Vol.2021-HPC-181 No.2
2021/9/27

演算精度の動的制御による Approximate Computingの実現に向けた予備評価

和田 康孝[1,a)]　小林 諒平[2]　坂本 龍一[3]　森江 善之[4]

概要：演算精度と実行性能あるいは消費電力等とのトレードオフを最適化する Approximate Computing 技術が浸透し始めている．Approximate Computing 技術を活用することで，アプリケーションを実行する際に，必要十分な精度の演算結果を得つつも，実行性能の最大化や消費電力の削減を可能とすることができる．今後さらにその効果を拡大させるためには，GPGPU や FPGA などのアクセラレータを搭載したシステムや，構成が異なるノードを複数台接続することで構成されるシステムなど，様々な状況に即して Approximate Computing を適用する必要がある．特に，アプリケーション実行時に，アプリケーションの構造やシステムの状況に応じて，動的に演算精度を調整することが重要となると考えられる．このような背景から，本稿では，アプリケーション実行時に動的に演算精度を変更・調整することを想定し，これをアプリケーションのレベルで適用した際の実行性能と演算結果への影響・トレードオフを評価する．

1. はじめに

半導体集積技術の進歩によるコンピュータシステムの性能向上が鈍化して以降，特に汎用システムの性能向上は，システムに搭載するプロセッサコア数やノード数の向上，またそれを利活用するための並列分散処理手法の向上によるところが大きい．そのほか，メモリ階層の利用最適化や，ノード間の通信削減など，様々な手法を組み合わせて適用することにより，絶えず，アプリケーション実行性能の向上を目指す取り組みがなされている．

しかし，コンピュータシステムに対する要求は止まるところを知らず，アプリケーションの実行性能のみならず，他の様々な要素も含めた性能向上が求められるようになってきた．特に HPC システムにおいては，供給・利用可能な消費電力の上限を考慮してシステムの運用をする必要が出てきており，それを考慮してアプリケーションの最適化を行う必要がある．また，深層学習のように，半精度あるいはそれ以下の精度のデータを多く扱うものから，科学技術計算のように，倍精度あるいはそれ以上の演算精度を必要とするものまで，対象とするアプリケーションの種類も，従来より格段に多くなっている．

従来の，アプリケーションの実行性能，つまり実行に要する時間のみを削減する最適化技術のみでは，このような要求に応えることは難しい．そこで，上記のような様々な要求を考慮し，その間のトレードオフを最適化することで，ユーザの要求に応えることのできるような仕組みが必要となる．そのためのひとつの考え方として，Approximate Computing（以下 AC）技術が近年浸透し始めている．

AC を活用し，アプリケーションを実行する際の演算精度を適切に制御することで，実行性能を向上させたり，消費電力を削減することができる．一方で，近年では GPGPU や FPGA など様々なアクセラレータデバイスを備えるシステムも多く用いられるようになっており，さらにその関係は複雑なものとなっている．また，様々な特性を持つアプリケーションへの適用を考えると，そのトレードオフの関係を単純に見積もることは難しい．アプリケーションごとの特性の違いはもちろんのこと，単一のアプリケーション内においても，その構造や演算内容によって，トレードオフの最適点は異なることが十分に考えられる．

そのため，アプリケーションの構造に応じて，その実行時に動的に演算精度を調整し，演算精度と実効性能のトレードオフを最適化する AC 手法が必要となる．本稿では，これをアプリケーションのレベルで適用した際の実行性能と演算結果への影響・トレードオフを評価する．

本稿の構成は以下のとおりである：まず，2 章にて，AC に関連する先行研究等について概観し，著者らが目指す演算精度の動的制御についてその概要を述べる．3 章では，演算精度の動的制御の実現に向けて重要となる，アプリ

[1] 明星大学
[2] 筑波大学 計算科学研究センター
[3] 東京工業大学
[4] 帝京大学
[a)] yasutaka.wada@meisei-u.ac.jp

ケーションの構造と演算精度がどのように実行性能に影響を与えるかという点について，予備的な評価を実施する．最後に，4 章で全体をまとめ，今後の方針について述べる．

2. Approximate Computing

演算精度を調整することにより，アプリケーションの実行性能と消費電力等の間でトレードオフを最適化する Approximate Computing（AC）は，システムの様々なレイヤで取り組みが行われている [1], [2].

2.1 関連研究

ハードウェアのレイヤにおいては，例えばアーキテクチャ上の工夫によって AC を適用するもの [3]，さらにロジックとアーキテクチャの連携によって効果を高めようとするもの [4] などがある．また，深層学習などの特定の分野のアプリケーションにおいて，処理結果を許容される範囲に留めつつ，処理性能を向上させる取り組みも存在する [5].

ソフトウェアやアプリケーションのレイヤにおいては，例えば，画像処理・音声処理においては，データを表現するビット数を削減したり，固定小数点数を用いて表現するなどの方法で，リアルタイム性能を高める手法は古くから取られており，それがハードウェアへの実装のしやすさや性能にもつながっている．また，広く用いられている単精度・倍精度のように規格として統一された浮動小数点数を用いず，アプリケーションの要求に応じて，仮数部や指数部のビット数を調整する方法も提案されている [6].

HPC システムで特に重要となるノード間ネットワークを対象としたものとしては，多少の誤りであれば放置してしまうネットワーク [7] も，Approximate Computing の一手法として考えることができる．

そのほか，ソフトウェアとハードウェアの特性を併用し，より効率よく AC を適用しようとするものも提案されている．例えば，アプリケーション内で繰り返し実行される処理の入出力を記憶しておき，ある程度の誤差を許容しつつそれを再利用することで性能を向上させる手法 [8], [9] や，FPGA の特性を利用して，より柔軟に浮動小数点数の精度を変更できるようにする取り組み [10], [11] も行われている．

しかし，その多くは，対象アプリケーションの特性として，元来高い演算精度を必要としないことがわかっている分野，つまり画像処理や深層学習などのアプリケーションを対象としたものであり，より適用範囲を拡げ，様々なアプリケーションで AC を利活用できるようにしていく必要がある．

2.2 演算精度の動的制御

2 章でも述べたとおり，Approximate Computing（AC）は様々なレイヤで様々な手法が提案されており，幅広い取り組みがなされている．しかし，今後さらに適用範囲を拡充し，様々なアプリケーションに対してその効果を発揮させていくためには，さらなる取り組みが必要となる．特に，世の中の多くのアプリケーションは，その実行を通して常に一定の特性・性能を示すものは少なく，通常は，それぞれの演算内容やその対象に応じて，常に変化する．また，アプリケーション内のデータの流れや依存関係によって，プログラム上の構造やデータ量等の特性が一致している場合でも，演算精度と性能の間のトレードオフの最適点が異なる場合も考えられる．そういったアプリケーションに対して，その実行全体を通して一定の方法で演算精度の調整を行ってしまうと，アプリケーションに内在する特性と相反する箇所が出てきてしまい，結果としてその効果を減じてしまう．

そこで，プログラムの構造やデータの依存関係等，実行の状況に応じて，動的に演算の精度を調整・最適化する必要が出てくる．しかし，プログラムの実行時に動的に演算精度を調整するためには，調整前後で演算対象のデータの形式を都度変更しなくてはならず，この手続きがオーバヘッドとして性能向上を阻害する可能性もある．この場合の演算精度調整においては，例えば，

- 演算時にデータ型を変換（キャスト）して用いる
- 演算精度を調整する前後で変換してメモリ上に格納する
- ノード間の通信等の過程でデータ型を変更する

といった方法が考えられるが，システムソフトウェアやライブラリ等のレイヤで実現する必要がある，アプリケーションレベル（ユーザレベル）では煩雑でプログラマビリティが低下する，変換そのものがオーバヘッドとなりうる，など，それぞれ課題を抱えている．このような課題を解決しつつ，より幅広く効果的に AC を適用し，演算精度の動的制御を可能とするためには，図 1 に示すように，演算精度と演算性能の関係，各種オーバヘッドに関する知見，プログラム構造が AC 適用に及ぼす影響，デバイスの特性など，様々な視点から取り組みを行い，統合することが必要となる．

本稿では，図 1 に示すような，演算精度の動的変更を可能とする AC 技術を実現することを目指し，その第一歩として，データ型を変更・調整した際に演算性能が定量的にどの程度推移するのかを，いくつかの基本的なベンチマークを用いて定量的に評価する．

3. 演算精度と実行性能に関する予備評価

本章では，演算精度の変更がアプリケーションの実行性能に与える影響を評価し，その結果について述べる．本稿では，行列積および Rodinia ベンチマーク [12], [13] を用いた．

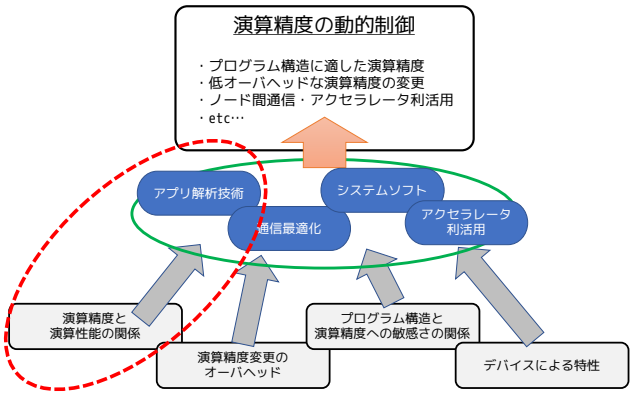

図 1 演算精度の動的制御の実現に向けた要素技術

3.1 評価環境

本稿で使用する評価環境は，筑波大学計算化学研究センター設置のスーパーコンピュータ “Cygnus” である．表 1 に，Cygnus のスペックのうち，本稿における評価と関連する項目を示す．本稿では，使用するデータ型と演算コストの関係を比較することを目的とするため，1 ノード上で評価を実施した．

表 1 性能評価環境（筑波大学計算科学研究センター “Cygnus”）

プロセッサ	Intel Xeon Gold 6126 2.6GHz 12 コア x 2 ソケット
メモリ	192GiB（255.9GB/s）
ストレージ（ローカル）	NVMe 3.2TB
ストレージ（共有）	Infiniband EDR 接続 NFS
OS	RHEL 7.9 （Linux Kernel 3.10.0）
コンパイラ	Intel Compiler 19.1.3

3.2 行列積による評価

ここでは，正方行列同士の積を求めるシンプルな行列積プログラムを用い，演算に用いるデータ型の違い，およびそれらを演算の際に変更した際の影響について評価を行った．本稿では，ユーザレベルでの演算精度変更による影響を調査するため，ソースコードレベルでデータ型を変更することにより評価を行った．ここでは C による実装を用い，特段の拡張を用いずに利用できる単精度（float）と倍精度（double）の間での性能の違いを観察する．コンパイルには Intel Compiler 19.1.3 を用い，最適化オプションは “-Ofast” とした．また，並列化等の影響を排除するため，ここでは，シングルノード，シングルスレッドで実行する．

図 2 に，行列サイズを 2,048 × 2,048 および 4,096 × 4,096 として実施した評価結果を示す．図 2 において，横軸は行列サイズおよび演算精度を表す．演算精度は，それぞれ以下のように定義される：

float 行列データを float で用意し，float のまま演算

double 行列データを double で用意し，double のまま演算

cast(f->d) 行列データを float で用意し，演算の際に double にキャストして用いる

cast(d->f) 行列データを double で用意し，演算の際に float にキャストして用いる

また，縦軸は，それぞれの行列サイズにおいて，float を基準とした相対性能を表す．

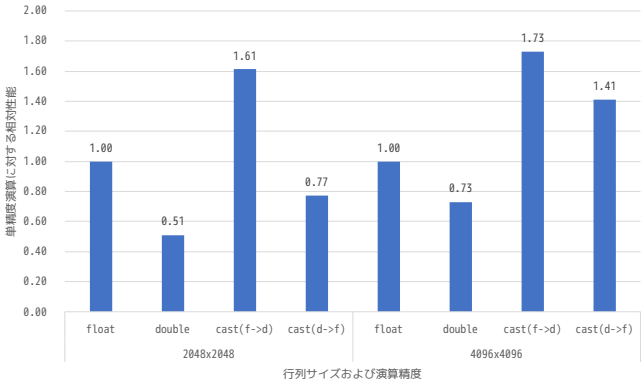

図 2 行列積による予備評価

図 2 において，float と double を比較すると，行列サイズ 2048x2048 の際におよそ 49 [%]，4,096 × 4,096 の際におよそ 27 [%] の性能低下がみられ，演算精度を高くすることにより，演算により多くの時間を要することがわかる．行列サイズが大きくなると，演算器だけではなく，メモリ階層の影響をより強く受けるため，単純に SIMD 幅等から想定されるような性能の変化とはならない．

一方，演算時にキャストしてデータ型を動的に変更する場合，float のまま，あるいは double のまま演算を行う場合よりも性能が向上する場合が見受けられる．プログラムの記述がコンパイラの最適化に与える影響など，様々な要因が考えられ，さらに詳細な解析が必要であるが，単純に演算精度による実行性能への影響に加え，プログラムの構造による影響も大きく，無視できないことが示唆された．

3.3 Rodinia ベンチマークによる評価

システムの性能評価を行うためのベンチマークアプリケーションは数多く考案されているが，本稿では，特に HPC システムへの AC の適用を目指していること，GPGPU や FPGA 等のアクセラレータデバイスを搭載したヘテロジニアスなシステムへの適用を考慮したいこと，様々な特性を持つアプリケーションへの AC の適用が今後必要となるであろうこと，を考慮し，Rodinia ベンチマーク [12], [13] を評価の対象として選択した．

演算精度の影響を評価するため，OpenMP 実装のうち，浮動小数点数による演算を含むもの（“backprop”，“cfd”，“hotspot”，“hotspot3”，“kmeans”，“LavaMD”，“leucoyte”，“lud”，“nn”，“particlefilter”，“srad_v1”，“srad_v2”）を対象とし，Rodinia ベンチマークに付属する実行スクリプトで指定されている入力データ・パラメータを用いて実行時間

の計測を実施した．この時，計測対象はそれぞれのプログラムの開始時から終了時までとし，シングルノード上で 24 スレッドによるマルチスレッド実行を行った．最適化オプションを “-Ofast” とした場合の評価結果を図 3 に，最適化オプションを “-O2” とした場合の評価結果を図 4 に，それぞれ示す．これらの図において，横軸はアプリケーションおよびデータ型を示しており，縦軸は単精度（float）を基準とした際の相対性能を示す．各アプリケーションにおいて，“original(f)” は，オリジナルソースでは float を用いた演算が主体となっており，“f -> d” はそれを double を用いるよう改変したものを示す．同様に，“original(d)” は，オリジナルソースでは double を用いた演算が主体となっており，“d -> f” はそれを float を用いるよう改変したものを示す．

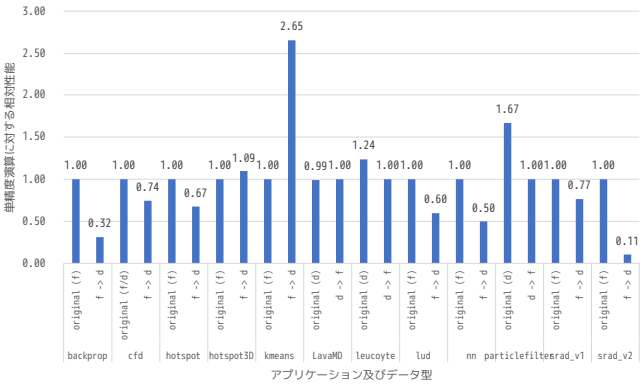

図 3 Intel Compiler “-Ofast” による評価

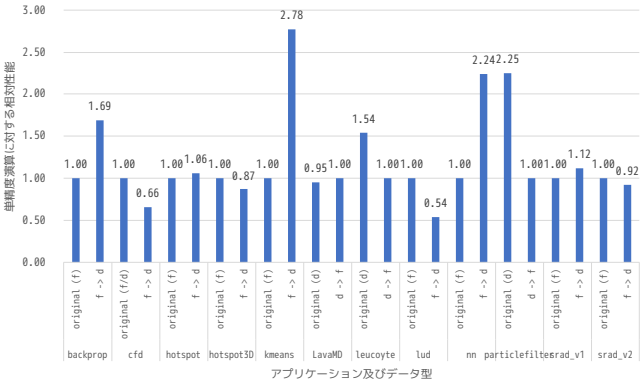

図 4 Intel Compiler “-O2” による評価

多くのアプリケーション（例えば，“cfd”，“hotspot3D”，“LavaMD”，“lud”，“srad_v1”）においては，float によるものと double によるものの差が少ない，あるいは double の方が若干性能が低くなっており，また，コンパイルオプションによる傾向の変化が少ない．一般的には，演算精度を上げることにより演算コストが増大し，実行性能が下がることが期待されるが，いくつかのアプリケーションでは，逆に性能が向上したり，あるいは想定される以上の性能低下が見られるものがある．

本稿では，このなかから，“backprop”，“kmeans” および “nn” を取り上げ，さらに評価を行う．その傾向を掴むことにより，より効率よく，演算精度と実行性能のトレードオフを最適化することが可能になるものと考えられる．

3.3.1 backprop

ここでは，Rodinia ベンチマークに含まれる “backprop” についてさらに評価を行った結果について示す．“backprop” は，多層ニューラルネットワークを用いた機械学習における，層間の重みを計算するベンチマークである．オリジナルソースでは float を用いて内部の演算を行っており，これを double に変更することで，実行性能がどのように変化するかを評価する．

まず，使用するデータサイズ，つまり入力層のサイズを変化させて評価を行った．その結果を図 5 に示す．図 3 および 4 では，プログラム全体を対象として実行時間を計測したが，ここでは，それに加え，ニューラルネットワークの構築や初期設定等を除いた学習処理のみを含む関数 “bpnn_train_kernel()” を対象とした計測も実施した．図 5 では，横軸が入力層のサイズ，縦軸が単精度（float）による演算を基準とした際の相対性能を示す．

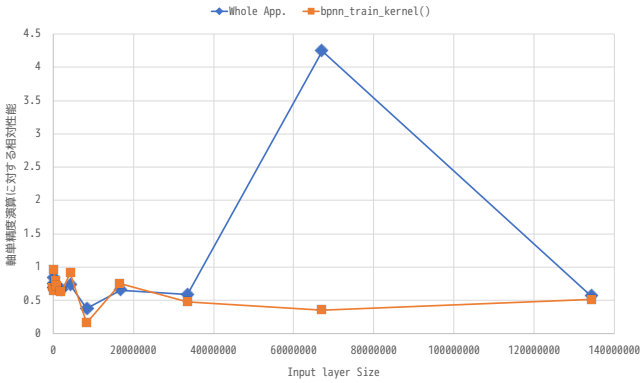

図 5 “backprop” プログラム全体・主要部分に対する評価

図 5 より，プログラム全体の比較的データサイズが小さい範囲，特に入力層のサイズが 20,000,000 より小さい範囲では，データ型の変化による性能の増減が大きいことがわかる．またよりデータサイズが大きくなると，float と double での相対性能がほぼ一定の値を取り始め，安定する．これは，データサイズが小さい場合には，キャッシュやコード最適化，SIMD 幅等の要素による影響がより強くなるのに対し，データサイズが一定以上大きくなると，メモリバンド幅等のスループットが支配的になるためであると考えられる．データサイズの増加に伴い，float から double に演算精度を増やすことによる性能向上がほぼ 0.5 倍で安定することから，比較的トレードオフのモデル化・定式化が行いやすいアプリケーションであると言える．

次に，同様の評価を，より小さい単位の関数ごとに

実施した．関数 “bpnn_train_kernel()” は演算の大部分を占めており，その内部では，関数 “bpnn_layerforward()” を 2 回，“bpnn_output_error()” と “bpnn_hidden_error()” をそれぞれ 1 回，さらに “bpnn_adjust_weights()” を 2 回呼び出している．そのうち，“bpnn_layerforward()” の 1 回目の呼び出しと，“bpnn_adjust_weights()” の 2 回目の呼び出しが “bpnn_train_kernel()” に要する時間のうちほとんどを占めている．そのため，図 6 には，これら 2 つの関数ごとにデータサイズを変更して評価した結果を示している．

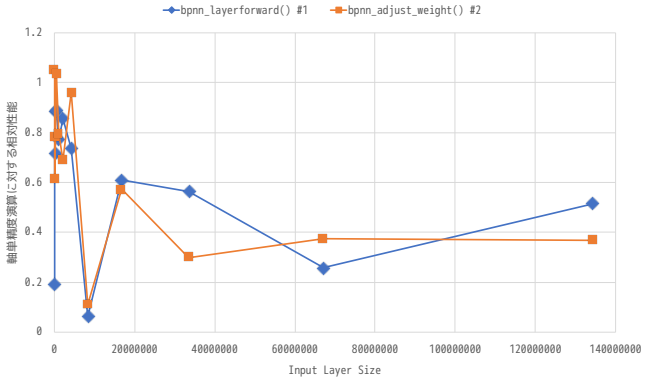

図 6 “backprop” 主要な関数ごとの評価

図 6 を見ると，図 5 同様，データサイズが小さい範囲ではデータ型の変更に対する性能の変化がばらつくのに対し，データサイズが大きくなるに従い，0.3 ~ 0.6 倍の範囲で推移するようになっている．ただし，関数ごとに性能の変化は少しずつ異なっていることから，より正確に演算精度と実行時間のトレードオフを最適化するためには，プログラムの特性・構造を考慮する必要があることが窺える．

3.3.2 nn

ここでは，Rodinia ベンチマークに含まれる “nn” についてさらに評価を行った結果について示す．“nn” は，k 近傍法による分類・クラスタリングを行うベンチマークである．オリジナルソースでは float を用いて内部の演算を行っており，これを double に変更することで，実行性能がどのように変化するかを評価する．使用するデータサイズズを変化させて評価を行った結果を図 7 に示す．図 7 では，プログラム全体に対する計測に加え，プログラムの主要部分を対象とした計測も実施した．ただし，このアプリケーションでは，データの読み込みと演算をメインループ内で繰り返す形をとっているため，プログラム全体に対する評価と，その主要演算部分に対する評価はほぼ一致する．また，図 7 では，横軸がデータサイズ，縦軸がオリジナルソースによる演算を基準とした際の相対性能を示す．

図 7 を見ると，“backprop” やその他のアプリケーションとは異なり，全体を double による演算に統一することで，全体の性能が向上していることがわかる．オリジナルソースにおける実装では，入力データは float として読み込むも

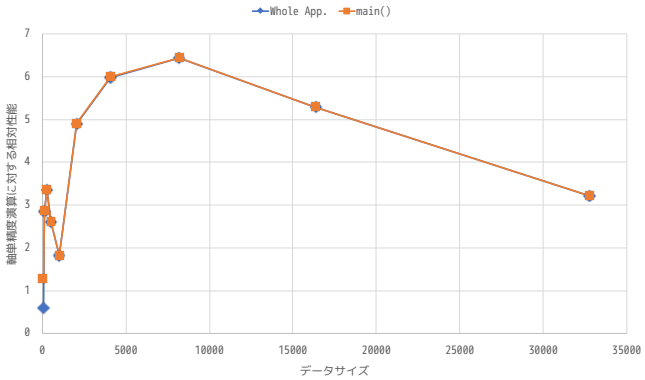

図 7 “nn” による評価

のの，メインループ内での演算結果は最終的に double 型のメンバを持つ構造体に格納されるなど，float と double が混在する形がとられている．そのため，全体を double に統一して演算を行った方が，コンパイラによる最適化等が適用されやすいということが要因の 1 つとして考えられる．また，データサイズが増えるに従い，一度 double による実装の方が性能が向上するが，その後さらにデータサイズを増加させると，性能差が小さくなっていくことがわかる．これは，データサイズが増えることで，データ量に対するメモリバンド幅の影響がより大きく現れるためと考えられる．

3.3.3 kmeans

ここでは，Rodinia ベンチマークに含まれる “kmeans” についてさらに評価を行った結果について示す．“kmeans” は，k 平均法による分類・クラスタリングを行うベンチマークである．オリジナルソースでは float を用いて内部の演算を行っており，これを double に変更することで，実行性能がどのように変化するかを評価する．使用するデータサイズを変化させて評価を行った結果を図 8 に示す．図 8 では，プログラム全体に対する計測に加え，プログラムの主要部分を対象とした計測も実施した．また，図 8 では，横軸がデータサイズ，縦軸がオリジナルソースによる演算を基準とした際の相対性能を示す．

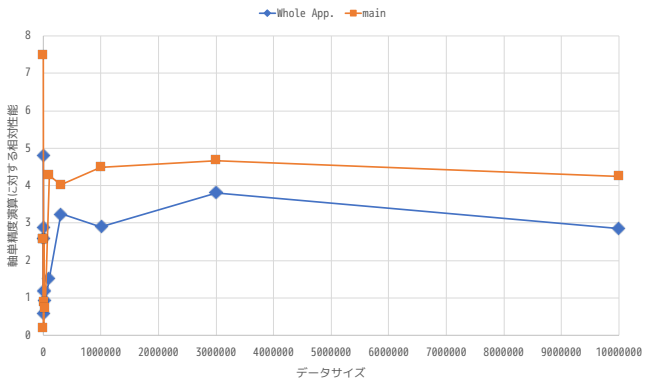

図 8 “kmeans” による評価

図 8 より，データサイズが小さい範囲では，データ型

の変更に対する性能の変動が大きくなっているのに対し，データサイズ 1,000,000 以上では，float による演算に対して double による演算の方が 3 ~ 4 倍程度高い性能を示すようになり，その増減が小さくなる．“nn” ではデータサイズの増加とともに性能差が小さくなっていたが，“kmeans” では，それが一定の差で安定していることがわかる．“nn” と比較して “kmeans” は比較的アクセスパターンがシンプルであるため，データサイズに対する性能の影響が少ないものと考えられる．

4. まとめ

本研究では，アプリケーション実行時に動的に演算精度を変更・調整することを想定し，これをアプリケーションのレベルで適用した際の実行性能と演算結果への影響・トレードオフを評価した．その結果，実アプリケーションは，単純に演算精度を低減させればその分実行性能が高まるという単純な関係にあるものではなく，場合によっては，演算精度を高めることによって，逆に性能が向上する場合があることが示唆された．

今後は，より多くの種類のアプリケーションに対して評価を行うとともに，プロファイラ等を用いたより詳細な解析を行い，演算精度変更に対する敏感さをアプリケーションやデータの構造・量に対してどのように関連づけることができるかを検討し，アプリケーションのレベルでより効率よく Approximate Computing 手法を適用する手法について検討を進める予定である．

謝辞 本研究成果の一部は筑波大学計算科学研究センターの学際共同利用プログラム（Cygnus）における 2021 年度課題「演算精度の動的変更によるノード間ロードバランシング」を利用して得られたものである．

参考文献

[1] Agrawal, A., Choi, J., Gopalakrishnan, K., Gupta, S., Nair, R., Oh, J., Prener, D. A., Shukla, S., Srinivasan, V. and Sura, Z.: Approximate Computing: Challenges and Opportunities, *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8.

[2] Mittal, S.: A Survey of Techniques for Approximate Computing, Vol. 48, No. 4, pp. 62:1–62:33 (online), available from ⟨https://doi.org/10.1145/2893356⟩.

[3] Karakoy, M., Kislal, O., Tang, X., Kandemir, M. T. and Arunachalam, M.: Architecture-Aware Approximate Computing, Vol. 3, No. 2, pp. 38:1–38:24 (online), available from ⟨https://doi.org/10.1145/3341617.3326153⟩.

[4] Shafique, M., Hafiz, R., Rehman, S., El-Harouni, W. and Henkel, J.: Invited - Cross-Layer Approximate Computing: From Logic to Architectures, *Proceedings of the 53rd Annual Design Automation Conference*, DAC ’16, Association for Computing Machinery, pp. 1–6 (online), available from ⟨https://doi.org/10.1145/2897937.2906199⟩.

[5] Chen, C.-Y., Choi, J., Gopalakrishnan, K., Srinivasan, V. and Venkataramani, S.: Exploiting approximate computing for deep learning acceleration, *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 821–826 (online), DOI: 10.23919/DATE.2018.8342119 (2018).

[6] Gustafson and Yonemoto: Beating Floating Point at Its Own Game: Posit Arithmetic, *Supercomput. Front. Innov.: Int. J.*, Vol. 4, No. 2, p. 71–86 (online), DOI: 10.14529/jsfi170206 (2017).

[7] Fujiki, D., Ishii, K., Fujiwara, I., Matsutani, H., Amano, H., Casanova, H. and Koibuchi, M.: High-Bandwidth Low-Latency Approximate Interconnection Networks, *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480.

[8] Osawa, H. and Hara-Azumi, Y.: Approximate Data Reuse-Based Accelerator Design for Embedded Processor, *ACM Trans. Des. Autom. Electron. Syst.*, Vol. 24, No. 5 (online), DOI: 10.1145/3342098 (2019).

[9] Sato, Y., Tsumura, T., Tsumura, T. and Nakashima, Y.: An Approximate Computing Stack Based on Computation Reuse, *2015 Third International Symposium on Computing and Networking (CANDAR)*, pp. 378–384 (online), DOI: 10.1109/CANDAR.2015.35 (2015).

[10] deepfloat: `https://github.com/facebookresearch/deepfloat`.

[11] Hara, T. and Hanawa, T.: Transprecision Calculation Platform Offloaded on FPGA, *The 5th cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming (xSIG 2021)* (2021).

[12] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing, *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54 (online), DOI: 10.1109/IISWC.2009.5306797 (2009).

[13] Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L. and Skadron, K.: A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads, *IEEE International Symposium on Workload Characterization (IISWC’10)*, pp. 1–11 (online), DOI: 10.1109/IISWC.2010.5650274 (2010).