CC-002

近似レベルを動的制御可能なアーキテクチャのための コンパイラフレームワークの検討

冨田 和孝 * 中村 朋生 * 小泉 透 * 入江 英嗣 * 坂井 修一 *
Tomida Kazutaka Nakamura Tomoki Koizumi Toru Irie Hidetsugu Sakai Shuichi

概要

計算速度や電力効率を改善するためのパラダイムの一つに、計算精度を引き換えに改善を図る Approximate Computing が研究されている。この手法では計算精度の低下によって生じる誤差をユーザーが許容できる範囲に収めることが必要である。本研究では、誤差を許容できる範囲が人の主観によって変化する場合があることに着目した。計算精度を動的制御可能なアーキテクチャと計算精度を段階的に変更可能な近似手法を用いるためのコンパイラフレームワークを提案する。アプリケーションに対して段階的に計算精度を変化させて実行命令数の変化を計測した。本提案により、同じプログラムにおける異なる精度と実行命令数での実行を実現し、近似精度の実行時切り替えを可能とした。

1 はじめに

計算速度や電力効率は、コンピュータシステムを考えるに際して非常に大きな要素である。その要素の改善のためのパラダイムの一つに Approximate Computing が研究されている。このパラダイムでは、計算誤差を許容することで、従来のコンピュータシステムに対して、非常に高い計算速度や電力効率を達成する技術である。また、誤差の大きさをユーザーが許容できる範囲内にすることで実用性が保たれる。[1] このパラダイムは、CNN などディープニューラルネットワークや、JPEG など画像を対象とした技術、ロボットの自己位置推定と地図作成を行う RGB-D SLAM などに広く適用できる。[2][3] ソースコードのどの部分に近似可能な場所が存在するかを静的に判定する手法 [4] や、フレームワーク [5] も存在し、アプリケーションレベルだけでなく、ハードウェアレベルでの近似手法 ([6][7][8] など) も研究されている。

Approximate Computing の手法によって生じる計算誤差は、ユーザーが許容可能な範囲であることが望ましい。しかし、その範囲はユーザーの主観によって動的に変化しうる。[9] 画像や動画、ゲームなどの画質が例として挙げられる。従来の静的な検査による近似レベル*1の推定 [10]、もしくはプログラマがソースコード内で近似レベルを与えるフレームワーク [11] では、想定したユーザーの許容範囲よりも広い許容範囲を持つユーザーに対してより良い効果を得ることが困難である。我々は、近似レベルを動的に制御することでこれらの課題を解決できると考えた。

本研究では、その手法の一つとして提案されているアーキテクチャ [12] に注目し、そのアーキテクチャが対象とする Loop Body Switching(LBS) と呼ばれるループへの近似手法に対するコンパイラフレームワークを提案する。対象とするアーキテクチャは近似レベルを ControrlStateRegister(CSR) と呼ばれる特殊レジスタに記憶し、その値を元に正確なループボディと近似されたループボディを切り替えて実行する。CSR の値を外部からの入力によって動的に変更できるようにすることで、実行時の動的な近似レベルの変更を可能にしている。本論文で提案するコンパイラフレームワークはソースコード中のユーザーが選んだ近似対象のループの直前に pragma を入れ、近似したループボディを所定の部分に書き込むことで、Loop Body Switching を行う為の命令と、ループ構造の変形を行う。第 2 章で研究背景として近似レベルを動的制御可能なアーキテクチャと LBS について述べる。第 3 章で本フレームワークについて述べ、第 4 章で評価と考察、第 5 章で関連研究について述べ、第 6 章で結論を述べる。

* 東京大学大学院情報理工学研究科 Graduate School of Information Science and Technology, The University of Tokyo

*1 ある部分の近似される度合いのこととする。近似レベルが高いとは強く近似されることを示す。

2 研究背景

本研究で対象としているアーキテクチャ [12]、および近似手法を以下に示す。

2.1 近似レベルを動的制御可能なアーキテクチャ

このアーキテクチャは近似レベルをハードウェアの内部ステートとして持つ。近似レベルはプログラム中の命令や、外部割り込み、ハードウェアによる検知で内部ステートが書き換えられることによって変更される。

2.2 Loop Body Switching

この手法は、Listing 1、Listing 2 に示すように近似ループボディを生成し、一部のイテレーションにおいて近似ループボディを元のループボディの代わりに実行するものである。ループボディをイテレーションごとに選択できるように、分岐命令の挿入、ループ構造の変更を行う。

Listing 1 近似対象のループの例

```
for(int i = 0; i < n; i++){
        a[i] = func(i,...);
}
```

Listing 2 変形されたループの例

```
for(int i = 0; i< n; i++){
        if(condition(approx_level)){
                //Exact Loop body
                a[i] = func(i,...);
        } else {
                //Approximate Loop Body
                a[i] = a[i-1];
        }
}
```

Listing 2 の変更後のループ構造は、正確なループボディと近似されたループボディ、ループ選択部分に分かれている。ループ選択部分では、変数 approx_level を用いてどちらのループボディを用いるか決定する。

Loop Body Switching のためのコードの書き換えなどでプログラマにかかる負担が大きい。

3 提案手法

Loop Body Switching のプログラマの負担を減らし近似レベルを動的制御するため、近似レベルを動的制御可能なアーキテクチャと協調したコンパイラを提案する。Loop Body Switching のループ選択部分の処理をハードウェアで実行する。ループボディの選択は、プログラム中の変数ではなく、近似レベルを動的制御可能なアーキテクチャの近似レベルを示す内部ステートによって決定される。内部ステートの値を用いてループボディを選択するための命令は approxbr として Listing 3 のように近似されたループボディのアドレスの差分を即値として持つものとして定義されている。

Listing 3 approxbr instruction

```
approxbr <offset to approximated loopbody>
```

提案するコンパイラは以下の特徴がある。

- プログラマにかかる負担の減少
- Loop Body Switching を行うループの決定
- ループ構造の変更

コンパイラがこれらの項目を自動で行えるようになれば、ループボディの近似方法を変えることで、様々なアプリケーションに適用可能になる。

3.1 Loop Body Switching を行うループの決定

近似したいループをプログラマが指定するための手法として、本研究では pragma を用いる。その理由は、プログラマができる限り少ないソースコードへの変更で目的を達成可能にするためである。Listing 4 に pragma の書き方の例を示す。

Listing 4 pragma の挿入の例

```
#pragma approx
for (int i = 0; i < n; i++){
        //loop_body
}
```

pragma を Clang のパーサーが処理した際に、近似したいループに Metadata を付与される。そして、Metadata の付与された LLVM IR が LLVM のミドルエンドに渡される。この Metadata を確認することで、ループ構造を変更するパスはプログラマが近似したいループを決定することができる。

3.2 命令の定義と変形後のループ構造

approxbr 命令を用いるためにループ構造に対して図 1 のような変形を行う。本研究ではこの変形を LLVM のミドルエンドにおける最適化パスを用いて行う。そのため、LLVM IR における approxbr の中間表現を定義する。LLVM IR においては、ある BasicBlock から遷移する BasicBlock への参照を命令が持っていなくては

ならない。そのため、Listing 5 のように第 1 引数に近似されたループボディへの参照、第 2 引数に正確なループボディへの参照を持つものとする。

Listing 5　approxbr instruction IR

```
approxbr <ref to approximated loop body> <
    ref to exact loop body>
```

今回のループボディの変形のアルゴリズムのために以下のようにいくつかの言葉を定義する。

- ループのそれぞれの BasicBlock の名前
  - Header：HeaderBB
  - Body：exBodyBB
  - Latch：LatchBB
  - Exit：ExitBB
- 新しく生成される BasicBlock の名前
  - approxbr 命令を含む BasicBlock：AxBrBB
  - ループボディを近似したもの：axBodyBB

3.3　ループ構造の変更

ループ構造の変更時には、approxbr 命令と、近似したループボディの生成、挿入が行われる。

正確なループボディと近似されたループボディへの分岐命令である approxbr 命令は、それらループボディの直前に挿入すればよい。そのようにすれば、近似対象のループを決定するだけで、自動的に approxbr 命令の挿入位置が決定する。

近似されたループボディは、プログラマが定義した関数や、ソースコードの別の箇所で書かれたボディのインライン展開を行うことで生成される。

以下にループ構造の変形のアルゴリズムを示す。

1. axBodyBB を生成
2. axBodyBB に axBodyBB から LatchBB への無条件分岐命令を追加
3. AxBrBB を生成
4. AxBrBB に exBodyBB と axBodyBB への分岐をもつ approxbr 命令を追加
5. HeaderBB の exBodyBB と ExitBB への条件分岐命令を削除
6. HeaderBB に AxBrBB と ExitBB への条件分岐命令を追加

これによって、図 1 の変形を行うことができる。exBodyBB は、HeaderBB の持つ分岐先の BasicBlock である。したがって、axBodyBB に制御を移すことで、ループボディ全体の書き換えが可能になる。

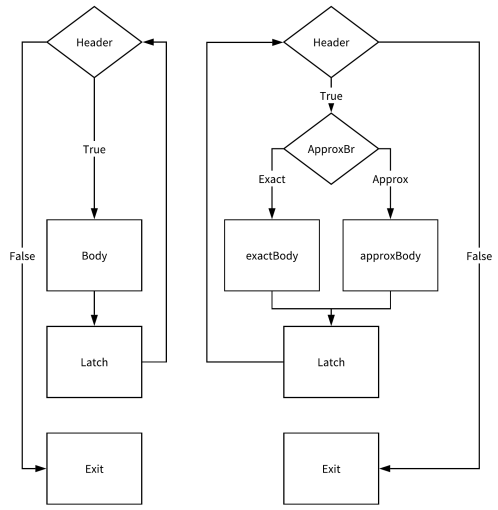


図 1　通常のループ構造 (左) と変形後のループ構造 (右)

4　評価・考察

提案手法を用いて実際のプログラムをコンパイルした場合、正しく近似の機会を得ることができているか、また実行命令数に変化があるかを評価する。命令セットシミュレータとして鬼斬弐 [13] を用いた。鬼斬弐には第 2 章で述べたアーキテクチャの一例が実装されている。鬼斬弐に実装されているものは、プログラム中の命令が CSR に近似レベルを設定する方法を用いている。CSR は 32bit のレジスタであり、2^{32} 段階の近似強度を持つことができる。CSR の Nbit までを近似強度に用いる場合、その近似強度を 0〜 $2^N - 1$ で表されるが、0 は全く近似を行わないものとしている。鬼斬弐に、実装されているものは N=5 である。ハードウェア内の擬似乱数生成器の値と CSR の値を用いて、ベルヌーイ分布から擬似的なサンプリングを行うことで正確なループボディと近似されたループボディを確率的に選択することで、approxbr 命令の処理を行なっている。

JPEG と kmeans の二つのプログラムを用いて実験を行った。JPEG は RGB 画像を JPEG に変換するもので、kmeans は RGB 画像中でクラスタリングを行うものである。今回の実験では、正しい動作の確認が目標であるため、近似されたループボディとして何も処理を行わないものを選択した。計測範囲は、近似手法を適用したループを含むループの入れ子構造の最も外側のループ

全体である。

JPEG は離散コサイン変換の内部のループに対し Listing 6 のように提案手法を適用した。結果を図 2 と図 3 に示す。図 2 から、近似レベルが増加するにしたがって、近似する機会の数を表す taken された approxbr 命令の数とその割合が増加していることがわかる。このことは、ループボディに対して適切な近似手法を用いることで、ループ全体の処理を近似レベルごとに異なる精度、速度、消費電力で行うことが可能になると考えられる。図 3 を見ると、実行命令数が減っていないことがわかる。これは、huffman 符号化が、近似したループボディが何も処理を行わないことにより画像の局所性を利用できなくなったためであると考えられる。そのため、計測区間内の huffman 符号化へ影響があることを考慮しその部分を無視した場合の実行命令数も計測した。huffman 符号化を除くと近似レベルに応じて実行命令数が減少していることがわかる。これは、huffman 符号化に影響を与えない近似ループボディを用いることができれば、全体として実行命令数が減少すると考えられる。このために、適切な近似手法を適用する方法には、提案手法において、近似されたループボディをユーザーが指定したり、コンパイラ側が自動で書き換えたりするための記法や、最適化手法を提案、実装することが考えられる。

この出力として得られた画像を図 4 に示す。ノイズが徐々に広がっていることがわかる。近似されたループボディの記述を工夫することで画像のノイズの影響を抑えることが可能になると考えられる。

kmens では、クラスタリングの距離計算のループに対して Listing 7 のように近似手法を適用した。結果は、図 5 と図 6 に示されている。この場合も同様に近似レベルに応じて、pproxbr 命令の数と割合が変化していること、実行命令数が変化していることが確認できた。kmeans の出力画像の例を 7 に示す。

Listing 6　dct 関数内への適用

```
#pragma approx //only add this pragma
for (i=8; i>0; i--)
{
        x8 = data [0] + data [56];
        x0 = data [0] - data [56];
        ......
        data [24] = (INT16) ((x0*c3 - x1*
            c7 - x2*c1 - x3*c5) >> s3);
        data [8] = (INT16) ((x0*c1 + x1*
            c3 + x2*c5 + x3*c7) >> s3);
        data++;
}
```

Listing 7　距離計算関数内への適用

```
#pragma approx //only add this pragma
for(i = 1; i < clusters->k; ++i) {
        d = euclideanDistance(p, &clusters
            ->centroids[i]);
        if (d < p->distance) {
                p->cluster = i;
                p->distance = d;
        }
}
```

5　関連手法

Approximate Comupting の近似手法は回路、アーキテクチャ、ソフトウェアといった幅広い領域で研究されてきた。[14][1] 本研究で提案するような近似手法を扱うためのフレームワークも研究されている。ACCEPT[10] は近似手法を適用するために独自の静的解析のルールを

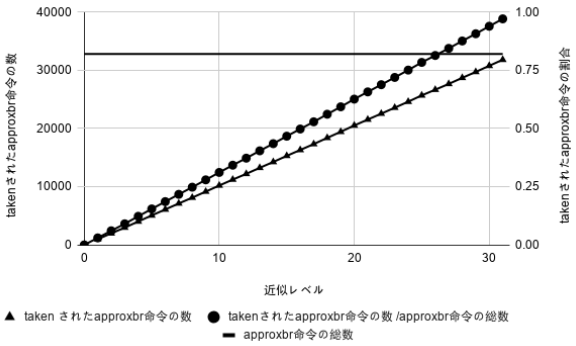


図 2　JPEG における taken された approxbr 命令の数と割合

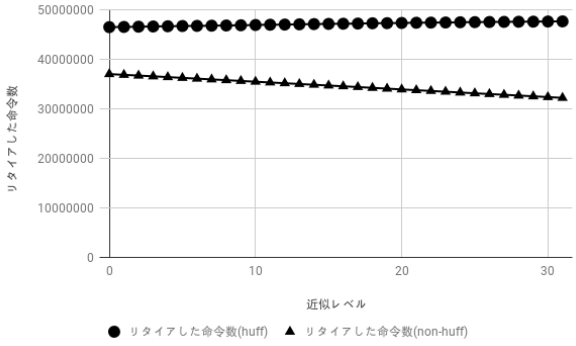


図 3　JPEG におけるリタイアした命令数

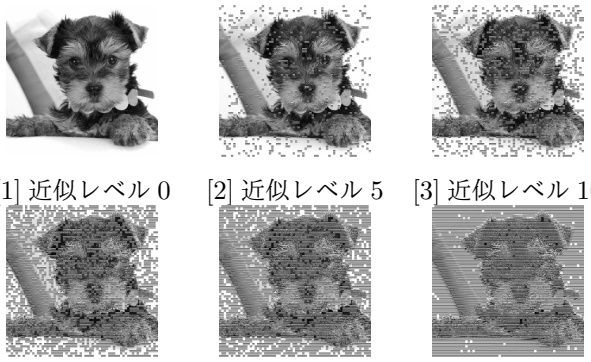
持っており、プログラマがコンパイラに近似しても良い変数と、正確に計算しなくてはいけない変数を独自の annotation で伝えることができる。Green[11] は、プログラマが出力の質 (Quality of Service:QoS) を定義し、ループや関数の変更方法を記述することで、コンパイラが設定された QoS を満たすようにバイナリを出力する。プログラマは QoS を計算する関数 QoS_Compute() と、その閾値として QoS_SLA を定義する。これらは実行時の計算精度の変更はできず、本研究と比べるとプログラマへの負担も大きい。

6 結論

本論文では、計算速度や、電力効率の改善のための一つのパラダイム Approximate Computing の手法を適用したプログラムが、ユーザーごとにより最適な近似強度で動作することを目的とした近似レベルを動的制御可能なアーキテクチャに注目した。
そのアーキテクチャが対象とする Loop Body Switching と呼ばれる近似手法に対して、必要な命令の生成と挿入位置の決定、ループ構造の最適化を行うフレームワークを提案した。命令の挿入位置は pragma によってプログラマからコンパイラに伝えられ、コンパイラミドルエンドで命令の生成、ループ構造の変更が行われる。これにより、近似レベルによって増減する近似の機会を利用することができるようになった。今後は、近似手法を適用するにあたって必要な記法や機能の提案、新しい近似手法の提案などが考えられる。

謝辞

本論文の研究は一部、民間共同研究 (株式会社富士通研究所)「Approximate Computing 自動適用技術の研究」によります



[1] 近似レベル 0　[2] 近似レベル 5　[3] 近似レベル 10
[4] 近似レベル 20　[5] 近似レベル 25　[6] 近似レベル 31

図 4　JPEG の出力画像の例

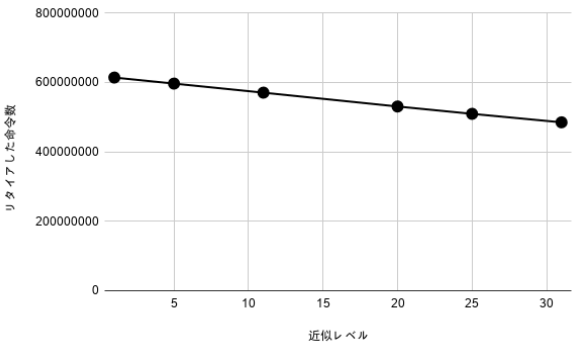


図 6　kmaens におけるリタイアした命令数

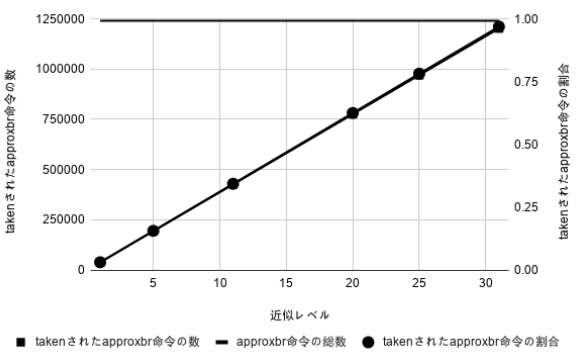
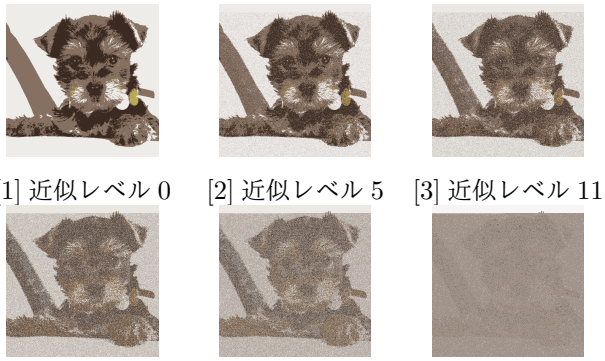


図 5　kmeans における taken された approxbr 命令の数と割合



[1] 近似レベル 0　[2] 近似レベル 5　[3] 近似レベル 11
[4] 近似レベル 20　[5] 近似レベル 25　[6] 近似レベル 31

図 7　kmeans の出力画像の例

参考文献

[1] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, Vol. 48, No. 4, March 2016.

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

[3] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, Vol. 34, No. 2, pp. 60–68, April 2017.

[4] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Asit K Mishra, Rajkishore Barik, and Somnath Paul. iact: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, p. 52, 2014.

[6] Matthias Jung, Deepak M. Mathew, Christian Weis, and Norbert Wehn. Invited - approximate computing with partially unreliable dynamic random access memory - approximate dram. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Trans. Comput. Syst.*, Vol. 32, No. 3, September 2014.

[8] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. Impact: imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pp. 409–414. IEEE, 2011.

[9] Joshua San Miguel and Natalie Enright Jerger. The anytime automaton. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 545–557. IEEE, 2016.

[10] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, Vol. 1, No. 2, 2015.

[11] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 198–209, 2010.

[12] 道上和馬, 中村朋生, 小泉透, 入江英嗣, 坂井修一. 近似レベルを動的制御可能なアーキテクチャの提案. *IPSJ SIG Technical Report*, 2020.

[13] 塩谷亮太, 五島正裕, 坂井修一. プロセッサ・シミュレータ「鬼斬弐」の設計と実装. 先進的計算基盤システムシンポジウム SACSIS2009, Vol. 2009, No. 4, pp. 120–121, 2009.

[14] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate computing: A survey. *IEEE Design Test*, Vol. 33, No. 1, pp. 8–22, Feb 2016.