A Novel Data Format for Approximate Arithmetic Computing

Mingze Gao, Qian Wang, Akshaya Nagendra, and Gang Qu

Electrical and Computer Engineering Department and Institute of Systems Research

University of Maryland, College Park

Introduction

- What is Approximate Computing (AC)?
  - Approximate (error) vs. accurate
- Why we need AC?
  - Power/energy efficiency
- Why AC works?
  - Many of the applications are error-tolerable, e.g. Machine Learning, Image/Signal Processing
  - Disable partial computation
- AC at different level
  - Arithmetic, Software, Compiler, Architecture, Memory, and Circuit

Approximate Arithmetic

- Observation:
  - Least Significant Bits (LSB) have much less contribution than Most Significant Bits (MSB) to the overall quality of the result.
- Approach:
  - Compute accurately on MSB
  - Apply approximation on LSB

Approximate Arithmetic

- Example: Compute S = A + B

$$A = 0011\ 1010\ 0001\ 1000_2$$
$$B = 0101\ 1011\ 1011\ 1000_2$$

- Build a 16-bit approximate adder that[1]:
  - 8-bit accurate adder for high 8 bits
  - 8 OR gates for low 8 bits

Error = 0.102% !

$$\begin{array}{ll} & 0111\ 1010\ 0111\ 1001_2 = 31353 \\ + & 0101\ 1011\ 1011\ 1000_2 = 23480 \\ \hline = & 1101\ 0101\ 1111\ 1001_2 = 54777 \end{array}$$

[1] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications," Circuits Syst. I Regul. Pap. IEEE Trans., vol. 57, no. 4, pp. 850–862, 2010.

Approximate Arithmetic

- However, what if the data is

$$A = 0000\ 0000\ 0101\ 1000_2$$
$$B = 0000\ 0000\ 1011\ 1000_2$$

$$
\begin{array}{rl}
 & 0000\ 0000\ 0111\ 1001_2 = 121 \\
+ & 0000\ 0000\ 1011\ 1000_2 = 184 \\
\hline
= & 0000\ 0000\ 1111\ 1001_2 = 249
\end{array}
$$

Error = 18.4% !

- We need a better approximate adder !

Approximate Arithmetic

- What we have learned:
  - "static" approximate adder vs. "dynamic" data
- Existing solutions:
  - Build additional discriminant circuit inside the approximate adder
- Drawbacks:
  - Fail to deliver significant power savings
  - Less accurate for multiplication

Approximate Integer Format

- Contribution: a novel Approximate Integer Format (AIF) and the corresponding computation mechanisms
- Desired properties of an ideal AIF
  - From "static" to "dynamic"
  - Cut-off the bitwidth of the operands
  - Suitable for all arithmetic operations
  - Provable error bound
  - Applicable to fixed point arithmetic

Preliminary -- Valid Block

- AIF is based on the segmentation of operands.
  - An n-bit positive integer N is segmented into [n/k] blocks with k bits per block.
  - Example: n = 16, k = 4, there are 4 blocks.

$A = 0000\ 1010\ 0001\ 1000_2$

- Definition 1: A valid block in a positive number is a block that has at least one '1' *before or inside* it.

$A = 0000\ 1010\ 0001\ 0000_2$

Preliminary -- Sentinel Bits

- Sentinel bits are used to truncate and round the less important bits to reduce bit-length of the operands
- Definition 2: The i_{th} sentinel bit *st[i]* of a number is defined as

$$st[i] = \begin{cases} 1, & block\ i\ is\ a\ valid\ block \\ 0, & block\ i\ is\ an\ invalid\ block \end{cases}$$

Preliminary --Precision Control

- Definition 3: The precision control 'pc' is the number of valid blocks in the number, from the leftmost one, that will be used in the computation.
- Example:

1500_{10} = 0000 0101 1101 1100_2
800_{10} = 0000 0011 0010 0000_2

  - Both have 3 valid block, st = 0111
  - If pc = 2, 2 blocks of each operand will be selected

1500_{10} = 0000 ***0101 1101*** 1100_2
800_{10} = 0000 ***0011 0010*** 0000_2

Preliminary --Rounding

- When we use sentinel bits to truncate the valid blocks, rounding is needed
- Two rounding techniques:
  - Classic rounding
    - For multiplication and division
  - Efficient rounding
    - For addition and subtraction

Classic Rounding

- Definition 4: The classic rounding of a number N at the i_{th} LSB means
  - adding the i_{th} bit to the $(i+1)_{th}$ bit
  - setting i_{th} bit and bits to its right to zero
- Example: $N = 263_{10} = 0000\ 0001\ 0000\ 0111_2$
  - From the 3rd least significant bit
    - $N = 0000\ 0001\ 0000\ 1000_2$
  - From the 4th bit
    - $N = 0000\ 0001\ 0000\ 0000_2$

Efficient Rounding

- Definition 5: The efficient rounding of A+B at the i_{th} bit is

$$A_{trunc} + B_{trunc} + Cin_{round}$$

  - A_{trunc} and B_{trunc} are obtained by truncating the i least significant bits from A and B
  - $Cin_{round} = (A_i \& B_i)$, AND i_{th} bits of A and B

Efficient Rounding -- Example

To compute $S = A + B$

$A = 0011\ 1010\ 1001\ 1000_2$

$B = 0000\ 1011\ 1011\ 1000_2$

Truncate A and B

$A_{trunc} = 0011\ 1010\ 0000\ 0000_2$

$B_{trunc} = 0000\ 1011\ 0000\ 0000_2$

Compute round-off carry in

$Cin_{round} = A7\ \&\ B7 = 1$

$S' = A_{trunc} + B_{trunc} + Cin_{round}$

$$\begin{array}{rr} & 0011\ 1010 \\ + & 1011 + 1 \\ \hline = & 0100\ 0110 \end{array}$$

S using efficient rounding:

$0100\ 0110\ 0000\ 0000_2 = 17920_{10}$

Accurate S:

$0100\ 0110\ 0101\ 0000_2 = 18000_{10}$

Approximate Integer Format

- Given a 4-block operand $A= b_3b_2b_1b_0$.
  - Only five possible values of A's sentinel bits st_a: 0000, 0001, 0011, 0111, 1111.
  - For the first four cases, the data A will be stored in following format:

st_a	b_2	b_1	b_0

  - For the last case of 1111, A will be stored as:

st_a	b_3	b_2	b_1

AIF Arithmetic --Addition

- Compute the sentinel bits of the result S: $st_s = st_A \mid st_B$
- Truncate i_{th} to $(i\text{-}pc+1)_{th}$ blocks of A and B to obtain A′ and B′, respectively
  - Suppose the leftmost '1' in st_s is in $st[i]$, and we plan to pick pc valid blocks
- Compute S′ = A′ + B′ and C_{out}
- Update st_s by $st_s[i+1] = C_{out}$
- Reformulate S in AIF using st_s and S′. Padding 0's if necessary

AIF Arithmetic --Addition Example

Original data

$A = 0011\ 1010\ 0001\ 1000_2$

$B = 0000\ 1011\ 1011\ 1000_2$

Data in AIF

$A' = 1111\ 0011\ 1010\ 0001_2$

$B' = 0111\ 1011\ 1011\ 1000_2$

Compute st_s:

$$\begin{array}{lr} & 1111 \\ \text{or} & 0111 \\ \hline = & 1111 \end{array}$$

Compute S'

$$\begin{array}{lr} & 0011\ 1010 + 0 \\ + & 1011 + 1 \\ \hline = & 0100\ 0110 \end{array}$$

Reformulate S in AIF:

$1111\ 0100\ 0110\ 0000_2 = 17920_{10}$

Accurate S:

$0100\ 0101\ 1101\ 0000_2 = 17872_{10}$

AIF Arithmetic --Multiplication

- Round the leftmost pc valid blocks of A and B into A' and B′
- Compute S′ = A' * B′
- Compute sentinel bits st_S using st_A and st_B and carry out
- Shift and reformulate S in AIF using S′ and st_S. Padding 0′s if necessary

Error analysis

- Let rounding error of A and B are er_A and er_B, respectively.
- Error of AIF based addition:
  - $Er_{add} = 2*\max(er_A, er_B)$
- Error of AIF based multiplication:
  - $er_A + er_B + er_A*er_B$
  - er_A<<1, er_B<<1, $Er_{mul} \approx er_A + er_B$

Negative AIF

- Deal with negative integer
  - Cannot use previous equation to compute st
- Solution:
  - Re-define the valid block
  - A valid block in a negative number is a block that has at least one '0' before or inside it.
  - Replace logic OR with logic AND when computing st
  - Arithmetic operations remains the same

High level programming language

- Introduce appropriate instructions and data type.
- Incorporate it in compiler and ISA
- Example: define apxint as the AIF data type

```
int main(){
        apxint a = 2341;
        apxint b = 546;
        apxint sum = a + b, prod = a * b;
        …
}
```

Compute in Caution

- Condition criterion, e.g. if, while condition
- Data value that the result is very sensitive to
- Functions that have periodical property, e.g. sin, cos, and modulo operation

Experiment Results: HW Cost

- Overhead Comparison of Arithmetic Units and The Sentinel Bits Computing Circuits

	cells	area	power(nW)
8-bit checker	10	23.93	61475.68
16-bit checker	17	39.89	132145.68
8-bit adder	91	212.12	752614.84
16-bit adder	230	322.33	2234652.24
32-bit adder	498	1116.93	4818821.94
8-bit multiplier	377	1037.62	2829745.1
16-bit multiplier	1406	4208.68	10815807.06
32-bit multiplier	4916	15126.01	34033690.3

Fibonacci Sequence: Accuracy

- First 40 elements in Fibonacci Sequence
  - A number is sum of its previous two: 1, 1, 2, 3, 5, 8, 13, 21, 34, …
  - Test the error propagation

#	pc=2	pc=3	pc=4	#	pc=2	pc=3	pc=4	#	pc=2	pc=3	pc=4
1~13	0	0	0	22	-0.02732	3.49E-05	0	31	-0.02291	-0.00035	2.30E-06
14	-0.00984	0	0	23	-0.02692	0	0	32	-0.02267	-0.00059	-8.51E-06
15	-0.01317	0	0	24	-0.02707	1.33E-05	0	33	-0.02276	-0.0005	-4.38E-06
16	-0.01691	0	0	25	-0.02912	0.000404	8.24E-06	34	-0.02273	-0.00053	-5.96E-06
17	-0.01858	0	0	26	-0.03225	0.000336	1.02E-05	35	-0.02274	-0.00052	-5.36E-06
18	-0.01985	0	0	27	-0.0375	0.000362	9.44E-06	36	-0.02274	-0.00052	-5.59E-06
19	-0.02173	-0.00044	0	28	-0.03947	0.000352	9.72E-06	37	-0.0328	-0.0001	1.05E-06
20	-0.02905	0.000183	0	29	-0.04118	0.000356	9.61E-06	38	-0.03621	-0.00046	-5.53E-06
21	-0.02625	-5.65E-05	0	30	-0.04205	0.000354	9.66E-06	39	-0.04003	-0.00064	-3.02E-06
								40	-0.04174	-0.00077	-3.98E-06

Real Life Application --IDCT

Which one is the original image?

PSNR = 41.76

(a)

(b)

PSNR = 89.64

PSNR =116.29

(c)

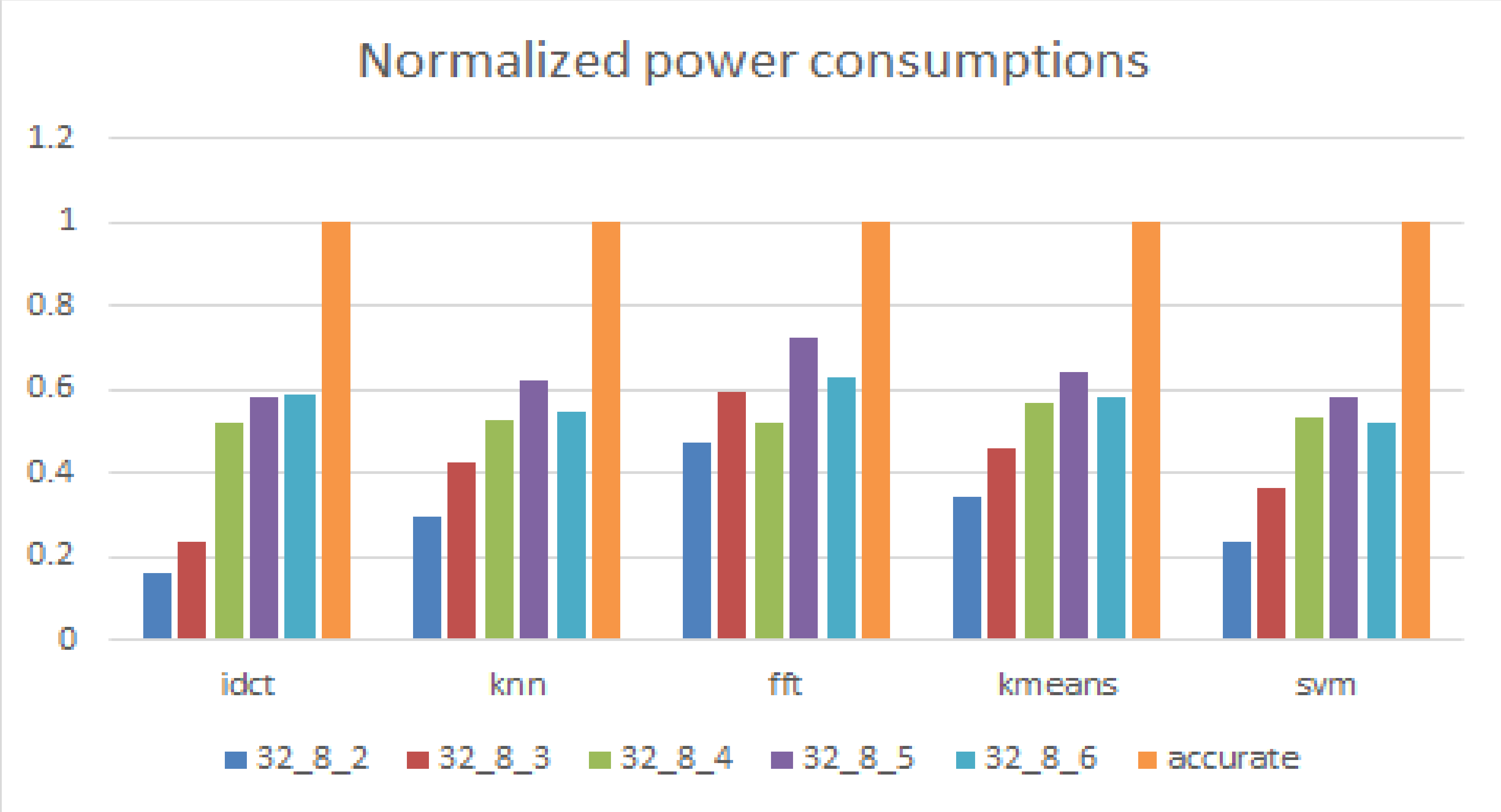
(d)

original image

Real Life Applications --Quality

AIF modules	IDCT	KNN	FFT	Kmeans	SVM
32_8_2	41.7579	92.9%	0.0081	1.125%	60.94%
32_8_3	64.6398	93.2%	2.79E-04	0.125%	85.11%
32_8_4	89.8324	93.1%	1.61E-05	0	85.98%
32_8_5	112.0872	93.2%	3.02E-06	0	85.59%
32_8_6	116.2944	93.2%	7.11E-08	0	86.42%
baseline	116.2949	93.2%	0	0	86.42%
Error Metric	PSNR	Classification accuracy	ARES	miss-clustered%	Classification accuracy

Final Result --Power Savings



Thank you!