Parallelism in Computer Arithmetic: A Historical Perspective

(***Invited Paper***)

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

Abstract— **Many early parallel processing breakthroughs emerged from the quest for faster and higher-throughput arithmetic operations. Additionally, the influence of arithmetic techniques on parallel computer performance can be seen in diverse areas such the bit-serial arithmetic units of early massively parallel SIMD computers, pipelining and pipeline-chaining in vector machines, design of floating-point standards to ensure the accuracy and portability of numerically-intensive programs, and prominence of GPUs in today's top-of-the-line supercomputers. This paper contains a few representative samples of the many interactions and cross-fertilizations between computer-arithmetic and parallel-computation communities by presenting historical perspectives, case studies of state of art and practice, and directions for further collaboration.**

*Keywords— **Arithmetic algorithms, DSP, Graphic processors, Parallelism, Pipelining, Recursive circuits, Residue arithmetic***

I. Introduction

This paper revolves around the notion of parallelism, so a precise definition of the word in the context of computer arithmetic is called for. At one extreme, given that computer arithmetic, at least up until now, has dealt with binary or binary-encoded operands, and considering that fundamental circuit operations are performed at the bit level, any circuit that manipulates multiple bits at once is parallel. At this extreme, a half-adder that produces its carry and sum outputs independently, using an AND gate and an XOR gate, is viewed as doing parallel processing. This definition of parallelism is clearly counterproductive. At the other extreme, parallelism is viewed as being entirely outside the circuit realm, requiring concurrency at the level of large functional units (ALUs, CPUs, GPUs). This view, too, excludes some interesting and important examples in the domain of computer arithmetic.

I take an expansive view that parallel processing exists at all three levels of circuits, function units, and compute nodes, using these three varieties of parallelism to present my examples of technology transfer and cross-fertilization between computer arithmetic and parallel systems research in Sections II, III, and IV of the paper. Section V concludes the paper and points to possible future work.

II. Circuit-Level Parallelism

Circuit designs for adders and multipliers, the two main workhorses in numerical computation, show many signs of cross-fertilization with parallel-computing research. I outline only the parallel-prefix approach to addition in this section and will cover recursive multiplication in Section III, although the latter idea has circuit-level embodiments as well.

Parallel-prefix computation (PPC) refers to simultaneous evaluation of x_0, $x_0 \otimes x_1$, $x_0 \otimes x_1 \otimes x_2$, … , $x_0 \otimes x_1 \otimes \ldots \otimes x_{k-1}$, which form prefixes of the expression $x_0 \otimes x_1 \otimes \ldots \otimes x_{k-1}$, where $\otimes$ is an associative binary operator. This computation is a very useful building-block in constructing parallel algorithms and leads to many practical applications with different instantiations of the objects x_i and operator $\otimes$. For example, indexing/labeling the 1 elements in a bit-vector corresponds to a parallel-prefix-sums computation on the vector elements.

PPC was first studied by Ladner and Fisher [1], who presented a divide-and-conquer scheme that solved the problem with optimal $O(\log n)$ latency and $O(n \log n)$ circuit cost, and also showed how the circuit cost can be reduced to the optimal $O(n)$. Being theoretical computer scientists, Ladner and Fisher considered the problem solved when they produced a design that was both time- and cost-optimal, and they did not devote any effort to VLSI considerations such as fan-out.

Before outlining methods for dealing with wiring and fan-out problems, let me define the instance of PPC that corresponds to an adder's carry network. The addition of two radix-r operands $a_{k-1} \ldots a_1 a_0$ and $b_{k-1} \ldots b_1 b_0$ to determine their sum $s_{k-1} \ldots s_1 s_0$ entails the determination of intermediate carries $c_1, c_2, \ldots, c_{k-1}$; for simplicity, we will assume $c_0 = c_{\text{in}} = 0$ and ignore the need for producing $c_k = c_{\text{out}}$. Using the auxiliary generate $g_i = (a_i + b_i \geq r)$ and propagate $p_i = (a_i + b_i = r - 1)$ binary signals, and defining the carry operator © on auxiliary signal pairs (g_i, p_i) as (g'', p'') © $(g', p') = (g'' \vee g'p'', p'p'')$, the intermediate carries can be computed based on $c_i = g_{0:i-1}$, where $(g_{0:i-1}, p_{0:i-1})$, $i \in [1, k-1]$, are prefixes of the expression (g_0, p_0) © (g_1, p_1) © (g_2, p_2) © … © (g_{k-2}, p_{k-2}). Once the intermediate carries are known, the sum digits can be computed easily from $s_i = (a_i + b_i + c_i) \bmod r$.

The task of adapting PPC-based addition to the constraints of VLSI design fell to Brent and Kung, who outlined their approach in a 1979 CMU tech report, later published in early 1982 [2]. They stated that previous work on addition had focused on the goal of reducing the number of logic gates but paid little attention to the problem of connecting the gates. Their design sacrificed some speed, increasing the latency from $\log_2 k$ carry-operator levels to $2 \log_2 k - 2$ levels, in the interest of reducing the wiring complexity and the attendant area requirements. Their PPC network, later dubbed the Brent-Kung design, inspired much subsequent research, leading, among others, to an adaptation of Kogge's and Stone's parallel algorithm [3] to what became known as Kogge-Stone carry networks and a number of hybrid BK-KS designs [4], later dubbed Han-Carlson designs [5]. To these, one must add the parameterized designs of Knowles [6] and those based on the high-valency prefix cells of Beaumont-Smith and Lim [7].

Many studies on the tradeoffs between latency, VLSI area, and, later, energy consumption ensued [8]. One direction of research was to relax the extreme fan-out restriction of the Brent-Kung scheme to take advantage of small, constant fan-outs that are quite manageable in modern circuit design. Subsequent combination of the carry-lookahead scheme with the carry-select idea led to even more efficient hybrid VLSI adders. Modern adder designs are predominantly hybrid, taking advantage of the strengths of various schemes to shape designs that are optimal for the particular set of usage requirements.

I end this section with a particularly useful taxonomy of PPC-adders, shown in Fig. 1 assuming a word width of $k = 16$, in the 3D space of logic levels (beyond the minimum $\log_2 k$), wire tracks (power of 2), and fan-out (power of 2, plus 1), due to Harris [9]. We see that the previously-discussed designs occupy certain points in the 3D space of the taxonomy, hinting at other designs that occupy other points.

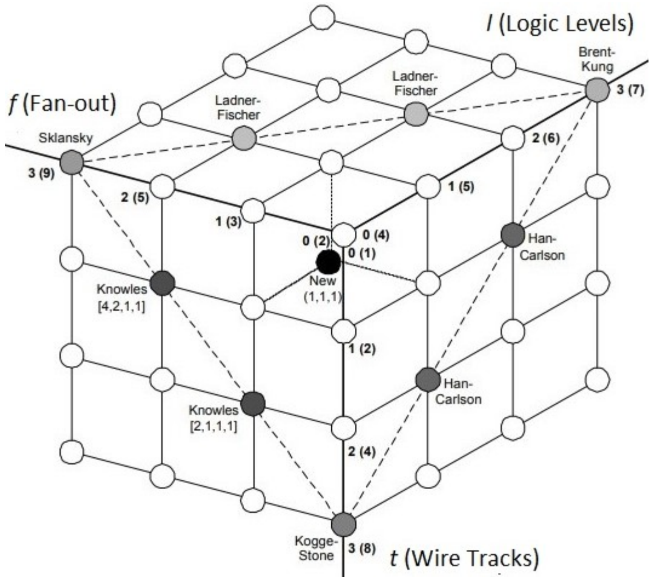

Fig. 1. Taxonomy of parallel-prefix networks, due to Harris [9].

III. Function-Level Parallelism

Many arithmetic computations are based on repeated invocation of basic building-block operations in time and/or space. Concurrency among the multiple invocations leads to pipelined and parallel implementations. A good example of function-level parallelism in terms of interaction and cross-fertilization with the parallel computing community is provided by recursive multiplication.

Recursive or divide-and-conquer multiplication [8] can be implemented in a variety of ways. For concreteness and simplicity, we focus on 2-way splitting, with power-of-2 operand widths. To compute the product xy, we split each of the two operands into high and low halves, resulting in $x = 2^{k/2}x_H + x_L$ and $y = 2^{k/2}y_H + y_L$. The product xy can then be computed from the outputs of four half-width multipliers via:

$$xy = (2^{k/2}x_H + x_L)(2^{k/2}y_H + y_L) = 2^k x_H y_H + 2^{k/2}(x_H y_L + x_L y_H) + x_L y_L$$

Designating the results of the four multiplications, from left to right in the expression above as p_1, p_2, p_3, and p_4, we have

$$xy = 2^k p_1 + 2^{k/2}(p_2 + p_3) + p_4$$

which implies 3 additions and 2 shifts to complete the process.

Time and area complexities of this recursive algorithm are provided by the following recurrences, assuming sequential computation of the four half-width products using a single half-width multiplier and counting bit-level operations:

$$T(k) = 4T(k/2) + \Theta(\log k) = \Theta(k^2)$$

$$A(k) = A(k/2) + \Theta(k) = \Theta(k)$$

With concurrent generation of the four partial products, using four half-width multipliers, as well as parallel operations throughout the process, the recurrences become:

$$T(k) = T(k/2) + \Theta(\log k) = \Theta(\log k)$$

$$A(k) = 4A(k/2) + \Theta(k) = \Theta(k^2)$$

Both schemes are better than the naïve pencil-and-paper or brute-force algorithm, and both are suboptimal with respect to the AT and AT^2 lower bounds $\Omega(k^{3/2})$ and $\Omega(k^2)$, derived by Brent and Kung [10].

Here is an improvement to the preceding recursive multiplication scheme, due to Karatsuba and Ofman [11]. Instead of forming p_2 and p_3 in the formulation above, form $p_5 = (x_H - x_L)(y_H - y_L)$. Then, compute the product xy from:

$$xy = 2^k p_1 + 2^{k/2}(p_1 + p_4 - p_5) + p_4$$

With the Karatsuba-Ofman trick and still counting bit-level operations, the area recurrence becomes

$$A(k) = 3A(k/2) + \Theta(k) = \Theta(k^{1.585})$$

where the exponent 1.585 is $\log_2 3$. Even though, in practice, the advantage of Karatsuba-Ofman multiplication algorithm does not kick in until k goes into 100s of bits, it provides major speedup for multiplying large numbers of the kinds needed in certain encryption schemes. Long multiplications are usually performed modulo-m, for some large integer m, creating the need for modular multiplication algorithms [12] [13].

After the reduction from $O(k^2)$ to $O(k^{1.585})$ discussed above, the exponent was reduced through further tricks, including those devised by Toom and cleaned up by Cook [14] [15] (1.465, 1.404, $1 + \varepsilon$) during the 1960s, later leading to the 1971 result of $\Theta(k \log k \log\log k)$ by Schonhage and Strassen [16] and another, thus far the best, theoretical result by Furer [17]. Whether the bit-level complexity of multiplication can be reduced to that of addition, that is, $\Theta(k)$, is an open question at this time. The improvements just discussed are partially graphed in Fig. 2 [18], where the "Toom" line corresponds to the first $\Theta(n^{1.465})$-time Toom-Cook algorithm. The graph is meant to illustrate only the general trends and crossover points; more precise comparisons with different assumptions may yield somewhat different results.

So, where is the connection of all this to the parallel computing community? The name "Strassen" in the preceding paragraph provides a hint. Strassen is credited with a matrix multiplication algorithm [19], which reduces the count of arithmetic operations in the multiplication of two $n \times n$ matrices from $\Theta(n^3)$ to $\Theta(n^{2.807})$, where 2.807 is $\log_2 7$, via a similar technique of avoiding some multiplications at the expense of introducing more additions.

It is interesting to ponder the question of why Strassen's method leads to relatively smaller savings in run-time compared with the Karatsuba-Ofman method. An intuitive explanation goes as follows. Strassen's method avoids 1 of 8 multiplications, versus Karatsuba-Ofman's 1 of 4. Also, looking beyond asymptotic complexities, while both integer addition and matrix addition are simpler than corresponding multiplication operations, the ratio $O(k^2)/O(k)$ of complexities for integer operations is larger than the $O(n^3)/O(n^2)$ for matrix operations, the problem size being k in the first case and n^2 in the second one. So, lurking behind asymptotic results is another potentially significant difference, when attempting to implement such recursive schemes.

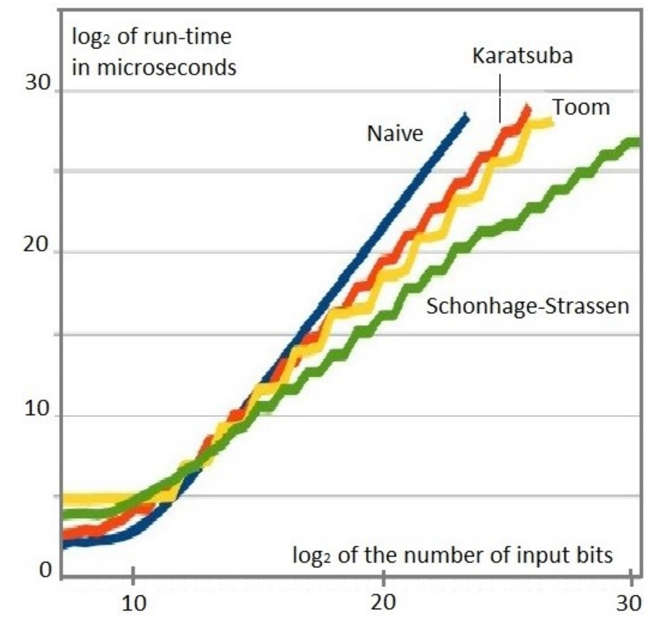

Fig. 2. Run-times of various multiplication algorithms for wide words [18].

Because the publication of Karatsuba-Ofman multiplication method pre-dates that of Strassen's matrix multiplication method by some 7 years, influencing, if it occurred, was in the direction opposite that of the PPC-addition of Section I. Given that Strassen later contributed to the theoretical study of the complexity of integer multiplication, it is safe to assume that he was inspired by Karatsuba's and Ofman's work in devising his matrix multiplication method [19].

IV. System-Level Parallelism

Parallelism at the system level takes the form of multiple independent or interacting arithmetic computation streams. Very early examples include arithmetic (often floating-point) co-processors that relieved the main CPU by performing arithmetic operations on demand and sending the results back to the CPU or to memory [20]. The modern embodiments of arithmetic co-processors are graphic processing units, or GPUs, that are optimized for carrying out a large number of arithmetic operations at high speed, without the overheads and unneeded "features" of a conventional CPU [21] [22] [23].

Let us take the computation of an n-point DFT as an example [24]. Given an n-element vector x as input, the output of DFT computation is the n-element vector y, such that

$$y_i = \sum_{j=0:n-1} \omega_n^{ij} x_j$$

where ω_n, a complex number, is a primitive nth root of unity. Straightforward computation of the y_is requires $\Theta(n^2)$ arithmetic operations (n sums, each involving n multiplications and n additions, with the powers of ω_n pre-computed and stored in a table). The contribution of the parallel algorithms community to the efficient computation of DFT is the Cooley-Tukey fast Fourier transform, or FFT, algorithm [25] [26], which takes advantage of the special structure of the computation, using divide-and-conquer to reduce the number of arithmetic operations from $\Theta(n^2)$ to $\Theta(n \log n)$.

As for hardware implementation, the processors in this example are much simpler than full-blown GPUs, because all they do is a butterfly operation: Receive inputs a and b, compute $a + b$ and $a - b$, and, finally, perform a multiplication by some constant c to form $c(a - b)$. A straightforward mapping of the computation to hardware leads to $n \log n$ butterfly processors, each of which performs a single butterfly operation [24]. Computer arithmetic and VLSI researchers have devised methods to save on the number of processors, while making each processor simpler and faster.

Let us first look inside each butterfly processor. There are two key savings that result from computer arithmetic research. The first is merged computation of the add and subtract operations, $a + b$ and $a - b$, using less hardware and energy compared with having two separate adders. The second is devising special methods for multiplication by constants [8], which entail less hardware (and, thus, less power) and increase the speed of forming $c(a - b)$, which is on the critical path within each processor. When the processors are shared, so that each processor needs to multiply by different constants in the course of its engagement, methods of multiplication by multiple constants can help reduce the complexity relative to the use of a full-blown multiplier.

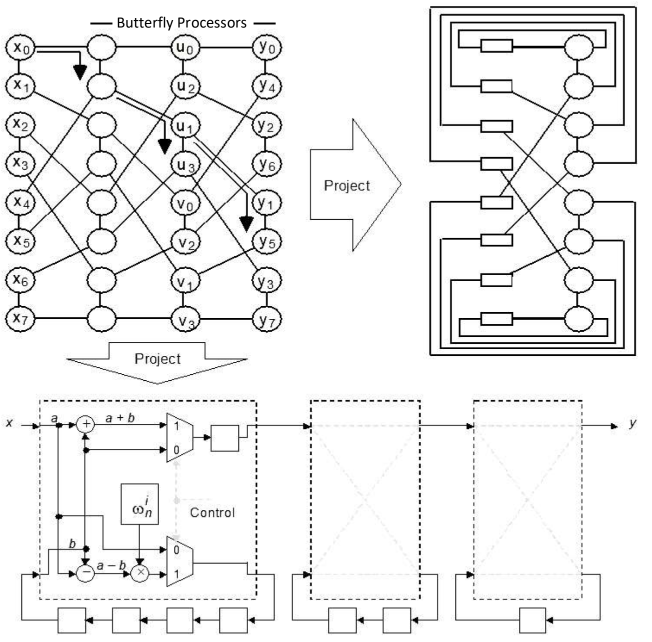

Fig. 3. A butterfly network and its projections onto *n* and log *n* processors.

Now, for the number of processors. The straightforward method of using *n* log *n* processors, interconnected into a somewhat trimmed butterfly network with *n* rows and log *n* columns [24], as in the top-left panel of Fig. 3, can be used as a basis for deriving reduced-cost implementations. Projecting the said graph in the horizontal direction allows us to use a single processor to do the jobs of all processors in the same row, reducing the processor count by a factor of log *n*, without a significant increase in computation delay, given that the columns of a butterfly network do not work at the same time. An extreme reduction in hardware is achieved if the butterfly network is projected in the vertical direction, leading to log *n* processors, each of which does the job of *n* processors appearing in the same column. In this case, computation time does increase substantially. Quite a few other intermediate realizations, ranging in hardware complexity between the just-mentioned extremes of *n* log *n* and log *n*, are possible.

Given the central importance of DFT in a wide array of applications [27] and its special structure admitting the application of many design tricks and optimizations, diverse hardware implementations, for different values of *n* and with different optimality criteria, have been proposed over the years, adding many new design points to those in Bergland's early survey [28]. It is virtually impossible to cite all contributions in this domain (a Google Scholar search for "FFT hardware implementation" returns 173,000 hits, with many additional hits reported if we replace FFT with DFT in the search phrase).

A key notion connects the discussions of FFT (Section III) and multiplication (Section II). Some of the asymptotically fastest multiplication algorithms for large numbers are FFT-based [29]. Besides DFT, several other transforms have been proposed, analyzed, and implemented over the years. Notable examples with high-speed implementations include Walsh-Hadamard, generalized, discrete orthogonal, arithmetic, and number-theoretic transforms [30].

V. Conclusion

In this paper, we have scratched the surface and presented just a few examples of the many interactions and cross-fertilizations that occur between designers of arithmetic algorithms and their hardware embodiments on one side and the parallel computation community on the other. It is hoped that this study can be extended to produce a catalog of important areas of overlap and technology transfer. A possible methodology for discovering pertinent examples is to examine cross-citations between the main conferences and journals where the two communities publish their work.

Th rise of interest in GPUs as essential components of large supercomputers, in order to achieve high computational power at low hardware and energy costs, has put computer arithmetic in the forefront of efforts to achieve the next performance milestone, that is, exascale machines capable of executing a peak of 10^{18} floating-point operations per second. Such systems are generally thought to be 3-4 years away. Modern GPUs will soon achieve a performance on the order of 100 gigaflops per watt, making an exascale system possible with only 10 MW of power for its computing nodes (memory and communication power must be accounted for separately). It is thus not surprising that computers at the leading edge of the Top-500 list [31] of world's most powerful machines are predominantly GPU-based in their designs.

Because such supercomputers are designed and built by several different computer vendors and research organizations, and they use diverse hardware components in their processing nodes, program portability and results reproducibility among them hinges upon adherence to standard representation formats and the attendant consistency of representation and computation errors. The IEEE 754 binary floating-point standard [32], first issued in 1985, was revised in 2008, and is now undergoing final deliberations for its 2018 version. Design of algorithms and computational frameworks to reduce floating-point errors is being actively pursued by both the computer-arithmetic and parallel-computing communities, as are scheme for computing with little or no error or with the provision of guaranteed error bounds.

References

[1] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, Vol. 27, No. 4, pp. 831-838, 1980.

[2] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Computers*, Vol. 31, No. 3, pp. 260-264, 1982.

[3] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrences," *IEEE Trans. Computers*, Vol. 22, No. 8, pp. 786-793, 1973.

[4] B. Sugla and D. A. Carlson, "Extreme area-time tradeoffs in VLSI," *IEEE Trans. VLSI Systems*, Vol. 39, No. 2, pp. 251-257, 1990.

[5] T. Han and D. A. Carlson, "Fast area-efficient adders," *Proc. 8th IEEE Symp. Computer Arithmetic*, Como, Italy, pp. 49-56, 1987.

[6] S. Knowles, "A family of adders," *Proc. 14th IEEE Symp. Computer Arithmetic*, pp. 277-281, Adelaide, Australia, 1999.

[7] A. Beaumont-Smith and C. C. Lim, "Parallel prefix adder design," *Proc. 15th IEEE Symp. Computer Arithmetic*, Vail, CO, pp. 218-225, 2001.

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

[9] D. Harris, “A taxonomy of parallel prefix networks,” *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, Pacific Grove, CA, Vol. 2, pp. 2213-2217, 2003.

[10] R. P. Brent and H. T. Kung, “The area-time complexity of binary multiplication,” *J. ACM*, Vol. 28, No. 3, pp. 521-534, 1981.

[11] A. Karatsuba and Y. Ofman, “Multiplication of many-digital numbers by automatic computers,” *Physics-Doklady*, Vol. 7, pp. 595-596, 1963

[12] N. Nedjah and L. de Macedo Mourelle, “A review of modular multiplication methods and respective hardware implementation,” *Informatica*, Vol. 30, No. 1, pp. 111-129, 2006.

[13] P. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, Vol. 44, No. 170, pp. 519–521, 1985.

[14] A. L. Toom, “The complexity of a scheme of functional elements realizing the multiplication of integers,” *Soviet Mathematics*, Vol. 4, No. 3, pp. 714-716, 1963.

[15] S. A. Cook, “Positive results,” Chapter 3 in PhD thesis entitled “On the minimum computation time of functions,” Harvard University, 1966.

[16] A. Schonhage and V. Strassen, “Schnelle multiplikation großer zahlen,” *Computing*, Vol. 7, Nos. 3-4, pp. 281-292, 1971.

[17] M. Furer, “Faster integer multiplication,” *SIAM J. Computation*, Vol. 39, No. 3, pp. 979-1005, 2009.

[18] T. Kortekaas, “Multiplying large numbers and the Schonhage-Strassen algorithm,” online document, accessed on May 31, 2018: https://tonjanee.home.xs4all.nl/SSAdescription.pdf

[19] V. Strassen, “Gaussian elimination is not optimal,” *Numerische Mathematik*, Vol. 13, pp. 354-356, 1969.

[20] J. F. Palmer, “The Intel 8087 numeric data processor,” *Proc. AFIPS National Computer Conf.*, Anaheim, CA, May 1980, pp. 887-893.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: a unified graphics and computing architecture,” *IEEE Micro*, Vol. 26, No. 2, pp. 39-55, 2008.

[22] J. Nickolls and W. Dally, “The GPU computing era,” *IEEE Micro*, Vol. 30, No. 2, pp. 56-69, 2010.

[23] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, “GPUs and the future of parallel computing, *IEEE Micro*, Vol. 31, No. 5, pp. 7-17, 2011.

[24] B. Parhami, *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum, 1999.

[25] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Mathematics of Computation*, Vol. 19, No. 90, pp. 297-301, 1965.

[26] J. W. Cooley, P. A. W. Lewis, and P. D. Welch, “Historical notes on the fast Fourier transform,” *Proceedings of the IEEE*, Vol. 55, No. 10, pp. 1675-1677, 1967.

[27] E. O. Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, 1988.

[28] G. Bergland, “Fast Fourier transform hardware implementations—a survey,” *IEEE Trans. Audio and Electroacoustics*, Vol. 17, No. 2, pp. 109-119, 1969.

[29] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, pp. 270-274.

[30] D. F. Elliott and K. R. Rao, *Fast Transforms: Algorithms, Analyses, Applications*, Elsevier, 1983.

[31] List of World’s Top 500 Supercomputers, updated twice per year: http://www.top500.org

[32] *IEEE Standard for Binary Floating-Point Arithmetic P754*, Std 754-2008, approved 12 June, IEEE Press, 2008.