T

Tabular Computation

Behrooz Parhami
Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA

Synonyms

Approximate computation; Iterative refinement; Table lookup

Definition

Big data necessitates the use of very large memories that can bring about other uses of such units, say, for storing precomputed values of functions of interest, to improve speed and energy efficiency.

Overview

Until the 1970s, when compact and affordable digital scientific calculators became available, we relied on pre-calculated tables of important functions that were published in book form (e.g., Zwillinger 2011). For example, base-10 logarithm of values in [1, 10], at increments of 0.01, might have been given in a 900-entry table, allowing direct readout of values if low precision was acceptable or use of linear interpolation to obtain greater precision. To compute log 35.419, say, one would note that it is $1 + \log 3.5419 = 1 + \log 3.54 + \varepsilon$, where log 3.54 is read out from the said table and ε is derived based on the small residual 0.0019 using some sort of approximation or interpolation. Once everyone became equipped with a sophisticated calculator and, later, with a personal computer, the use of tables fell out of favor. 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.

Implications of High Volume for Big Data

Studies at IBM and Cisco Systems show that as of 2012, we generated 2.5 exabytes of data per day (Cisco Systems 2017; Jacobson 2013) [1 exabyte = 1 quintillion or 10^{18} bytes = 1 gigagigabyte], and this is set to explode to 40 yottabytes per day [1 yottabyte = 1 septillion or 10^{24} bytes = 1 teraterabyte] by 2020. Multiply the latter number by 365 and then by the number of years, and the extent of data becomes even more mind-boggling. Organizing, managing, and updating such large volumes of data will be quite challenging.

A direct consequence of high data volume is high computational load, as confirmed by a sim-

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_169-1

ple back-of-the-envelope calculation. Processing 1 petabyte of data, at around 100 clock cycles or instructions per byte, means a processing time of 10^{17} clock cycles. With an optimistic clock rate of 10 GHz (10^{10}), we will need a processing time of 10^7 s ($\sim$4 months). This is already impractically long, and it becomes even more so for data volumes much larger than a petabyte. It follows that replacing many cycles of computation with a single table lookup will offer immense performance benefits.

Simultaneously with the exponential growth of data production rate (Chen and Zhang 2014; Hilbert and Gomez 2011), we have been experiencing an exponential reduction in the cost of computer memory and storage, as depicted in Fig. 1 (McCallum 2017). The reduced cost of memory and storage, combined with more advanced data compression schemes (Storer 1988), renders big-data applications feasible while also enabling new categories of applications, which would not even be contemplated in the absence of inexpensive storage. One such area is increased reliance on large-scale tables for performing or facilitating computation.

Why Table-Based Computation?

Function evaluation methods are studied in the field of computer arithmetic (Parhami 2010), dealing with algorithms and hardware designs for performing computations within arithmetic/logic units found as components of CPUs in general-purpose computer architectures (Parhami 2005), as well as within more specialized computational structures, such as graphic processing units or GPUs (Owens et al. 2008).

Computing any function requires time and other resources, such as energy. If a particular function value, say, $f(a)$, is needed many times in the course of different computations or the same computation performed on different inputs, it makes sense to store the computed value and use it any time $f(a)$ is needed, without having to recompute. 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 (Smith 1982) that is consulted before triggering the requisite calculations in the event of a cache miss (Gabbay and Mendelson 1998).

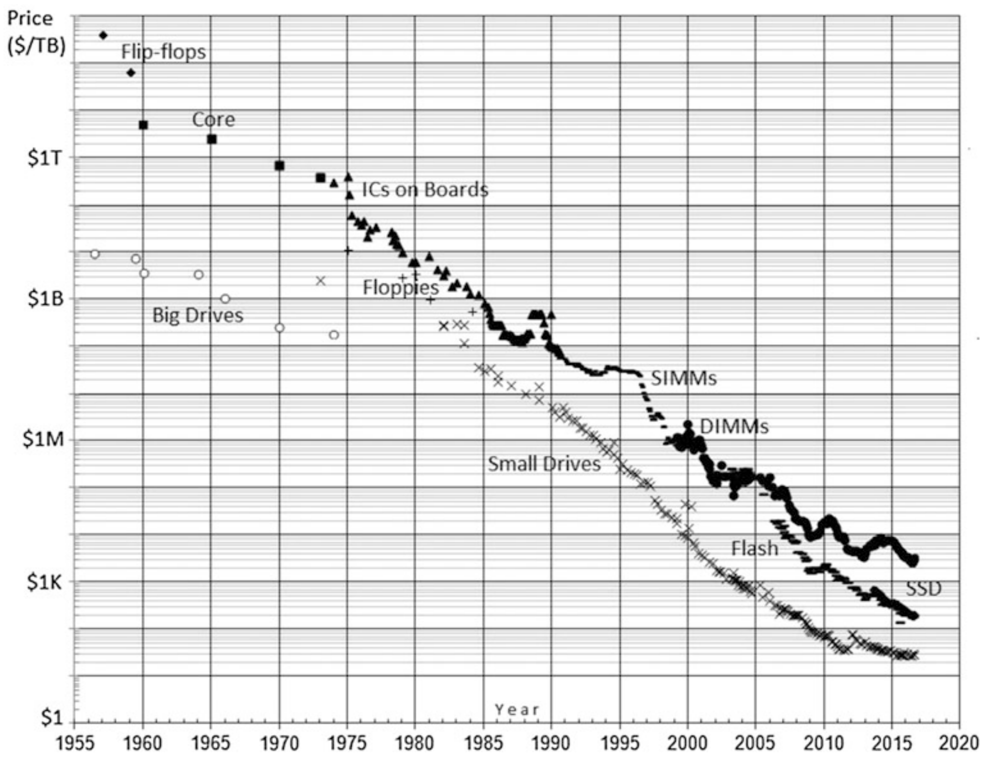
One way to reduce the required table size is to subject the operands to some preprocessing or to allow some post-processing of the value(s) read out from the table(s). This kind of indirect table lookup provides a range of trade-offs between pure table lookup and pure computational or circuit-based approach. 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 preprocessing phase and one addition along with a 2-bit right shift in the post-processing stage. Additional optimizations can be applied to this method (Vinnakota 1995).

Reading a stored value requires less energy than recomputing it, 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), the scheme 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 (Mittal 2016), 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.

Lookup Table Implementation Options

The conceptually simplest method for implementing a lookup table is to allocate 2^u words, each v bits wide, if there are u input operand(s) bits and the result is to be v bits wide. The table can then be filled with precomputed function values, essentially resulting in an electronic version



Tabular Computation, Fig. 1 Exponentially declining cost of computer memory and storage (McCallum 2017)

of the mathematical tables of yore found in pages of handbooks.

This is rarely practical or efficient. One optimization is to allocate space for all table entries but only fill them when the values become needed for the first time. In this way, the table will act as a sort of cache memory which will generate a "miss" upon first access to a particular entry. When a miss is encountered, the requisite computation is performed to derive the value, which is then forwarded to both the computation that led to the miss and to the corresponding table entry for possible future use.

A second optimization is to look for simple transformations in the input operands that can be performed efficiently in terms of time and energy while leading to substantial savings in table size. If a two-operand function is symmetric, for example, the table size can be cut in half by determining which operand is larger and ordering the operands. Such optimizations are function-dependent and provide a wide range of trade-offs between computation time and lookup table size.

A third optimization is to divide the domain of interest into a number of subranges and implement a separate lookup scheme for each subrange. For subranges where the function variations are less pronounced, smaller tables will do, whereas subranges with significant variations necessitate larger lookup tables. Related to this last optimization is segmenting the domain of interest in a nonuniform manner, using a special mechanism to map input values into corresponding subranges and thus associated lookup tables (Lee et al. 2003).

An attractive alternative to parallel-access tables, with original or transformed operand bits presented in parallel to form an index into the table, is a combination of bit-serial computation with table lookup. The scheme, known as distributed arithmetic (White 1989), allows certain computations to be implemented through table lookup, despite the large number of input values involved. The speed of distributed arithmetic can be respectable and the implementation cost modest if the computation is pipelined at a high clock rate. It is also possible to do intermediate schemes where the computation is not fully serial, but nibble- or byte-serial.

Table-based methods can be implemented in a variety of ways and new schemes continue to emerge in light of ongoing research. To keep our discussion manageable, we focus on the basics of two general methods, the interpolating memory scheme and multipartite table method, in the following two sections. More details on these methods, including additional references, as well as other pure and hybrid methods can be found in books, periodicals, and conferences on computer arithmetic (e.g., Parhami 2010).

Interpolating Memory

One can combine the lookup process for reading out approximate function values with the interpolation scheme needed for greater precision into a single-hardware unit, dubbed interpolating memory (Noetzel 1989).

Consider the computation of $f(a)$, where the parameter a falls between x_i and x_{i+1}, two consecutive lookup table indices, that is, $x_i < a < x_{i+1}$. Then, using linear interpolation, the value of $f(a)$ can be approximated as

$$f(a) = f(x_i) + (a - x_i) \times f'(x_i)$$

where $f'(x_i)$ is the derivative or slope of the function at x_i. More generally, the values $A_i = f(x_i)$ and $B_i = f'(x_i)$, or a value that better approximates intermediate values between $f(x_i)$ and $f(x_{i+1})$, are stored, with a post-processing circuit computing $f(a) = A_i + (a - x_i) \times B_i$.

Practically, x_i corresponds to several leading bits of a and $a - x_i$ is represented by the remaining bits of a. This scheme simplifies the implementation and reduces its complexity. Let a_u be defined by a few upper bits of a, and let a_r be the value represented by the remaining bits of a, so that a_u and a_r collectively span all the bits of a. Then, $f(a) = f(a_u) + s(a_u) \times a_r$.

One can increase the computational precision by using second- or third-degree interpolation, which, for the same precision, would need smaller tables (Noetzel 1989) but requires more external circuitry, providing a spectrum of implementation options. For example, with second-degree interpolation, we have $f(a) = f(a_u) + s(a_u) \times a_r + t(a_u) \times a_r^2$. With the latter formula, a squarer and two multipliers are needed for highest-speed implementation. Alternatively, one can use a single multiplier to perform the operation $a_r \times a_r$ and the other two multiplications sequentially. If the function must be evaluated for several different x values, the computation can be pipelined, so that the maximum processing rate is achieved while using only a single multiplier and one adder.

Bipartite and Multipartite Tables

Interpolating memory, discussed in the preceding section, requires the use of a multiplier to achieve the greater precision afforded by linear interpolation. Multiplication is time-, cost-, and energy-intensive compared with addition, so table lookup schemes that avoid multiplication would be desirable if they offer adequate precision using only addition as pre- and post-processing operations. Schemes in this category are known as multiplier-less methods (e.g., Gustafsson and Johanson 2006).

Bipartite tables (Das Sarma and Matula 1995) 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, collectively spanning the entire width of the operand x. We can approximate $f(x)$ as the sum $g(x_u, x_m) + h(x_u, x_r)$. Essentially, the decomposition into x_u, x_m, and x_r creates intervals corresponding to different values of x_u and subintervals corresponding to different values of x_m. 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 various tradeoffs in precision and table size, allowing the designer to choose an optimal scheme, given application requirements.

The basic ideas discussed above have been extended and refined in a variety of ways. Bipartite tables can be optimized by taking advantage of symmetry (Schulte and Stine 1999; Stine and Schulte 1999). Extension to tripartite (Muller 1999) and multipartite tables (De Dinechin and Tisserand 2005; Kornerup and Matula 2005) has also been attempted.

Iterative Refinement Methods

Table lookup can be used in conjunction with iterative refinement methods to obtain results of higher precision that are impossible or cost-ineffective to derive via pure table lookup. The iterative refinement process essentially replaces an interpolation scheme that would be much more complex, if the same precision were to be achieved. We present just one example, but there are many more.

Consider finding the square root of z, starting with an initial approximation $x(0)$. The following iterative scheme yields an approximation $x(j + 1)$ with maximum error 2^{-2k} from a previous approximation $x(j)$ with maximum error 2^{-k}:

$$x(j + 1) = ½(x(j) + z/x(j))$$

The scheme's quadratic convergence ensures that a small number of iterations will suffice. For example, if an initial table entry provides the square root of z with 16 bits of precision, one iteration produces a 32-bit result and two iterations will yield a 64-bit result. The method just discussed has many variations and alternatives, including versions that are division-free (Cray Research 1989), given that division is a slower and more complicated operation compared with multiplication.

Future Directions

After decades of being restricted to low-precision arithmetic, table lookup is emerging as a feasible method of attack to higher-precision computation, aided by the lower cost and higher density of memory devices as well as improved algorithms for reducing the required table size via advanced pre- and post-processing schemes.

In addition to general techniques discussed in the preceding sections, a large number of special algorithms and associated tweaks can be used to make implementations more efficient or better-suited to specific application domains. Methods such as multilevel table lookup (Parhami 1997); bit-level, application-specific optimization of lookup tables (Parhami and Hung 1994); and accurate error analysis (Tang 1991), accompanied by further reduction in memory cost and increase in memory density, will enable an expansion of application domains that lend themselves to table lookup processing. Advances in lookup table size and performance may also lead to the adoption of logarithmic number representation (Chugh and Parhami 2013) in lieu of the currently prevalent floating-point representation, which would lead to additional side benefits.

Cross-References

▶ Energy Implications of Big Data
▶ Storage Hierarchies for Big Data
▶ Storage Technologies for Big Data

References

Chen CLP, Zhang C-Y (2014) Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci 275:314–347

Chugh M, Parhami B (2013) Logarithmic arithmetic as an alternative to floating-point: a review. In: Proceedings of 47th Asilomar conference on signals, systems, and computers, Pacific Grove, pp 1139–1143

Cisco Systems (2017) The Zettabyte Era: trends and analysis, White Paper, http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

Cray Research (1989) Cray 2 computer system functional description manual, Cray documentation

Das Sarma D, Matula DW (1995) Faithful Bipartite ROM reciprocal tables. In: Proceedings of 12th symposium on computer arithmetic, Bath, pp 17–28

De Dinechin F, Tisserand A (2005) Multipartite table methods. IEEE Trans Comput 54(3): 319–330

Gabbay F, Mendelson A (1998) Using value prediction to increase the power of speculative execution hardware. ACM Trans Comput Syst 16(3):234–270

Gustafsson O, Johanson K (2006) Multiplierless piecewise linear approximation of elementary functions. In: Proceedings of 40th Asilomar conference on signals, systems, and computers, Pacific Grove, pp 1678–1681

Hilbert M, Gomez P (2011) The world's technological capacity to store, communicate, and compute information. Science 332:60–65

Jacobson R (2013) 2.5 quintillion bytes of data created every day: how does CPG & retail manage it? IBM Industry Insights, http://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/

Kornerup P, Matula DW (2005) Single precision reciprocals by multipartite table lookup. In: Proceedings of 17th IEEE symposium on computer arithmetic. Cape Cod, pp 240–248

Lee DU, Luk W, Villasenor J, Cheung PYK (2003) Non-uniform segmentation for hardware function evaluation. In: Proceedings of 13th international conference on field-programmable logic and applications, Lisbon. LNCS, vol 2778. Springer, pp 796–807

McCallum JC (2017) Graph of memory prices decreasing with time (1957–2017). http://www.jcmit.net/mem2015.htm

Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4):62

Muller J-M (1999) A few results on table-based method. Reliab Comput 5:279–288

Noetzel AS (1989) An interpolating memory unit for function evaluation: analysis and design. IEEE Trans Comput 38(3):377–384

Owens JD et al (2008) GPU computing. Proc IEEE 96(5):879–899

Parhami B (1997) Modular reduction by multi-level table lookup. Proc 40th Midwest Symp Circuits Syst 1:381–384

Parhami B (2005) Computer architecture: from microprocessors to supercomputers. Oxford University Press, New York

Parhami B (2010) Computer arithmetic: algorithms and hardware designs, 2nd edn. Oxford University Press, New York

Parhami B, Hung CY (1994) Optimal table lookup schemes for VLSI implementation of input/output conversions and other residue number operations. In: Proceedings of IEEE workshop on VLSI signal processing VII, La Jolla, pp 470–481

Schulte MJ, Stine JE (1999) Approximating elementary functions with symmetric Bipartite tables. IEEE Trans Comput 48(8):842–847

Smith AJ (1982) Cache memories. ACM Comput Surv 14(8):473–530

Stine JE, Schulte MJ (1999) The symmetric table addition method for accurate function approximation. J VLSI Signal Process 21:167–177

Storer J (1988) Data compression. Computer Science Press, Rockville

Tang PTP (1991) Table-lookup algorithms for elementary functions and their error analysis. In: Proceedings of symposium on computer arithmetic, Bath, pp 232–236

Vinnakota B (1995) Implementing multiplication with split read-only memory. IEEE Trans Comput 44(11):1352–1356

White SA (1989) Application of distributed arithmetic to digital signal processing: a tutorial review. IEEE Trans Acoustics Speech Signal Process 6(3):4–19

Zwillinger D (2011) CRC standard mathematical tables and formulae, 32nd edn. CRC Press, Boca Raton