A Case for Table-Based Approximate Computing

Behrooz Parhami
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
parhami@ece.ucsb.edu

***Abstract*—The plummeting cost and rising density of memory units is leading to a resurgence of interest in tabular computing, already popular in the context of FPGA circuits. Use of tables offers a natural framework for bimodal or multimodal schemes that combine a quick, direct readout of approximate values with a refinement mechanism for improved precision when needed. In this paper, after reviewing ideas and methods relevant to the two domains of table-based and approximate computing, we discuss key benefits of using tables to realize the latter. We also point out how the covered methods are connected with related notions of adaptive-precision arithmetic and lazy computation.**

Keywords*— *Approximation; Arithmetic; Interpolating memory; Memory technology; Multi-level table; Multipartite table

I. INTRODUCTION

As an undergraduate engineering student, I remember using precomputed tables of useful functions, published in mathematical handbooks [1]. Compact, affordable digital calculators didn't become available until my graduate-student days in the early 1970s. We are now used to computing any value on-the-fly as needed, given the availability of inexpensive and lightning-fast hardware. However, computing complicated functions can be time- and energy-intensive, resources that are at a premium in many application domains. Time and energy waste become even more daunting when the same function values tend to be recomputed over and over again.

Having been dismissed, table-based computing returned in two main forms in the 1990s: to speed up normal, circuit-based computations by providing an initial estimate that would then be refined, and to reduce complexity and power consumption [2]. In an encyclopedia article [3] and a companion paper [4], I have argued that the age of big data, combined with advances in memory technology that have led to significant reduction in cost and incorporation of desirable features such as non-volatility, has brought table-based computing back into the spotlight.

As for memory cost reduction [5], the six decades since the 1950s have seen the price of memory decrease from many dollars to a few micro-cents per byte, a scaling of 10^9. Regarding density [6], memory chips that used to contain a few kilobytes now offer terabytes, another scaling of 10^9. Here, I argue that approximate computing [7], brought about by the need for fast, energy-efficient computation in application domains with no need for high precision, provides an additional motivating factor for using tables.

II. TABULAR COMPUTING: A CAPSULE REVIEW

Consider computing the function $y = f(\mathbf{x})$, where $\mathbf{x}$ is a vector representing the arguments (total of u bits) and the v-bit result y is the function value. A naïve lookup table for evaluating this function will be of size $2^u \times v$ bits, accessed by a u-bit address and delivering a v-bit word as output. This is quite practical when there are a fairly small number of low-precision arguments to deal with (u is small), thus the method's affinity with approximate computing. This naïve approach has limited applicability.

Reducing the required table size is accomplished by subjecting the u input bits to some pre-processing steps or allowing the v-bit output to be post-processed. A well-known example is that of multiplication of k-bit numbers, that is, computing $f(a, b) = ab$, with $u = v = 2k$, leading to the table size $2^{2k+1}k$. From the identity $ab = [(a + b)^2 - (a - b)^2]/4$, we see that by computing the two $(k + 1)$-bit values $a + b$ and $a - b$ in the pre-processing stage, accessing a squaring table of size $2^{k+2}(k + 1)$ twice, or two tables in parallel, and finally doing an addition and a 2-bit right-shift in the post-processing stage, extensive table-size reduction is possible. For example, the table size is reduced by a factor of 1890 (945, if two parallel tables are used for higher speed) in 12-bit multiplication. Additional optimizations can be applied to this method [8]. Such trade-offs between the circuit- and time-complexity of pre- and post-processing steps and the table size are commonly encountered.

In the rest of this section, I present brief discussions of some of the main ideas in the field of tabular computation. For more details, the reader is referred to the author's textbook on computer arithmetic and encyclopedia article on tabular computation [2] [3].

Interpolating memory: To compute $f(a)$, where a falls between two consecutive table entries ($x_i < a < x_{i+1}$), we can use $f(a) = f(x_i) + (a - x_i) f'(x_i)$, in which $f'(x_i)$ is the derivative or slope of the function at x_i. Taking x_i to be several most-significant bits of a and $a - x_i$ to be the remaining bits, and reading out $f(x_i)$ and $f'(x_i)$ from tables, the interpolating memory scheme [9] needs two post-processing operations: one multiplication and one addition.

Nonuniform segmentation: The interpolating memory scheme of the preceding paragraph divides the function-evaluation domain into equal-width segments. By using non-uniform segments, the table size can be reduced at the expense of more complex pre-processing [10].

Distributed arithmetic: Evaluation of linear functions of the form $a_1x_1 + a_2x_2 + \ldots + a_nx_n$, where the a_is are known constants, can be performed via bit-serial table-lookup [11]. As the ith bits of the inputs $x_1, x_2, \ldots, x_n$ arrive in turn, beginning with the least-significant bit, the n bits are used as index into a 2^n-entry table that stores the sum $x_1(i)a_1 + x_2(i)a_2 + \ldots + x_n(i)a_n$. The value read out is combined with a running total, after shifting the latter to the right by 1 bit. Thus, with k-bit input operands, distributed arithmetic entails k computation cycles, each consisting of a table access followed by a shift-add operation.

Bit-level optimization: Rather than viewing each operand in the argument vector $\mathbf{x}$ of $f(\mathbf{x})$ as an indivisible entity, we can decompose the arguments into their constituent bits and come up with an evaluation method that is optimized at the bit level. Distributed arithmetic, just discussed, is one instance of this approach. Other examples exist [12].

Iterative refinement: Most functions of interest can be computed iteratively, beginning with a reasonably close estimate, which may be read out from a table. For example, given an estimate for $\sqrt{z}$ with max error no greater than 2^{-a}, a new estimate with the much smaller error bound of 2^{-2a} can be obtained from the recurrence: $x^{(j+1)} = (x^{(j)} + z/x^{(j)})/2$. So, beginning with an estimate having an error no greater than $2^{-1} = 1/2$, the error can be reduced quadratically to 2^{-2}, 2^{-4}, 2^{-8}, and so on, reaching k bits of precision in $\log_2 k$ iterations. Because division is a more complex operation than multiplication, division-free versions of the refinement recurrence have been proposed (e.g., [13]).

Multi-level tables: For certain functions, it is possible to devise a multi-table scheme, in which each table receives chunks of the input as well as bits from other tables, with the final table in the chain, or some specially designed post-processing circuit, producing the output [14]. The design and optimization schemes are application-dependent.

Bipartite tables: To avoid the latency, cost, and energy requirements of multiplication implied by interpolating memory, multiplier-less methods [15] have been proposed. Bipartite tables [16] offer one such scheme. Consider a k-bit operand divided into a few of its most-significant bits, x_u, a few middle bits, x_m, and the rest of the bits x_r. We can approximate $f(x)$ as the sum $g(x_u, x_m) + h(x_u, x_r)$. Essentially, the decomposition of x creates intervals corresponding to x_u values and subintervals corresponding to x_m values. Function values are stored for each subinterval in the table $g(x_u, x_m)$. The ingenuity of the method is that instead of storing slopes for the various subintervals, as in interpolating memory, a common slope for each interval is stored, allowing the multiplication of the slope $s(x_u)$ and the displacement x_r to be performed by the second lookup table which yields the value $h(x_u, x_r)$. Selection of the number of bits in x_u and x_m offers tradeoffs in precision and table size. Various optimizations and improvements to the basic scheme above have been offered (e.g., [17], [18]).

Multipartite tables: Extension of the notion of bipartite tables to tripartite [19] and multipartite tables [20], [21] are quite natural to contemplate.

III. KEY CHALLENGES OF APPROXIMATE COMPUTING

One of the requirements of arithmetic operations, whether approximate or with regular precision, is reducing errors to the bare minimum and providing estimates for average-case and worst-case errors [22]. Standard floating-point arithmetic requires that the error in each arithmetic operation be limited to a single rounding error, that is, 1/2 *ulp* (half unit in least-significant position) [23]. Such results are referred to as correctly or properly rounded [24]. Even so, the direction of the error is often unknown. This property is rather expensive to ascertain, particularly in computing functions such as sine and log, so we often settle for an error of no more than 1 *ulp* (faithfully-rounded results), with corresponding time, cost, and energy savings. Approximate computing takes this notion further by tolerating multi-*ulp* error bounds for additional savings.

A wide array of specific designs and design frameworks have been proposed for arithmetic circuits that trade off precision for speed and energy economy, beginning with the two fundamental arithmetic building blocks: Adders (e.g., [25]) and multipliers (e.g., [26]). Evaluation of an approximate computing scheme is often done in the context of particular applications, based on the quality of outcomes compared with those based on regular computations.

So as to provide the readers with a complete example of how the trade-offs implied by approximate computing are provided, we describe the adder design cited above [25]. Each of the four sub-adders in Fig. 1 computes the sum of the corresponding operand segments, either with a predicted/estimated carry-in or with the actual carry-in from the preceding segment, the former resulting in faster operation but with results that are potentially imprecise. In addition to the trade-off just noted, we also have trade-offs in designing the carry predictors, with higher accuracy requiring more complex circuitry and, thus, greater VLSI area requirement and power consumption.

Maximum benefits of approximate computing accrue when an adaptive scheme is used to adjust the precision to application requirements, thus keeping the precision and the attendant hardware resources used to a bare minimum. Under such an adaptive scheme, a light-weight, quality-monitoring process is required, (which may be invoked sparingly; say, once per m invocations of the approximation scheme) to keep a tab on errors and to adjust the available parameters if the error becomes unacceptable. Design of such monitoring schemes is a great challenge. Furthermore, any error-bound estimates or guarantees will suffer as a result of periodic, rather than continuous, quality checks.

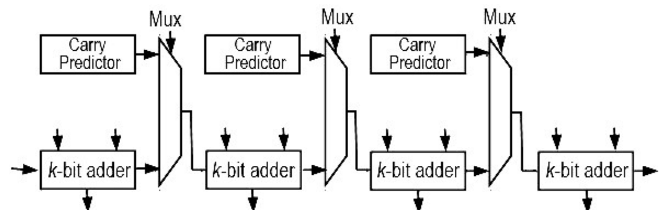

Fig. 1. A configurable approximate-adder design [25].

IV. Basic Benefits of Using Tables

We can rigorously analyze the error implied by a given approximate-computation scheme, such as the adder depicted in Fig. 1. Usually, however, the best we can do is to provide error bounds, perhaps with an accompanying probability distribution. For any particular set of input parameters, the actual error may be significantly less than what the analysis suggests. This can lead to error over-estimation and an associated over-design to taper the effect of such worst-case errors.

When we obtain approximate values from a table, the actual error in each table entry is known. So, it is quite feasible to store in the table the direction of the error (1 bit) and a tight, low-precision upper bound for it (several bits, depending on need), as in Fig. 2. The error-direction field, which tells us whether the table entry has been rounded up or down, would allow us to easily obtain directionally rounded values [23] if needed. The tight error upper-bound would enable the derivation of a more precise result via an adjustment process, if needed.

The "Adjustment" block may be activated to produce a higher-quality result according to the computational needs. Once we have a mechanism to produce higher-precision results, we can also use it to check the quality of the approximate result on a periodic basis. As shown in Fig. 2, the adjusted or higher-precision result may be sent to a quality-monitoring unit which renders judgment on the result quality, without increasing the critical-path delay.

The scheme depicted in Fig. 2 is a simple form of adaptive-precision arithmetic [27], which has been found desirable in computational geometry to keep floating-point errors in check. It is also related to the notion of lazy arithmetic [28], a scheme that entails expending as little effort on computational steps as possible, correcting any problems from low-quality results by backtracking and/or re-doing the affected computations.

Certain exotic number representation schemes already use tabular computation extensively, with excellent results. Residue number system (RNS) arithmetic [29], e.g., benefits from tables for both arithmetic operations as well as for I/O conversion and reconversion processes. Another example is logarithmic number system (LNS) arithmetic [30], which provides an attractive alternative to floating-point when precision requirements are not very high.

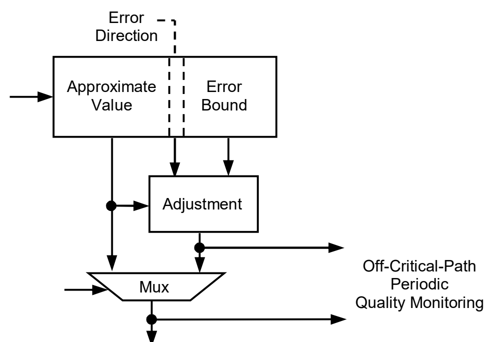

Fig. 2. Table-lookup scheme for approximate computing.

V. Variations and Optimizations

The basic benefits discussed in Section IV can be expanded by combining simple table-lookup with general and application-specific optimizations and design tricks. We already have a wide collection of pre- and post-processing methods to help reduce the table size, as mentioned near the beginning of Section II. Reducing binary arithmetic operations to intermediate or auxiliary unary operations [2] greatly reduces the table size. Bit- or digit-serial operations, a la distributed arithmetic, is another generally applicable strategy.

Other techniques reviewed in Section II must be studied with regard to relevance to and impact on approximate computing. An example is depicted in Fig. 3, which shows a possible adaptation of multi-level tables to approximate addition. The k-bit x and y operands are divided into u-bit upper parts, m-bit middle parts, and $(k - u - m)$-bit remainder or tail parts. A direct table for addition would need 2^{2k} entries. The scheme shown in Fig. 3 requires a level-1 table of size 2^{2u} and a level-2 table with 2^{u+2m} entries, which are collectively much smaller. As is the case for nearly all tabular function-evaluation schemes, pre-, mid-, and post-processing logic can be inserted on various data paths of Fig. 3 in an effort to further reduce the table size or to optimize the resulting accuracy.

Lookup-table entries need not be filled at the outset, making the computational tables behave like caches. The first use of each table entry would lead to a miss, which invokes a computation to fill the entry. In this way, only the table entries actually used are ever filled, leading to potential resource savings. This would be helpful in the case of large tables that are sparsely used. Going a step further, there may not be a need to set aside table entries for all possible input parameter values. Each time $f(\mathbf{x})$ is computed, the table entry $<\mathbf{x}, f(\mathbf{x})>$ is stored in a cache-like structure. The resulting scheme will then resemble memo-caches used in speculative execution methods [31] [32].

Our discussion in this paper has been almost exclusively in terms of hardware implementation. To provide a more complete picture, we need to touch upon the roles played by software and algorithms in approximate computing in general, and table-based methods in particular. Beginning with characteristics that make certain applications in AI and other domains inherently error-resilient, algorithm and software designers can devise computation scheme that lead to higher speed and lower energy consumption. Special hardware assists may or may not be required, but as usual, the greatest benefits are accrued when hardware and software strengths are combined.

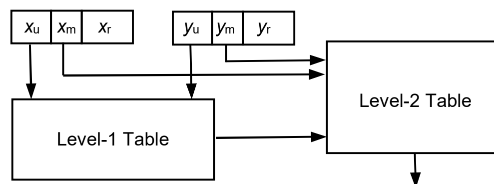

Fig. 3. A possible 2-level tabular scheme for approximate addition.

VI. Conclusion

There has been much work on approximate computing as a way to rein in the excessive complexity and waste of chip real estate and energy resources for providing extra precision where it is not needed. We have argued that tabular computation has a natural affinity with the aims of approximate computing, offering benefits in hardware speed, complexity, and energy consumption.

Research can continue in several directions. First, we should endeavor to assess the applicability of the notions enumerated at the end of Section II to making approximate computing more cost- and energy-efficient. A possible use of 2-level table-lookup was discussed in Section V. We are now working on nonuniform segmentation, distributed arithmetic, bit-level optimization, iterative refinement, bipartite and multipartite tables, and other schemes to come up with design strategies and assessing the resulting cost and energy benefits.

Some work can be carried out based on computational efficiencies in general, but deeper studies that take the requirements of specific application domains into account are needed to advance the state of the art. Of particular interest are applying the methods enumerated in Section II to the design of various kinds of neural networks.

We are also contemplating the use of mixed analog-digital representations [33] that combine the benefits of simple, low-energy analog processing with high precision of digital representations, to be invoked when needed. Two such representations are candidates for consideration: Continuous-digit number systems [34] [35] and residue number representations with analog digits [36].

References

[1] *CRC Standard Mathematical Tables and Formulae*, now in its 33rd ed., CRC Press, 2018.

[2] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, 2nd ed., 2010.

[3] B. Parhami, "Tabular Computation," *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya (eds.), Springer, 2019.

[4] B. Parhami, "The Return of Table-Based Computing," *Proc. 52nd Asilomar Conf. Signals, Systems, and Computres*, October 2018, to appear in early 2019.

[5] J. C. McCallum, "Graph of Memory Prices Decreasing with Time (1957-2017)," www.jcmit.net/mem2015.htm

[6] C. A. Mack, "Fifty Years of Moore's Law," *IEEE Trans. Semiconductor Manufacturing*, Vol. 24, No. 2, pp. 202-207, May 2011.

[7] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys*, Vol. 48, No. 4, p. 62, 2016

[8] B. Vinnakota, "Implementing Multiplication with Split Read-Only Memory," *IEEE Trans. Computers*, Vol. 44, pp. 1352-1356, 1995.

[9] A. S. Noetzel, "An Interpolating Memory Unit for Function Evaluation: Analysis and Design," *IEEE Trans. Computers*, Vol. 38, No. 3, pp. 377-384, 1989.

[10] D. U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung, "Nonuniform Segmentation for Hardware Function Evaluation," *Proc. 13th Int'l Conf. Field-Programmable Logic and Applications*, LNCS, Vol. 2778, Springer, pp 796-807, 2003.

[11] S. A. White, "Application of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. 6, No. 3, pp. 4-19, 1989.

[12] B. Parhami and C. Y. Hung, "Optimal Table Lookup Schemes for VLSI Implementation of Input/Output Conversions and Other Residue Number Operations," *Proc. IEEE Workshop VLSI Signal Processing*, pp. 470-481, 1994.

[13] Cray Research, "Cray 2 Computer System Functional Description Manual," Cray documentation, 1989.

[14] B. Parhami, "Modular Reduction by Multi-Level Table Lookup," *Proc. 40th Midwest Symp. Circuits and Systems*, Vol. 1, pp. 381-384, 1997.

[15] O. Gustafsson and K. Johanson, "Multiplierless Piecewise Linear Approximation of Elementary Functions," *Proc. 40th Asilomar Conf. Signals, Systems, and Computers*, pp. 1678-1681, 2006.

[16] D. Das Sarma and D. W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 17-28, 1995.

[17] M. J. Schulte and J. E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. Computers*, Vol. 48, No. 8, pp. 842-847, 1999.

[18] J. E. Stine and M. J. Schulte, "The Symmetric Table Addition Method for Accurate Function Approximation," *J. VLSI Signal Processing*, Vol. 21, pp. 167-177, 1999.

[19] J.-M. Muller, "A Few Results on Table-Based Methods," *Reliable Computing*, Vol. 5, No. 3, pp. 279-288, 1999.

[20] F. De Dinechin and A. Tisserand, "Multipartite Table Methods," *IEEE Trans. Computers*, Vol. 54, No. 3, pp. 319-330, 2005.

[21] P. Kornerup and D. W. Matula, "Single Precision Reciprocals by Multipartite Table Lookup," *Proc. 17th IEEE Symp. Computer Arithmetic*, pp 240-248, 2005.

[22] P. T. P. Tang, "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th IEEE Symp. Computer Arithmetic*, pp. 232-236, 1991

[23] IEEE Standard for Floating-Point Arithmetic, IEEE-754-2008 (currently in final stages of evaluation for an updated 2018 version).

[24] V. Lefevre and J.-M. Muller, "Correctly Rounded Functions for Better Arithmetic," *Proc. 34th Asilomar Conf. Signals, Systems, and Computers*, pp. 875-878, 2000.

[25] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On Reconfiguration-Oriented Approximate Adder Design and Its Application," *Proc. IEEE/ACM Conf. Computer-Aided Design*, pp. 48-54, 2013.

[26] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," *Proc. IEEE Int'l Conf. VLSI Design*, pp. 346-351, 2011.

[27] J. R. Shewchuk, "Robust Adaptive Floating-Point Geometric Predicates," *Proc. 12th ACM Symp. Computational Geometry*, pp. 141-150, 1996.

[28] D. Michelucci and J. M. Moreau, "Lazy Arithmetic," *IEEE Trans. Computers*, Vol. 46, No. 9, pp. 961-975, 1997.

[29] A. R. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*, World Scientific, 2007.

[30] M. Chugh and B. Parhami, "Logarithmic Arithmetic as an Alternative to Floating-Point: A Review," *Proc. 47th Asilomar Conf. Signals, Systems, and Computers*, pp. 1139-1143, 2013.

[31] F. Gabbay and A. Mendelson, "Using Value Prediction to Increase the Power of Speculative Execution Hardware," *ACM Trans. Computer Systems*, Vol. 16, No. 3, pp. 234-270, 1998.

[32] G. Tziantzioulis, N. Hardavellas, and S. Campanoni, "Temporal Approximate Function Memoization," *IEEE Micro*, Vol. 38, No. 4, pp. 60-70, July/August 2018.

[33] B. Parhami, "Analog Representations in Digital Arithmetic: A Review," *Proc. 52nd Asilomar Conf. Signals, Systems, and Computers*, 2018.

[34] A. Saed, M. Ahmadi, and G. A. Jullien, "A Number System with Continuous Valued Digits and Modulo Arithmetic," *IEEE Trans. Computers*, Vol. 51, No. 11, pp. 1294-1305, 2002.

[35] M. Mirhassani, M. Ahmadi, and G. A. Jullien, G.A., "Digital Multiplication Using Continuous Valued Digits," *Proc. Int'l Symp. Circuits and Systems*, pp. 3263-3266, 2007.

[36] B. Parhami, "Digital Arithmetic in Nature: Continuous-Digit RNS," *Computer J.*, Vol. 58, No. 5, pp. 1214-1223, May 2015.