ数列になる。

5.8 ニューラルネットワークの推論

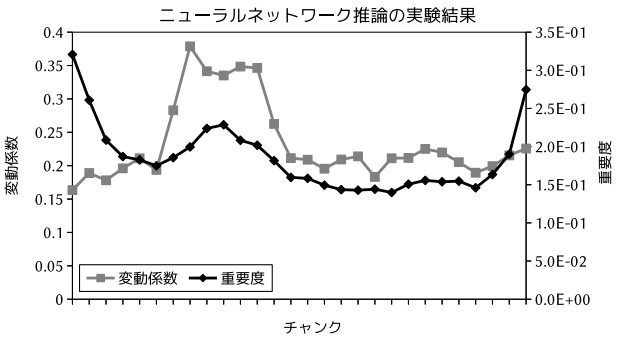

図 5 ニューラルネットワーク推論の推論の入力データの重要度とその変動係数

図 5 はニューラルネットワークの推論プログラムの入力データの各チャンクの重要度とその変動係数である。横軸はチャンクの番号、縦軸は第 1 軸（左）が変動係数、第 2 軸（右）が重要度であり、10000 枚の画像についての平均である。プログラムの出力は第 2 層で softmax をかける前の値とした。softmax は exp 関数を用いて正規化することで複数の値の順位は変えずに合計を 1 にする処理であり、わずかな値の違いが拡大されるためである。

重要度の変動係数はソーベルフィルタよりは小さいもののバイラテラルフィルタよりは数倍程度大きい。ニューラルネットワークの推論ではこれまで実験したアプリケーションと違い出力の値そのものではなく出力ベクトルに含まれる値同士の大小関係のみが重要であるため、この値が大きいか小さいか直接判断することは難しい。これについて詳しくは第 6.1 章を参照のこと。

重要度はチャンク 6 から 12 でその他のチャンクより明らかに大きくなっている。これがネットワークの特性によるのか、またはソーベルフィルタの実験結果で分析したように画像の特性によるのかの分析は今後の課題である。

6. 議論

6.1 ニューラルネットワークにおける変動係数の定量評価

ニューラルネットワークの推論では、重要度の変動係数が計算誤差に与える影響を調べることが他のアプリケーションより難しい。これは重要度の変動係数と計算誤差を結びつけることが難しいことに由来する。

```
1  kv::autodif<double> y0, y1;
2
3  // do inference
4
5  if (y0 > y1)
6      p = 0;
7  else
8      p = 1;
9
10 return p;
```

図 6 2 クラス分類のニューラルネットワーク推論プログラムの例

本稿で計算したニューラルネットワークの推論における入力データの重要度の変動係数は、計算誤差の変化を直接表していない。ニューラルネットワークの推論においてユーザが興味がある計算誤差は推論精度であり、例えば Approximate Memory を使わない場合に比べて 0.1% 以内の精度低下に抑えるなどがありえる。一方本稿で計算した重要度が表すのはネットワークの最終層の出力が数値としてどの程度変わるかであり、推論精度の変化量ではない。今回のような分類問題では推定結果は最終層のノードのうち出力が最も大きいもののインデックスである。従って最終層の変化量が大きくてもノードの出力値の大小が変わらなければ推論精度に影響はなく、逆に変化量が小さくても大小関係が大きく変われば推論精度は大きく劣化する。

ニューラルネットワークの推論精度を入力に関して直接微分することは本稿の枠組みでは意味がない。**図 6** は分類すべきクラス数を 2、最終層の出力を `y0` および `y1` としたときの推論結果を `p` を返すプログラム例である。本稿の枠組みでは `if (y0 > y1)` は `y0` と `y1` に含まれる微分値を無視して通常の値のみ比較する。従って `p = 0` ではそれまで計算した微分値と関係なく単に `p` に 0 が代入され微分値は消えてしまう。

この問題は 2 つの変数の微分値とそれらの大小関係が入れ替わる確率を結びつけることができれば解決できる可能性がある。しかしこれには直感的には変数の値そのものが必要である。Approximate Memory の制御では変数の真の値は完全には分からず、この条件のもとでの検討は今後の課題である。

6.2 重要度を用いた具体的な制御方法

入力データの重要度を用いた Approximate Memory の具体的な制御方法は検討中である。重要度は微分値の平均であるため、重要度にエラー率をかけたものがメモリアクセス 1 回あたりの計算誤差の期待値となる。しかし単純にこの値を利用することは以下の課題により難しい。

第一に、メモリアクセス 1 回あたりの計算誤差の期待値からプログラム全体の計算誤差を見積もることは単純ではない。これはソフトウェアから見たメモリアクセスの多くはキャッシュヒットし DRAM に到達しないからである。

我々は文献 [8] でプログラム実行時にあるメモリ領域に発生する DRAM の内部操作回数を計測しており、この手法の応用で本問題を解決できる可能性がある。

第二に、出力の中間変数に関する微分値はエラーにより中間データが変わると変化してしまう。本稿では微分値を計算する入力データはプログラム全体の入力データと同じとした。しかし実行時に制御パラメータを決めるためには、例えばループのイテレーションごとの入力データに関し微分値を計算することが有用と考えられる。ループのあるイテレーションの入力データはそれ以前にエラーが発生すると変化するため、微分値にデータ依存性がある場合には微分値も変わってしまう。

7. 関連研究

文献 [13] は本稿と同様にプログラムの自動微分を用いて変数の重要度を定義し、Approximate Computing を適用する変数を決定する。本稿と本文献の違いは以下の 2 点である。(1) 本文献でも複数の入力データの取りうる範囲を入力とし、微分値の取りうる範囲を事前計算する。しかしアプリケーションごとの微分値の範囲の大小は議論しておらず、事前計算した範囲が実行時の微分値をどの程度近似できるのかは不明である。(2) 本文献では変数ごとに重要度を算出する。よって Approximate Computing の適用には変数ごとに計算の正確さ切り替えるためにプログラムの大幅な変更が必要である。一方本稿では複数のデータをまとめた重要度を計算することで、プログラム変更なしに Approximate Memory を制御可能にすることを目指す。

文献 [14] は Approxmate Computing でエラー混入を許す変数を発見するため、ユーザが指定した変数の相対的な重要度を計算する。ある実行パスにおける変数の重要度はその変数が出力に与える影響の線型回帰とその変数に適用される演算子の重みから計算される。最終的にある変数の重要度は出力変数に至る複数のパスをシンボリック実行で統合し得られる。本文献は変数の「相対的な」重要度を計算する点、単体の変数が対象で Approximate Memory には直接適用できない点が本研究と異なる。相対的な重要度は変数同士で比較し値が低い変数に Approxmate Computing を適用するために使われ、その絶対値には意味がない。一方本稿では重要度の値自体の変動を Approximate Memory の制御に利用することが目標である。

プログラム自動微分については文献 [10, 11] が詳しい。利用可能な実装としては本稿で用いた kv ライブラリ以外に Python 向けの自動微分ライブラリ[*7]が存在する。また自動微分はニューラルネットワークの学習において盛んに利用されている技術である。本稿ではプログラム自動微分は既存技術として利用しそれ自体の改良は不要である。

[*7] https://github.com/google/jax

8. 結論

Approximate Memory を実用化するため、計算結果の誤差をユーザの許容範囲に抑えた上で最大限の高速化を得る制御手法が必要である。入力データのずれと出力のずれの関係であるプログラムの微分値の利用が有望だが、微分値の実行時の計算は計算時間の観点から困難である。そこで本稿で複数のアプリケーションの代表的な入力に対し実際に微分値を計算し、アプリケーションによってはデータごとの微分値の変動が小さいことを示した。このようなケースでは事前計算した微分値を他の入力データに対する Approximate Memory 制御にも応用できる可能性が高い。

謝辞 本研究は、JST、ACT-I、JPMJPR18U1 の支援を受けたものである。

参考文献

[1] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006).

[2] Chang, K. K., Kashyap, A., Hassan, H., Ghose, S., Hsieh, K., Lee, D., Li, T., Pekhimenko, G., Khan, S. and Mutlu, O.: Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization, *International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*, pp. 323–336 (2016).

[3] Hassan, H., Pekhimenko, G., Vijaykumar, N., Seshadri, V., Lee, D., Ergin, O. and Mutlu, O.: ChargeCache: Reducing DRAM latency by exploiting row access locality, *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 581–593 (2016).

[4] Zhang, X., Zhang, Y., Childers, B. R. and Yang, J.: Restore truncation for performance improvement in future DRAM systems, *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 543–554 (2016).

[5] Akiyama, S. and Shioya, R.: The Granularity Gap Problem: A Hurdle for Applying Approximate Memory to Complex Data Layout, *International Conference on Performance Engineering (ICPE)*, pp. 125 – 132 (2021).

[6] Stazi, G., Adani, L., Mastrandrea, A., Olivieri, M. and Menichelli, F.: Impact of Approximate Memory Data Allocation on a H.264 Software Video Encoder, *High Performance Computing. ISC High Performance 2018. Lecture Notes in Computer Science*, pp. 545–553 (2018).

[7] Koppula, S., Orosa, L., Yağlıkçı, A. G., Azizi, R., Shahroodi, T., Kanellopoulos, K. and Mutlu, O.: EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM, *International Symposium on Microarchitecture (Micro)*, p. 166–181 (2019).

[8] Akiyama, S.: A Lightweight Method to Evaluate Effect of Approximate Memory with Hardware Performance Monitors, *IEICE Transactions on Information and Systems*, Vol. E102-D, No. 12, pp. 2354–2365 (2019).

[9] Tovletoglou, K., Mukhanov, L., Nikolopoulos, D. S. and Karakonstantis, G.: HaRMony: Heterogeneous-Reliability Memory and QoS-Aware Energy Manage-

ment on Virtualized Servers, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 575–590 (2020).

[10] Griewank, A. and Walther, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA (2008).

[11] Naumann, U.: *The Art of Differentiating Computer Programs*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA (2012).

[12] Xiao, H., Rasul, K. and Vollgraf, R.: Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, *arXiv:1708.07747*, pp. 1 – 6 (2017).

[13] Vassiliadis, V., Riehme, J., Deussen, J., Parasyris, K., Antonopoulos, C. D., Bellas, N., Lalis, S. and Naumann, U.: Towards Automatic Significance Analysis for Approximate Computing, *International Symposium on Code Generation and Optimization (CGO)*, p. 182–193 (2016).

[14] Metwalli, S. A. and Hara-Azumi, Y.: SSA-AC: Static Significance Analysis for Approximate Computing, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 24, No. 3 (2019).