The Return of Table-Based Computing

Behrooz Parhami

Dept. of Electrical and Computer Engr., Univ. of California, Santa Barbara, USA; parhami@ece.ucsb.edu

Abstract

Tabular computing, quite common in the pre-electronic-computer days, is regaining importance, given substantial reduction in the cost of memory and higher computational requirements in the age of big data. An added bonus of table-based computing is greater control over the error characteristics of the results, which favors the use of tables over alternative modes of approximate computing. After presenting examples of architectural schemes and applications for table-based computing, we conclude with an assessment of cost, performance, and energy benefits.

Keywords—Approximation; Bipartite/multipartite table; Computer arithmetic; Interpolation; Memory technology; Multi-level table; Residue number system.

1. Introduction

Until the 1970s, when compact and affordable digital scientific calculators became available, we relied on published pre-calculated tables of important functions [1]. For example, base-10 logarithm of values from 1 to 10, at increments of 0.01, might have been given in a 900-entry table, allowing direct read-out of values if low-precision was acceptable or use of linear interpolation to obtain greater precision. To compute log 35.419, say, one notes that it is 1 + log 3.5419 = 1 + log 3.54 + *r*, where log 3.54 is read out from the said table and *r* is derived based on the small residual 0.0019 using some sort of approximation or interpolation. The use of tables fell out of favor, once everyone became equipped with a sophisticated calculator and, later, with a computer.

Table-based computation returned in at least two different forms in the 1990s. One was to speed up normal, circuit-based computations by providing an initial estimate that would then be refined. The other was to reduce complexity and power consumption [2].

2. Essentials of Tabular Computation

Simultaneously with the exponential growth of data production rates, bringing about the age of big data [3], we have been experiencing an exponential reduction in memory cost, as depicted in Fig. 1 [4]. This trend has rendered big-data applications feasible, while also enabling new categories of applications, which would not even be contemplated were it not for inexpensive storage. One such area is increased reliance on large-scale tables for performing or facilitating computation.

Computing a function $f(\mathbf{x})$, where $\mathbf{x}$ is the parameter vector, requires time and other resources, such as energy. If a particular function value is needed many times in the course of different computations, it makes sense to store the computed value and use it when needed. Storage can be accomplished via conventional tables that are accessed by operand values used as index into the table or may entail some form of cache structure that is consulted before triggering the requisite calculations to fill the needed entry in the event of a cache miss.

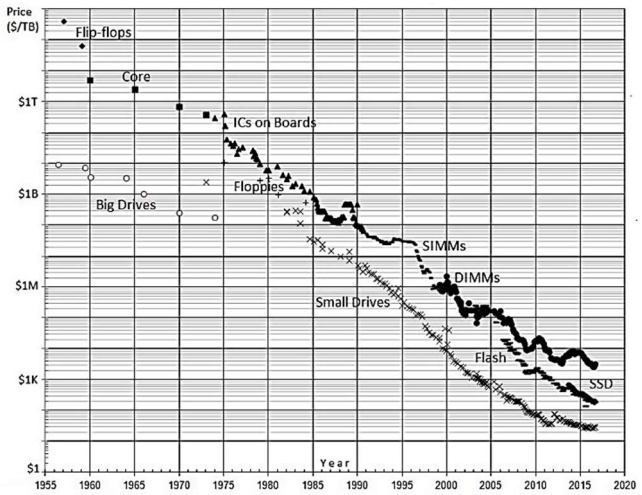

Fig. 1 Exponentially declining memory cost [4].

Table size can be reduced by subjecting the operands to some pre-processing or by allowing some post-processing of the value(s) read out from the table(s). This kind of indirect table lookup provides a range of trade-offs between pure table-lookup and pure computational or circuit-based approach. For example, the two-operand multiplication operation $p(a, b) = ab$ can be performed via two accesses to a smaller squaring table, using the formula $ab = [(a + b)^2 - (a - b)^2]/4$, whose use entails two additions in the pre-processing phase, and one addition along with a 2-bit right-shift in the post-processing stage.

Reading stored values requires less energy than recomputing them, particularly with emerging non-volatile memory technologies [5], and reading may also be faster, particularly for a computationally complex function. Because table size grows exponentially with the number of bits in the input operand(s), table lookup is particularly efficient for low-precision data, although continual increase in size and reduction in cost of memory is expanding the method's applicability.

Low-precision computation, as embodied in the rapidly expanding field of approximate computing [6], is an increasingly important part of the workload in modern computing. One advantage of table-based approximate computing is that the exact error for each table entry is knowable, whereas in a circuit-based approach, often a bound on the error is the best we can provide.

3. Multi-Table Interpolating Memory

The interpolating memory scheme [7] represents a hardware realization of the standard interpolation method for computing the value of the function $f(x)$, for $a < x < b$, given the values of $f(a)$ and $f(b)$. The scheme can be optimized to require less hardware, while offering desired error characteristics. Linear interpolation requires using 2 tables (function value and slope) along with a multiplier and an adder, while quadratic interpolation needs 3 tables, a squarer, two multipliers, and two adders.

Trade-offs exist between data-path widths and cost, error, and latency. Consider the computation of $y = \log_2 x$, where h upper bits of x, that is $x_{k-1:k-h}$, are used as an index into 2^h-entry tables, reading out y_{approx}, which is then refined using 1st-, 2nd-, or 3rd-degree interpolation. Error curves shown in Fig. 2 indicate that it is rarely advantageous to go beyond linear interpolation for achieving an overall error of 10^{-7}, say. The table size reduction from 2^{10} to 2^6 or 2^4 wouldn't offset the added hardware needed for higher-degree interpolation.

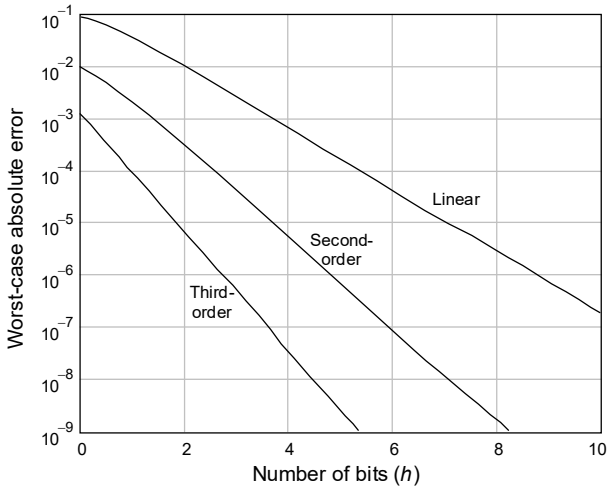

Fig. 2 Lookup table accuracy-cost trade-offs [7].

The curves for other functions of common interest are quite similar to those shown in Fig. 2 for $y = \log_2 x$, save small upward or downward shifting. In most cases, the required data-path width is within ±1 of that needed for the logarithm function, making it quite feasible to design a general-purpose interpolating memory unit, which can be customized for needed functions by plugging in ROMs with the required contents or by dynamically loading its RAM tables as appropriate.

4. Two Other Multi-Table Methods

In the three decades following the formulation of the interpolating memory concept discussed in Section 3, new methods and variations for tabular function evaluation have been proposed. In this section, we review two other classes of methods based on using multiple tables. In Section 5, we briefly review a few other methods.

Bipartite tables [8] obviate the need for a multiplier in linear interpolation through a clever trick, thus saving computation time and energy. Consider a k-bit operand x 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 $g(x_u, x_m) + h(x_u, x_r)$. The decomposition of x into x_u, x_m, and x_r creates intervals corresponding to different values of x_u and subintervals associated with different values of x_m. Function values are stored for each subinterval in the table $g(x_u, x_m)$. Instead of storing slopes for the various subintervals, as in interpolating memory, a common slope for each interval is stored (Fig. 3), allowing the multiplication of the slope $s(x_u)$ and the displacement x_r to be performed by the second lookup table, yielding the value $h(x_u, x_r)$.

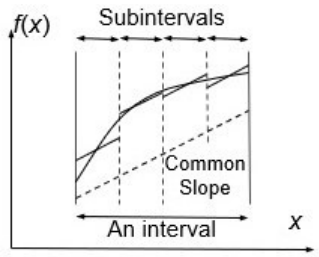

Fig. 3 The bipartite table-lookup concept.

Selection of the number u of bits comprising x_u and m forming x_m offers various tradeoffs in precision and table size, allowing the designer to choose an optimal scheme, given application requirements. Ignoring the second-order effect of different table widths on cost, and focusing only on the number of table entries, the bipartite table lookup method reduces the naïve table size of 2^k to $2^{u+m} + 2^{k-m}$, which is, very roughly speaking a factor of 2^m smaller. Much additional work has been done on optimizing bipartite tables (e.g., [9]) and on extending the concept to multipartite tables [10].

Both interpolating memory and bipartite tables use two different tables in their computation scheme. A different way of using two tables is to have one table feed the other one, instead of the two being accessed in parallel. Many configurations, with and without pre-, mid-, and post-processing, are possible [11]. A simple example is shown in Fig. 4. The level-1 table provides the approximate sum of x and y by considering only the upper u bits of each operand. This approximate sum can be refined by the level-2 table, which also considers the next m bits of the operands. Compared with a single table of size $2^{2(u+m)}$ for considering the said bits at once, the 2-level architecture requires a total table size of $2^{2u} + 2^{u+2m}$, which is, in very rough terms, a factor of 2^u smaller.

By adding a multiplexer to the design, as shown at the bottom of Fig. 4, the 2-level tabular scheme just discussed can become a dual-precision adder in which the level-2 table is circumvented when the precision of the level-1 table alone would do, thus saving time and energy.

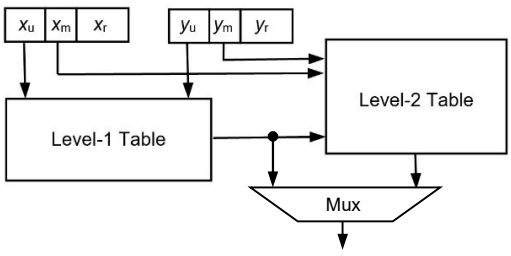

Fig. 4 Two-level lookup table for approximate sum.

5. Other Implementation Options

Here is a brief listing of several other methods and refinements to methods already discussed.

Nonuniform segmentation: The interpolating memory scheme of Section 3 divides the function evaluation domain into uniform segments, defined by the number h of high-order operand bits used to address the tables. Pre-processing can be used to divide the said domain into nonuniform segments [12], the idea being that we need fewer segments in regions where the function varies slowly or smoothly and more segments where the function is ill-behaved and exhibits rapid changes. This approach reduces the required table size, while also increasing computational accuracy.

Distributed arithmetic: Linear functions of the form $a_1x_1 + a_2x_2 + \ldots + a_nx_n$, involving the constants a_i are known constants, can be evaluated via bit-serial table-lookup [13]. 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 input 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 right-shifted running sum. 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 endeavor to devise an evaluation method that is optimized at the bit level. Distributed arithmetic, just discussed, is one instance of this approach. Other examples exist [14].

Iterative refinement: Many functions 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 an 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 of at most 1/2, the error can be reduced successively 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., [15]).

Approximate computing: Even though approximate computing represents a class of applications, not one specific method, its affinity with tabular calculation make it worth a mention here [16].

6. Evaluation and Comparisons

When we go from high-precision to lower-precision computing, circuit complexity tends to decrease polynomially as a function of the data-path width, whereas table size shrinks exponentially, thus making the tabular scheme more attractive below a certain precision. The exact crossover point is highly technology-dependent and not much can be deduced in general.

We thus focus on the particular implementation of a function-evaluation unit, depicted in Fig. 5 [16], as a case study, and present a parametrized evaluation that provides insight via ranges for the various improvement factors of interest. The approximate function evaluator of Fig. 5 monitors computation errors in a unit that is situated off the critical path, in order not to slow down the maim operation. One can view the circuit as having two modes of operation. In normal, approximate mode, the unit operates at high speed and low energy. Should higher precision be needed, at the cost of slower operation and greater energy consumption, the evaluator can be brought in. One important feature of the design is its ability to offer directional rounding [17], using the error-direction bit, a functionality that is quite difficult to provide in circuit-based approximation schemes.

The error-monitoring unit can decide automatically whether the error tolerance limit has been exceeded and to effect switching to higher-precision computation. The scheme of Fig. 5 is a simple form of adaptive-precision arithmetic [18], which has found applications when floating-point errors much be kept in check. It is also related to the notion of lazy arithmetic [19], an approach to expending as little time and energy to a computation as possible, keeping a tab on result quality, and backtracking and re-evaluating, if needed.

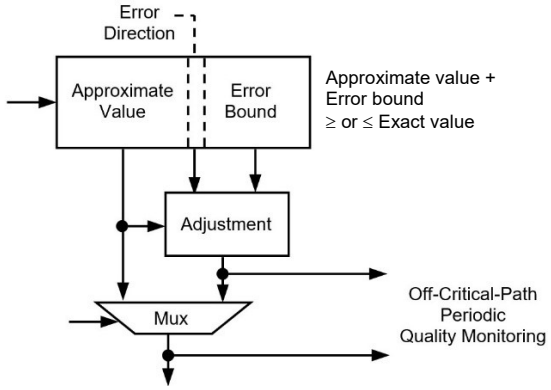

Fig. 5 Tabular scheme for approximate computing.

Let us take a memory cell to be equivalent to g NANDs in terms of chip area usage and energy consumption. This parameter varies for different kinds of memories as well as implementation technologies. It has been reported that one SRAM cell needs area equal to 0.5 gate. Perhaps one should pessimistically use $g = 1$ to account for address decoding and other peripheral/overhead circuity.

Let us take integer square-rooting for word widths up to 16 bits as a simple example. A pure table-lookup scheme requires tables of sizes $2^8 \times 4$ (complexity ~1K gates), $2^{12} \times 6$ (~25K), and $2^{16} \times 8$ (~524K), for 8-, 12-, and 16-bit inputs, respectively. For 8- and 12-bit inputs, these numbers compare favorably with the best reported results for FPGA realizations, as shown in Tables 1 and 2.

The reference for Table 1 [20] reports on 8- and 16-bit implementations of non-restoring array square-rooting (it also covers 32- and 64-bit operands, which are unsuitable for pure tabular realization). The implementations use a Xilinx Virtex E FPGA, with 1200 CLBs, 2400 LUTs, and 2M system gates. The 12-bit input entry has been derived by means of interpolation, based on supplied results (quadratic rise in hardware complexity, linear increase in latency, with word width k). Gate-equivalent estimates for hardware complexity are obtained from percentage utilization of FPGA resources cited by the authors and the total number of system gates.

The reference for Table 2 [21] offers a variety of methods with a range of cost-performance outcomes, when implemented on Xilinx Spartan E (with 0.5M system gates, 1164 slices, and 4-input LUTs), so I have quoted the paper's best results, which correspond to a non-restoring algorithm. Again, interpolation is used for 12-bit input. In this case, obtaining the gate-equivalent estimates was more difficult due to incomplete data, so the gate estimates cited must be used with care. The two sets of data are similar and serve to confirm each other.

Table 1 FPGA-based integer square-rooters [20].

Bits	CLBs	LUTs	Gates	Delay
8	12	21	~18K	15 ns
12	25	40	~37K	22 ns
16	42	73	~63K	40 ns

Table 2 FPGA-based integer square-rooters [21].

Bits	CLBs	LUTs	Gates	Delay
8	10	17	~12K	9 ns
12	22	39	~26K	20 ns
16	39	71	~47K	37 ns

7. Conclusion

Before the ubiquity of digital aids to computation, we used tables, published in reference books, to directly read out a reasonably accurate value for a function of interest, with additional precision obtainable through interpolation. The availability of cheap, high-speed digital circuits removed the need for the tables. But now, with energy consumption a primary concern and the proliferation of applications in which approximate values of functions will do, combined with the availability of inexpensive memories, table-based computing is both viable and useful [22]. One advantage of table-based approach to approximate computing is that the exact error for a table entry is knowable, whereas in a circuit-based method, often a bound on the error is the best we can provide. Tabular computation, particularly at low precisions, leads to speed, cost, and energy economy benefits.

A couple of final notes are in order: Throughout our discussion, we have assumed table access via what is sometimes referred to as "trivial hashing," that is, operand bits or simple transformations thereof (via pre-processing) are used as an index or address into the table(s). Clearly, more sophisticated hashing can be used to reduce the table size, when accesses to the table(s) are expected to be sparse. Additionally, if the table entry for $f(\mathbf{x})$ is stored along with input parameters as $\langle \mathbf{x}, f(\mathbf{x}) \rangle$ in an associative memory [23], already used successfully in packet routing and several other applications, significant storage savings can be achieved in the case of sparse accesses.

References

[1] D. Zwillinger, *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] C. L. P. Chen and C.-Y. Zhang, "Data-Intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data," *Information Sciences*, Vol. 275, pp. 314-347, 2014.

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

[5] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of Emerging Nonvolatile Memory Technologies," *Nanoscale Research Letters*, Vol. 9, No. 1, p. 526, 2014.

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

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

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

[9] 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.

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

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

[12] 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*, pp. 796-807, 2003.

[13] S. A. White, "Application of Distributed Arithmetic to Digital Signal Processing," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. 6, pp. 4-19, 1989.

[14] 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*, 1994.

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

[16] B. Parhami, "A Case for Table-Based Approximate Computing," *Proc. 9th IEEE Information Technology, Electronics & Mobile Communication Conf.*, Vancouver, Canada, November 2018, to appear.

[17] IEEE Standard for Floating-Point Arithmetic, IEEE-754-2008 (in final stages of revision for an updated version).

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

[19] D. Michelucci and J. M. Moreau, "Lazy Arithmetic," *IEEE Trans. Computers*, Vol. 46, pp. 961-975, 1997.

[20] S. Samavi, A. Sadrabadi, and A. Fanian, "Modular Array Structure for Non-Restoring Square Root Circuit," *J. Systems Architecture*, Vol. 54, No. 10, pp. 957-966, 2008.

[21] A. P. Ramesh and I. J. Kumar, "Implementation of Integer Square Root," *Int'l J. Engineering Science and Innovative Technology*, Vol. 4, No. 1, pp. 105-113, January 2015.

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

[23] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE J. Solid-State Circuits*, Vol. 41, No. 3, pp. 712-727, 2006.