Lookup Table Allocation for Approximate Computing with Memory under Quality Constraints

Ye Tian, Qian Zhang, Ting Wang and Qiang Xu
Department of Computer Science & Engineering, The Chinese University of Hong Kong
Shenzhen Research Institute, The Chinese University of Hong Kong
{tianye, qzhang, twang, qxu}@cse.cuhk.edu.hk

***Abstract*—Computation kernels in emerging recognition, mining, and synthesis (RMS) applications are inherently error-resilient, where approximate computing can be applied to improve their energy efficiency by trading off computational effort and output quality. One promising approximate computing technique is to perform approximate computing with memory, which stores a subset of function responses in a lookup table (LUT), and avoids redundant computation when encountering similar input patterns. Limited by the memory space, most existing solutions simply store values for those frequently-appeared input patterns, without considering output quality and/or intrinsic characteristic of the target kernel. In this paper, we propose a novel LUT allocation technique for approximate computing with memory, which is able to dramatically improve the hit rate of LUT and hence achieves significant energy savings under given quality constraints. We also present how to apply the proposed LUT allocation solution for multiple computation kernels. Experimental results show the efficacy of our proposed methodology.**

I. Introduction

Emerging recognition, mining, and synthesis (RMS) applications are usually error-resilient. They often process noisy and redundant datasets and have many "acceptable" outputs with limited user perceptual capability. As the system does not need to perform exact computation, one promising solution to achieve high energy efficiency is approximate computing, which trades off computational effort and output quality [1, 2].

Another effective technique for energy savings and performance improvements is computing with memory [3], which stores results of frequently-used functions in a lookup table (LUT) and reuses pre-stored outputs directly without conducting computation. For instance, the energy consumption of the $\sin x$ function involving hundreds of instructions is about 100nJ, while read operation of a 512kB SRAM at 32nm technology only consumes about 100pJ [4].

Many computation kernels are very likely to have similar outputs when encountering similar input patterns. Based on this observation of similarity, Rahimi et al. explores the combination of approximate computing and computing with memory [5, 6, 7, 8, 9]. Instead of searching for the exactly matching entry in the LUT, approximate computing with memory stores only a subset of function responses in associative memories and finds a similar entry to output an approximate result.

Due to the increasing energy consumption of search and read operation required by a larger lookup table [9], only a small portion of function responses can be pre-stored in the LUT for computation reuse. For patterns whose output quality cannot be guaranteed with this LUT, the processor needs to perform accurate computation. Consequently, which patterns to store decides the hit rate (i.e. reuse rate) and the energy saving. To achieve a higher hit rate, most existing solutions store output values for those frequently-appeared input patterns, without considering output quality and/or intrinsic characteristics of the target kernel. In addition, sharing the lookup table for multiple functions has not been explored in the literature yet.

Based on the above, in this paper, we propose a novel lookup table allocation technique for approximate computing with memory under quality constraints, which can be applied to various computation kernels. The main contributions of this paper are as follows:

- We present how to select proper data points to store in the lookup table for a given computation kernel, which dramatically improves the hit rate of LUT, thereby achieving significant energy savings under given quality constraints;
- We propose a novel lookup table allocation technique for multiple computation kernels, which is able to maximize the total energy saving under a given size of the lookup table.

The remainder of this paper is organized as follows. Section 2 presents related works on approximate computing with memory and the motivation for this paper. Section 3 details the proposed lookup table allocation technique. Experimental results comparing existing method and the proposed technique are then presented in Section 4. Finally, Section 5 concludes this paper.

II. Related Works and Motivation

Computing with memory achieves energy savings by storing results of frequently-used functions in a lookup table. The LUT can be stored in the memory of the processor and retrieved based on hashing function [10, 11]. Richardson caches the results of Ackerman's function and replaces subsequent function calls with table lookup, achieving real speedup as much as 1473 times [12]. Heydon et al. propose a technique for recording and propagating precise dependencies for efficiency when caching a function call [13]. Recently, a number of computing with memory techniques [14, 15] propose to use memristor-based ternary content-addressable memories (TCAMs) for low-power search and read operations of the LUT. In the above techniques, only exactly matched entries in the LUT can be reused without taking advantage of the error-resilience capability of RMS applications.

The energy consumption of a computation kernel with associative computing using LUT can be calculated as

$$rE_t + (1-r)(E_c + E_t) = E_c - (rE_c - E_t),$$

where r is the hit rate of the LUT, E_t represents the energy for search and read operation of the LUT, and E_c represents the energy consumption of the original computation. Therefore energy saving can be calculated as

$$E_s(\%) = \frac{rE_c - E_t}{E_c} = r - \frac{E_t}{E_c}.$$

As $E_t << E_c$ in most cases, the hit rate r determines energy saving of lookup table based computing.

To improve the hit rate, many previous works apply inexact match considering the similarity of input patterns in RMS applications. Rahimi et al. employ voltage overscaling on associative memristive memory modules and enables inexact match that allows the search pattern to be within a specified Hamming distance from the prestored patterns [9]. As the significance of different bits of the input operand could be quite different, Imani et al. propose a resistive configurable associative memory (bitline configurable and row configurable), which can perform approximate search on selected bit indices or selected patterns [6]. They also design a multi-stage content addressable memory in [16], which matches patterns stage by stage from the most significant bits.

To the best of our knowledge, existing approximate computing with memory techniques profile the occurrence frequency of the input patterns offline and store the function responses for those frequently-appeared input patterns. Such simple lookup table allocation strategy is not very effective. As shown in [17], when the search pattern is within 2-bit Hamming distance, the hit rate improvement is only about 10% than that of exact match. More importantly, the increase of energy saving is at the cost of uncertain output quality loss, and hence the scope is limited to certain applications with extremely high error-tolerance capability (e.g., image processing).

Motivated by the above, in this paper, instead of filling LUT according to the occurrence frequency of input patterns, we propose a novel lookup table allocation technique considering the intrinsic characteristics of the target computation kernel (e.g., input-output relationship and input-frequency relationship), which aims at optimizing LUT hit rate under quality constraints to maximize its energy saving.

III. Methodology

In this section, we firstly show how to select data points to store in the LUT for single function approximation given table size and output error bound. Then we present lookup table allocation method for multiple functions.

A. Points Selection for Single Function

1) Problem Formulation: We define a data point as (x, y), where x is the input and y is the output of the target function or computation kernel. Both x and y can be scalars or vectors of arbitrary dimensions. I represents the set of all possible data points.

Suppose the size of the lookup table is s, which means the table has s rows and can only store s points, denoted by $(cx_i, cy_i), i = 1, 2, \ldots, s$, where cy_i doesn't need to be the output for cx_i. We define input bounds (denoted by $d_i, i = 1, 2, \ldots, s$) for (cx_i, cy_i) as follows,

$$d_i = \max \text{dist}, s.t. |y - cy_i| \leq \text{bound}, \forall |x - cx_i| \leq \text{dist}.$$

So d_i is determined by (cx_i, cy_i) and the predefined error bound (denoted by bound). And all data points satisfying this input bound are defined as neighbors of the centroid.

For example, Fig. 1 shows some data points around the centroid (cx, cy) in the input space. For data point (x_{j1}, y_{j1}), as $|y_{j1} - cy| >$ bound, the input bound d for (cx, cy) must be smaller than $|x_{j1} - cx|$. Data points within the dashed circle are neighbors of (cx, cy). Therefore, for (x_{j2}, y_{j2}), though $|y_{j2} - cy| <$ bound, it is not a neighbor. Based on this definition of input bound, for any possible point (x_0, y_0), as long as the distance between x_0 and cx_i is not larger than d_i, the distance between y_0 and cy_i will satisfy the error bound and y_0 can be approximated with cy_i. And we say (x_0, y_0) is a neighbor of (cx_i, cy_i) and is covered by the LUT.

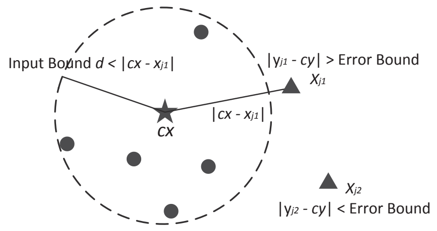
Input Bound $d < |cx - x_{j1}|$
$|y_{j1} - cy| >$ Error Bound
X_{j1}
cx
$|cx - x_{j1}|$
X_{j2}
$|y_{j2} - cy| <$ Error Bound

Fig. 1. Definition of Input Bound for the Centroid (cx, cy) Based on Error Bound

Given $(cx_i, cy_i), i = 1, 2, \ldots, s$ and error bound of output approximation, we can calculate input bounds $d_i, i = 1, 2, \ldots, s$, and then decide whether a certain data point in I is covered. We use Φ to represent the set of all points covered by all prestored points in LUT (i.e. $(cx_i, cy_i), i = 1, 2, \ldots, s$). For any point $(x_j, y_j) \in I$, p_j represents the probability that the data point appears during online computation. The hit rate of LUT is defined as the probability that the acceptable output can be found in the lookup table for arbitrary input and thus can be calculated by

$$\text{hit rate} = \sum_{(x_j, y_j) \in \Phi} p_j.$$

Our objective is to determine the s points to store and maximize the online hit rate, i.e.

$$\max_{(cx_1, cy_1), (cx_2, cy_2), \ldots, (cx_s, cy_s)} \sum_{(x_j, y_j) \in \Phi} p_j,$$

and therefore we can obtain maximum energy saving.

2) Flow of Data Selection for Single Function: To solve this problem, as shown in Fig. 2, our method contains the following three steps.

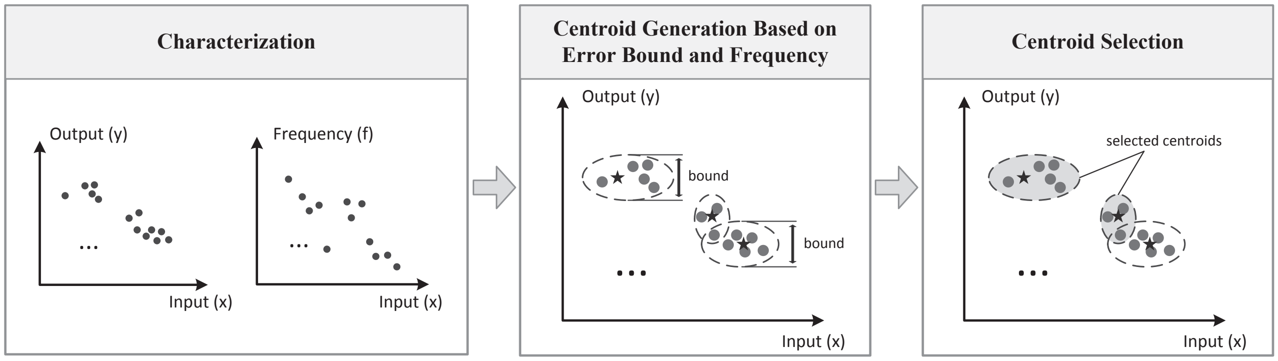

Fig. 2. Flow of Data Selection For Single Function

- **Characterization.** For the target computation kernel, find all possible data points $(x, y) \in I$ and frequency f_j that each data point (x_j, y_j) appears. Frequency is used to approximate the probability p_j that (x_j, y_j) appears.
- **Centroid Generation Based on Error Bound and Frequency.** Centroids are intialized and updated with the inputs, outputs and frequency of their neighbors. Neighbors of certain centroid are all points within the input bound from the centroid.
- **Centroid Selection.** Select s centroids among all centroids generated from step 2 to store in the LUT to maximize the hit rate.

Details of step 2 and step 3 will be explained in the following subsections.

```
Input :
        all data points $(x, y)$,
        frequency $f$ that each point appears,
        output error bound
Output:
        centroids $(cx_1, cy_1), (cx_2, cy_2), \cdots$
1 repeat
2     Centroid Initialization: find the data point
      with highest $f$ among uncovered points and
      set it as the new centroid $(cx^{(0)}, cy^{(0)})$;
3     repeat
4         Neighbors Update: update the neighbors
          and input bound of the centroid
          $(cx^{(i)}, cy^{(i)})$ based on output error bound;
5         Centroid Update: update the centroid to
          $(cx^{(i+1)}, cy^{(i+1)})$ based on $f$;
6     until Centroid doesn't change or No. of
      updates $> Th_1$;
7     Centroid Finalization: find the centroid
      among $(cx^{(0)}, cy^{(0)}), (cx^{(1)}, cy^{(1)}), \cdots$ with
      the highest sum of $f$, and output this centroid;
8     Update covered data points;
9 until sum of $f$ of all covered points $> Th_2$;
```
Algorithm 1: Process of Centroid Generation

3) Centroid Generation Based on Error Bound and Frequency: Process of centroid generation is shown in Algo. 1. To initialize a new centroid, the data point with highest frequency among uncovered data points is chosen to be the original centroid. The centroid is modified repeatedly until it doesn't change any more or the number of updates is larger than certain threshold. Then another new centroid is initialized and modified until the sum of frequency of all covered points is larger than certain threshold.

Neighbors Update Based on Error Bound. As mentioned above, input bound d of (cx, cy) is defined as

$$d = \max \text{dist}, s.t. |y - cy| \leq \text{bound}, \forall |x - cx| \leq \text{dist}, (x, y) \in I.$$

To calculate d, we check all data points having similar inputs with cx. Input distances between x and cx are calculated and sorted in ascending order. As shown in Fig. 3, $d_{j,1} < d_{j,2} < \cdots < d_{j,k} < d_{j,k+1} \cdots$, where input distance $d_{j,l} = |x_{j,l} - cx|, l = 1, 2, \cdots$. Then from the point with smallest input distance $(x_{j,1}, y_{j,1})$, output error $e_{j,l} = |y_{j,l} - cy|$ is calculated, and then compared with the error bound (denoted by e) until we find the first point $(x_{j,k+1}, y_{j,k+1})$ whose output error is larger than the error bound. Input bound is set as $d = d_{j,k}$, and data $(x_{j,1}, y_{j,1}), (x_{j,2}, y_{j,2}), \cdots, (x_{j,k}, y_{j,k})$ are neighbors of the centroid.

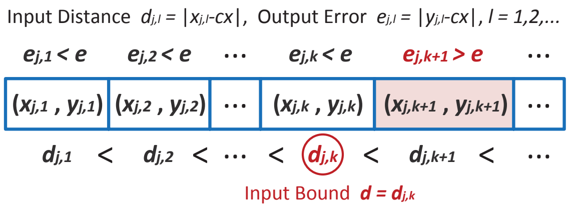
Input Distance $d_{j,l} = |x_{j,l}-cx|$, Output Error $e_{j,l} = |y_{j,l}-cx|$, $l = 1,2,\ldots$
$e_{j,1} < e$
$e_{j,2} < e$
…
$e_{j,k} < e$
$e_{j,k+1} > e$
…
$(x_{j,1}, y_{j,1})$
$(x_{j,2}, y_{j,2})$
…
$(x_{j,k}, y_{j,k})$
$(x_{j,k+1}, y_{j,k+1})$
…
$d_{j,1} < d_{j,2} < \cdots < d_{j,k} < d_{j,k+1} < \cdots$
Input Bound $d = d_{j,k}$

Fig. 3. Calculation of Input Bound

Centroid Update Based on f. We use frequency of data point as its weight when updating the centroid, i.e.

$$(cx^{(i+1)}, cy^{(i+1)}) = \frac{\sum\limits_{(x_j,y_j)\in \text{neighbors of } (cx^{(i)},cy^{(i)})} f_j \times (x_j, y_j)}{\sum\limits_{(x_j,y_j)\in \text{neighbors of } (cx^{(i)},cy^{(i)})} f_j}.$$

With the above formula, centroid $(cx^{(i+1)}, cy^{(i+1)})$ is closer to neighbors with higher frequency compared to the previous centroid $(cx^{(i)}, cy^{(i)})$.

Condition to End the Centroid Update. If the centroid does not change after one update, the computation has converged. We add another condition to terminate the update, i.e. number of updates is larger than certain threshold. During update process, the neighbors are decided according to distance of input and output and then centroid is calculated with frequency. The neighbors and the centroid may oscillate between certain states and the computation will never converge. Therefore the constraint on number of updates is necessary for convergency.

Centroid Finalization. The above update process calculates a sequence of centroids $(cx^{(0)}, cy^{(0)}), (cx^{(1)}, cy^{(1)}), \cdots$. We find the centroid with the maximum sum of f of its neighbors to output, and then update covered points to determine whether the generation process can end.

Condition to End the Centroid Generation. We can continue computing centroids until all data points are covered. However for many computation kernels, there can be millions of data points, among which many points have quite low probability to appear. Considering scalability of the algorithm, we can ignore those low-frequency points and end the generation process when the sum of f of covered data points is larger than certain threshold.

4) Centroid Selection: With centroids generated from step 2, we then choose s centroids and maximize the sum of f of covered data points. There are mainly three steps of centroid selection as follows.

- Calculate sum of f of neighbors of all unselected centroids. For the kth centroid,
$$A_k = \sum_{(x_j,y_j)\in \text{neighbors of } (cx_k, cy_k)} f_j.$$
- Select the centroid with the largest sum of f. The centroid (cx_i, cy_i) and its input bound d_i will be stored in the lookup table as one entry (or one row).
- Remove all neighbors of the new selected centroid from neighbors of remaining unselected centroid. As these data points have already been covered by the table, we should not take them into account for later selection.

We repeat these three steps s times and obtain s entries to store.

Using the proposed method, as shown in Figure 4, there could be a phenomenon that for data point (x, y), cx_1 is the closest to x among all centroids but (x, y) is covered by (cx_2, cy_2). Therefore, we need to check all covered points and calibrate the input bound of the centroid if needed. This phenomenon will not happen in most cases unless two centroids are very close to each other.

During online computation with table lookup, for coming input operand x, the processor reads the row whose cx has the nearest input distance from x. If the distance is within the input bound, the corresponding cy is used directly to approximate the exact output y.

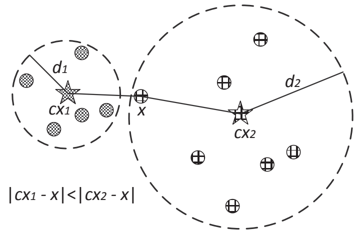

Fig. 4. Covered Data Point (x, y) Cannot be Approximated with the Nearest Centroid

B. Table Allocation for Multiple Functions

In practical situation, several applications or different computation kernels in one application will share the memory to store lookup tables. Therefore we need to allocate the space for LUTs for multiple functions. In this paper, we assume error bound of each kernel is given. To determine the error bound, the user should consider error accumulation if some kernel uses approximate output of another.

1) Problem Formulation: Suppose the total the memory space for LUTs is M. n functions or computation kernels $F_1, F_2, \cdots, F_n$ will share this memory and their memory spaces and corresponding hit rates of LUT are $m_1, m_2, \cdots, m_n$ and $r_1, r_2, \cdots, r_n$ respectively. The energy for search and read the table once is represented by E_t, and energy for computing function F_i once is represented by $E_{ci}, i = 1, 2, \cdots, n$. For the ith function, energy consumption of one computation using LUT can be calculated as
$$r_iE_t + (1 - r_i)(E_{ci} + E_t) = E_{ci} - (r_iE_{ci} - E_t).$$
Therefore energy saving of one computation, denoted by E_{si}, can be calculated as $E_{si} = r_iE_{ci} - E_t$. Suppose the ith function is executed for c_i times, then the energy saving of the whole lookup table for these n functions, denoted by E_s can be calculated as $E_s = \sum_{i=1}^{n} r_ic_iE_{ci} - \sum_{i=1}^{n} c_iE_t$, where c_iE_{ci} is actually the total energy consumption of the ith function, denoted by E_i. Our objective is to maximize energy saving, which is equivalent to the following problem:
$$\max_{m_i, i=1,2,\cdots,n} \sum_{i=1}^{n} E_ir_i,$$
where r_i is the hit rate of LUT with the memory space m_i for the ith function.

2) Constrained Optimization Problem: With the method to select data points for single function, for every function F_i, we can characterize a set of memory sizes and corresponding hit rates, i.e. $(m_{i,1}, r_{i,1}), (m_{i,2}, r_{i,2}), \cdots$. We define variable z_i to represent which choice we make for F_i, i.e., the memory size and hit rate of F_i is m_{i,z_i} and r_{i,z_i}. Then the problem can be modeled as a constrained optimization problem (COP).

There exists many kinds of solvers for COP. In this paper, we choose Gecode and the constraints on relationship between m_i and r_i can be implemented with element constraints. For example, the statement "element(home, c, x, y);" constrains the variable y to be the element of array c at index x. (The argument "home" represents an object that might store information which is useful for posting propagators and branchers and we don't need to be concerned with it when modeling with Gecode.) Therefore, for the ith function, we can use constraints:

$$\text{sizeArray}_i = [m_{i,1}, m_{i,2}, \cdots];$$
$$\text{rateArray}_i = [r_{i,1}, r_{i,2}, \cdots];$$
$$\text{element}(\text{home}, \text{sizeArray}_i, z_i, m_i);$$
$$\text{element}(\text{home}, \text{rateArray}_i, z_i, r_i);$$

where z_i is the index of elements we choose from these two arrays. The pseudocode in Gecode for the COP is shown in Algo. 2. After putting these constraints into Gecode solver, we can obtain memory sizes to allocate for all functions.

```
Input :
        $M$ (total memory space),
        $(m_{i,1}, r_{i,1}), (m_{i,2}, r_{i,2}), \cdots$ (relationship
    between memory size and hit rate of
    $F_i, i = 1, 2, \cdots, n$),
        $E_i$ (total energy consumption of
    $F_i, i = 1, 2, \cdots, n$)
Output:
        $\boldsymbol{z_i}$ (index of choice of memory size and
    hit rate for $F_i, i = 1, 2, \cdots, n$),
        $\boldsymbol{m_i}$ (memory size for $F_i, i = 1, 2, \cdots, n$),
        $\boldsymbol{r_i}$ (hit rate for $F_i, i = 1, 2, \cdots, n$)
1 Objective: $\max \sum_{i=1}^{n} E_i \boldsymbol{r_i}$
2 Constraint 1: $\sum_{i=1}^{n} \boldsymbol{m_i} \le M$
3 Constraint 2:
4 for $i$ from $1$ to $n$ do
5     $\text{sizeArray}_i = [m_{i,1}, m_{i,2}, \cdots]$;
6     $\text{rateArray}_i = [r_{i,1}, r_{i,2}, \cdots]$;
7     $\text{element}(\text{home}, \text{sizeArray}_i, \boldsymbol{z_i}, \boldsymbol{m_i})$;
8     $\text{element}(\text{home}, \text{rateArray}_i, \boldsymbol{z_i}, \boldsymbol{r_i})$;
9 end
```

Algorithm 2: Pseudocode in Gecode

IV. Experimental Results

A. Experimental Setup

As discussed above, energy saving of the lookup table can be calculated as

$$E_s(\%) = \frac{rE_c - E_t}{E_c} = r - \frac{E_t}{E_c},$$

where r is the hit rate of LUT (i.e. sum of frequency of covered data points), E_c is the energy consumption for calculating the kernel and E_t is the energy for table search and read operation. To estimate E_c, we simulate benchmark applications on x86_64 CPU with gem5 simulator, and then transform the simulated statistics (e.g., how many instructions executed) and micro-architecture configurations to the power simulator McPAT. As $E_t << E_c(\frac{E_t}{E_c} < 1\%)$ for these computation kernels, energy saving of LUT can be approximately represented by the hit rate r, therefore we only show the hit rate of LUT in the results.

In the experiment, we compare the hit rate of LUT between our table allocation method and existing method which simply chooses highest-frequency data points. We apply our method for single function to four commonly-used benchmarks of approximate computing as shown in Table I and use datasets from [18]. $Th1$ and $Th2$ in Algo. 1 are set as 8 and 90% respectively. Then we compare hit rate for inversek2j under different error bounds.

TABLE I
Benchmarks for Lookup Table Allocation for Approximate Computing with Memory

Computation Kernel	Inputs	Outputs
Savity-Golay Filter	5 Inputs	1 Output
Black Sholes	5 Inputs	2 Outputs
inversek2j	2 Inputs	2 Outputs
Fast Fourier Transform (fft)	20 Inputs	20 Outputs

B. Comparison of LUT Hit Rate for Various Kernels

Given table size and error bound of output, the proposed method selects proper data points to store and obtains higher hit rate of LUT. As shown in Fig. 5, for all four computation kernels, we compare hit rate of proposed method (denoted by "Our Method") and the existing method (denoted by "Highest Frequency") of four table sizes (i.e. 64, 32, 16 and 8). Error bound is set according to relative error of the kernel's output. Table with larger size has a higher hit rate as more information is prestored. Hit rate of our method is much larger (about 2 to 3 times) than that of existing method in all cases.

C. Comparison of LUT Hit Rate under Different Error Bounds

To prove the efficacy of our method, we change the error bound of output for inversek2j and compare hit rate under different error bounds. The error bound is set as the relative error and we apply our method when errors are $1\%, 5\%, 10\%$ and 20% respectively.

As shown in Fig. 6, when the error bound is 1%, the hit rates of our method and highest frequency method are the same if the table size is 32 or 64. As the bound is too strict and most centroids generated don't cover other points, selecting the centroid is equivalent with selecting the point with highest frequency. In other cases for any error bound or any table size, our method for table allocation achieves a much higher hit rate than existing method.

When the error bound is 20%, 30 centroids are generated and all data points have been covered, which means a LUT with 30 points stored can cover all data points and the hit rate is 1. However, using highest frequency method, the hit rate of a LUT with 64 points is still smaller than 1. This is because the error bound is very relaxed, some high-frequency data points can be approximated with only one point. The results indicate that the proposed method can construct a lookup table based on table size and error bound, which fully utilizes intrinsic characteristic of the target kernel.

V. Conclusion

LUT based approximate computing is one of the most promising energy-efficient computing techniques for RMS applications. Existing solutions simply stores high-frequency patterns without considering the output quality and the intrinsic characteristic of the target kernel. In this paper, we propose

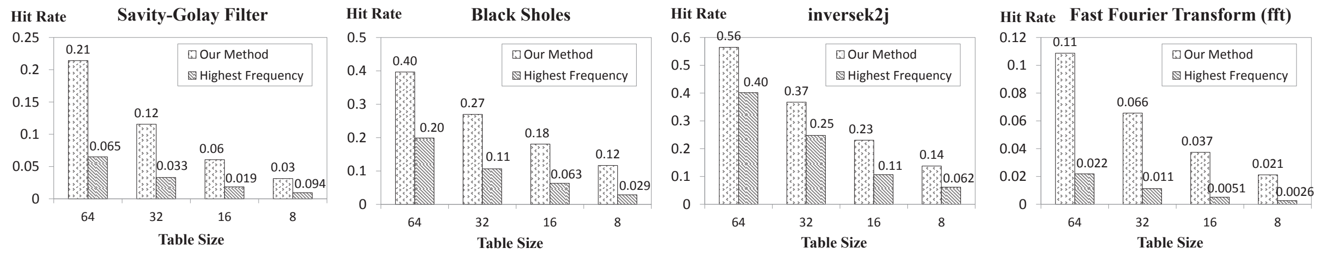

Fig. 5. Comparison of LUT Hit Rate between Our Method and Highest Frequency Method for Various Kernels (Error Bound: 5%)

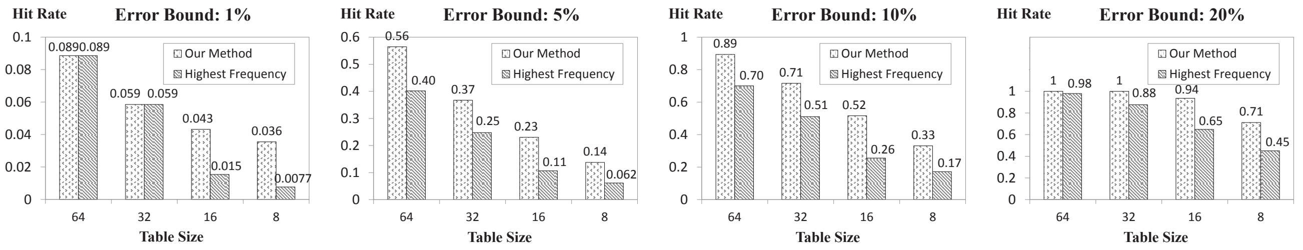

Fig. 6. LUT Hit Rate for inversek2j under Different Error Bounds

a novel lookup table allocation technique for approximate computing with memory based on the input-output relationship of the kernel and frequency distribution of different patterns, which guarantees output quality and achieves much higher LUT hit rate when compared to existing techniques, thereby achieving much better energy savings. We have also shown how to conduct effective lookup table allocation for multiple computing kernels.

VI. Acknowledgement

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61432017 and 61532017.

References

[1] S. Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):62, 2016.

[2] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel. Invited: Cross-layer approximate computing: From logic to architectures. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[3] P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In *IEEE Symposium on Computer Arithmetic*, pages 232–236, 1991.

[4] J. Cong, M. Ercegovac, M. Huang, S. Li, and B. Xiao. Energy-efficient computing using adaptive table lookup based on non-volatile memories. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 280–285. IEEE, 2013.

[5] J. Arnau, J. Parcerisa, and P. Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 529–540. IEEE, 2014.

[6] M. Imani, A. Rahimi, and T. Rosing. Resistive configurable associative memory for approximate computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1327–1332. IEEE, 2016.

[7] A. Rahimi, A. Ghofrani, M. Lastras-Montano, K. Cheng, L. Benini, and R. Gupta. Energy-efficient gpgpu architectures via collaborative compilation and memristive memory-based computing. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[8] A. Goel and P. Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):143–154, 2010.

[9] A. Ghofrani, A. Rahimi, M. Lastras-Montaño, L. Benini, R. Gupta, and K. Cheng. Associative memristive memory for approximate computing in gpus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 6(2):222–234, 2016.

[10] T. Kohonen. *Associative memory: A system-theoretical approach*, volume 17. Springer Science & Business Media, 2012.

[11] X. Liu, C. Deng, B. Lang, D. Tao, and X. Li. Query-adaptive reciprocal hash tables for nearest neighbor search. *IEEE Transactions on Image Processing*, 25(2):907–919, 2016.

[12] S. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. 1992.

[13] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN Notices*, volume 35, pages 311–320. ACM, 2000.

[14] J. Li, R. Montoye, M. Ishii, and L. Chang. 1 mb 0.41 μm^2 2t-2r cell nonvolatile tcam with two-bit encoding and clocked self-referenced sensing. *IEEE Journal of Solid-State Circuits*, 49(4):896–907, 2014.

[15] F. Alibart, T. Sherwood, and D. Strukov. Hybrid c-mos/nanodevice circuits for high throughput pattern matching applications. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 279–286. IEEE, 2011.

[16] M. Imani, Y. Cheng, and T. Rosing. Processing acceleration with resistive memory-based computation. In *Proceedings of the Second International Symposium on Memory Systems*, pages 208–210. ACM, 2016.

[17] M. Imani, S. Patil, and T. Rosing. Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 373–378. EDA Consortium, 2016.

[18] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34(2):60–68, 2017.