Fast Exponential Computation on SIMD Architectures

A. Cristiano I. Malossi, Yves Ineichen, Costas Bekas, and Alessandro Curioni
IBM Research - Zurich, Switzerland
Cognitive Computing & Computational Sciences Department
{acm,yin,bek,cur}@zurich.ibm.com

***Abstract*—State of the art implementations of the exponential function rely on interpolation tables, Taylor expansions or IEEE manipulations containing a small fraction of integer operations. Unfortunately, none of these methods is able to maximize the profit of vectorization and at the same time, provide sufficient accuracy. Indeed, many applications such as solving PDEs, simulations of neuronal networks, Fourier transforms and many more involved a large quantity of exponentials that have to be computed efficiently.**

In this paper we device and demonstrate the usefulness of a novel formulation to compute the exponential employing only floating point operations, with a flexible accuracy ranging from a few digits up to the full machine precision. Using the presented algorithm we can compute exponentials of large vectors, in any application setting, maximizing the performance gains of the vectorization units available to modern processors. This immediately results in a speedup for all applications.

***Keywords*-Exponential Function; SIMD Architectures; Approximate Computing; Neural Network; Fourier Transform; Wave Equations**

I. Introduction

The natural exponential function $f(x)$, hereafter simply referred to as *exponential function*, is defined as

$$f(x) = e^x \equiv \exp(x), \tag{1}$$

where x is the exponent and e the Euler's number, i.e., the base of the exponential function.

Many problems such as neuronal network simulations [1], Fourier transforms, wave equations as well as lumped models for cardiovascular problems [2], [3], radioactive decay, and population growth models [4], imply repeated evaluation of the exponential function. In particular, in all these applications, the time required to evaluate (1) might represent the main computational bottleneck, limiting the over-all time-to-solution for the problem.

In this paper we present a novel fast and accurate method to evaluate the exponential function with an arbitrary degree of accuracy. Our method is based on IEEE manipulation implementations (see Section II) coupled with polynomial interpolation. In order to harvest the full power of H/W features of modern processors, such as short vector instruction units, we reengineered the algorithm. Single instruction multiple data (SIMD) units apply the same operations on a vector like register in a single cycle. For example on a vector register of 4 double-precision values, a SIMD unit can execute an operation, i.e., scaling by the values by a constant, with one single instruction on all 4 values simultaneously.

In the following we target double-precision architectures, i.e., x in (1) is defined in the approximate interval (-746; 710) to respect the IEEE limits. However, our algorithm can be adapted without major modifications to arbitrary and variable precision arithmetic architectures (single-precision, quadruple-precision, GPUs, FPGAs, etc.). The main advantages of our approach are:

1) In case of streams of exponentials, our algorithm enables the use of only SIMD instructions, while existing algorithms still require a number of non-vectorizable operations.
2) The formulation empowers the user to specify the required accuracy of computing the exponential. This is very important because the required accuracy depends on the application and in many cases an approximation to a couple of digits suffices.
3) Ultimately this allows us to maximize performance, and at the same time, it minimizes energy consumption.

In Section II we discuss existing techniques that are used to compute the exponential (1) before we move on to our new approach in Section III. Finally we benchmark our approach in Section IV and conclude with a summary of features and remarks in Section V.

II. State-of-the-art methods

Several methods exist in the literature to compute exactly or approximately the exponential function. In the following we list the most widely used approaches, indicating major pros and cons.

A. Power series/Taylor expansions

By using power series or Taylor expansions, equation (1) can be re-written as

$$e^x = \sum_{k=1}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

The main advantage of this strategy is that the accuracy of the exponential function can be controlled by varying k. In the limit (k towards infinity) the sum converges to the exact value of the exponential function. However, this approach has a very slow convergence rate for increasing values of k, unless x is close to zero[1]. Note that even with Horner's method it requires too many floating-point multiply-add operations to obtain the desired accuracy. This is a severe disadvantage and limits the efficiency of an implementation.

[1] If the range of value of x is limited and known a priori, the rule $e^x = e^z \cdot e^{x-z}$ can be applied, being z a constant.

B. Look-up tables

The exponential function can be analytically converted in a base-two expression and subsequently decomposed in its integer x_i and fractional x_f parts, i.e.,

$$e^x = 2^{x \log_2(e)} = 2^{x_i + x_f}, \tag{2}$$

where we remark that $x_f \in [0, 1)$. Different algorithms based on one or more look-up tables can be used to evaluate 2^{x_i} and 2^{x_f} separately. Examples of this approach are [5]–[7], as well as the patents [8], [9]. Look-up tables are in general faster than Taylor expansions, as they are based on linear interpolation, however they do not fully exploit floating-point arithmetics and cannot be implemented using only SIMD instructions, due to the presence of conditional statements.

C. IEEE-745 manipulation

A well know trick to get a fast approximation of the exponential is to manipulate the components of a standard IEEE-745 floating-point representation [10]. This technique has been first published by Schraudolph in [11], and later extended in [12]. More in detail, IEEE-745 floating-point numbers are represented in the form

$$(-1)^s(1 + m)2^{x - x_0}, \tag{3}$$

where s is the sign bit, m the mantissa (i.e., a binary fraction in the range $[0, 1)$), and x the exponent, shifted by a constant bias x_0. For double-precision floating-point numbers, the IEEE-745 standard specifies that x_0 is equal to 1023, and that the length of the exponent x and the mantissa m are 11 and 52 bits, respectively, for a total of 64 bits (8 bytes), including the sign bit. Schraudolph noted the similarity between (2) and (3) and developed a technique to take advantage of the 2^x power implied in the standard floating-point representation. To do that, he shifts the exponent by the number of bits required to obtain the integer part of the exponential (i.e., 2^{x_i}), and then he approximates the fractional part 2^{x_f} with $(1 + m)$. This approach can be summarized in three steps: *(i)* store the manipulated input exponent x in an 32 bit integer variable `int i` as

$$\texttt{int i} = A \cdot x + B - C, \tag{4}$$

with $A = S/\ln(2)$, $B = S \cdot 1023$, $C = 60801$, being $S = 2^{20}$ the shift factor, *(ii)* concatenate the 32 bit integer `int i` with another 32 bit integer `int j` to form a 64 bit line, and *(iii)* interpret the 64 bit line as a double-precision floating-point number, which actually coincides with the approximation of e^x in (1). See [11, Figure 1] for more details.

The approximation developed by Schraudolph relies on simple arithmetics, which consists of a single floating-point multiply-add in (4), where A, B, and C are precomputed constants. However, the accuracy of the approximation is very low, i.e., only a single-digit, even though the C constant has been computed to minimize the RMS relative error.

III. Methodology

Even though several exponential function algorithms exist in the literature, none of them is able to profit from the SIMD capabilities of modern architectures and, at the same time, provide sufficient accuracy. We propose a new algorithm to accurately evaluate the exponential function, while extensively using vector instructions and thus attain an optimal hardware utilization.

The main idea of our methodology is to combine the manipulation of the standard IEEE-745 floating-point representation, as proposed by Schraudolph in [11], with a polynomial interpolation of the fractional part 2^{x_f}. We will show that the resulting algorithm can be written in a very compact form relying on only SIMD instructions. This enables a fast and energy-aware implementation of the exponential function, well suited for state-of-the-art architectures and supercomputers, such as the IBM® POWER7, POWER8 and IBM® BG/Q.

A. Derivation

Our starting point is equation (4), which is derived in [11]. To recover accuracy, without compromising the performance, we make the following modifications:

1) We use a single 64 bit `long int` instead of two contiguous 32 bit `int`: this simplify the conversion to `double`, and allows to profit from all the 52 bits of the double-precision mantissa.
2) As a consequence from point 1), all operations exclusively use double-precision, setting the shift factor S equal to 2^{52}.
3) We set $C = 0$, as this constant is useless with our subsequent modifications.
4) from (2), we write the following equality

$$e^x = 2^{x_i} \cdot 2^{x_f} \approx 2^{x_i} \cdot (1 + m - \mathcal{K}), \tag{5}$$

where $\mathcal{K}$ is a correction function (defined below) to improve the accuracy of the approximation.
5) Solving equation (5) for $\mathcal{K}$ we get

$$\mathcal{K} = 1 + m - 2^{x_f}.$$

Moreover, we note that $m \equiv x_f$, so that we can define the correction function as

$$\mathcal{K}(x_f) = 1 + x_f - 2^{x_f},$$

where we remark that x_f is defined in the limited domain $[0, 1)$.
6) We model the correction function $\mathcal{K}(x_f)$ with a polynomial $\mathcal{K}_n(x_f)$ in the form

$$\mathcal{K}_n(x_f) = a \cdot x_f^n + b \cdot x_f^{n-1} + c \cdot x_f^{n-2} + \ldots,$$

where $n \in [1, 10]$ denotes the order of the polynomial interpolation. The coefficients $\{a, b, c, \ldots\}$ are pre-computed according to the chosen interpolation.
7) Last, we plug our polynomial in the original expression, i.e.,

$$\texttt{long int i} = A\left(x - \ln(2) \cdot \mathcal{K}_n(x_f)\right) + B. \tag{6}$$

Note that the expression in (6) is similar to the original one in (4). However, as we show in the following, the new expression can match the accuracy of the standard reference implementation, i.e., by computing it to 16 digits.

Algorithm I **Input:** x and n; **Output:** $f(x) \approx e^x$

1: $x = x \cdot \log_2(e)$
2: $x_f = x - \texttt{floor}(x)$
3: $x = x - \mathcal{K}_n(x_f)$, with $\mathcal{K}_n(x_f) = a \cdot x_f^n + b \cdot x_f^{n-1} + c \cdot x_f^{n-2} + \ldots$
4: Compute the `long int i` as: $2^{52} \cdot x + \mathrm{B}$
5: Read the `long int i` as a `double` and return the value as the approximated exponential e^x

Kernel I: Machine independent C++ implementation of the scalar version of the exponential function with a polynomial of degree 5. The ● green line leads to the accuracy in Figure 1b black/orange solid lines, while the ● yellow line extend the accuracy as in Figure 1b black/orange dashed lines. All the variables starting with the prefix "`COEFF_`" are precomputed constants stored as macros.

```
1  double fast_exp_P5 (double x)
2  {
3      x *= COEFF_LOG2E;
4
5      const double fractional_part = x - floor(x);
6
7      x -= ( ( ( (           COEFF_P5_A
8         * fractional_part + COEFF_P5_B )
9         * fractional_part + COEFF_P5_C )
10        * fractional_part + COEFF_P5_D )
11        * fractional_part + COEFF_P5_E )
12        * fractional_part + COEFF_P5_F;
13
14     long int castedInteger = (long int) (COEFF_A * x + COEFF_B);
15     long int castedInteger = (long int) (COEFF_A * static_cast<long double> (x) +
16                                          COEFF_B);
17     return reinterpret_cast< double& > ( castedInteger );
18 }
```

B. Algorithm

From the practical point of view, an efficient algorithmic workflow of the methodology described in the previous section is provided in Algorithm I. In the following we provide more details on the five steps in the algorithm:

Step 1: The result of this line can be stored back in x. Note that in [11] the author divides x by $\ln(2)$, which leads to the same result, but with an additional cost for a floating-point division operation.

Step 2: The use of `floor` allows a correct approximation also for negative exponents.

Step 3: All the required coefficients are pre-computed;

Step 4: For the machine precision accuracy, two static cast might be required in C++ (see later Kernel I).

Step 5: This step in C++ can be performed as a reinterpret cast.

C. Implementation

The implementation of Algorithm I can be tuned according to the target architecture to maximize performance.

In Kernel I our algorithm is implemented in standard C++. This scalar version can be used on any architecture, since does not make use of architecture-dependent SIMD vector instructions. The example in Kernel I make use of a polynomial of degree 5; the implementation for other degrees is straightforward. Note that the value of the coefficients of the polynomial, identified by the prefix "`COEFF_P5`" are independent from the implementation; in other words, the user can decide the preferred polynomial for his implementation without changing the code.

Kernel II: IBM C++ implementation of the SIMD vector version of the exponential function with a polynomial of degree 5 on IBM BG/Q, where the input vector is assumed to be divisible by 4. The ● green and the ● yellow lines correspond to two possible alternative implementations; the ● blue line is used for OpenMP parallelism (only together with ● yellow lines); all other lines are present in both versions. All the variables starting with the prefix "`COEFF_`" are precomputed constants stored as macros.

```
1  vector double fast_expd2_P5 ( vector double x_vd )
2          void fast_vexp_P5  (double * y, double * x, const int& vsize)
3  {
4      const vector4double a_vd = vec_splats(COEFF_P5_A), d_vd = vec_splats(COEFF_P5_D);
5      const vector4double b_vd = vec_splats(COEFF_P5_B), e_vd = vec_splats(COEFF_P5_E);
6      const vector4double c_vd = vec_splats(COEFF_P5_C), f_vd = vec_splats(COEFF_P5_F);
7      const vector4double coeffA_vd  = vec_splats(COEFF_A),
8      const vector4double coeffB_vd  = vec_splats(COEFF_B);
9      const vector4double M_LOG2E_vd = vec_splats(COEFF_LOG2E);
10
11     #pragma omp parallel for schedule(static)
12     for (int j = 0; j < vsize ; j+=4) {
13         x_vd = vec_mul(x_vd, M_LOG2E_vd);
14         vector4double x_vd = vec_mul(vec_ld(0, &x[j]), M_LOG2E_vd);
15         vector4double fractional_part_vd = vec_sub(x_vd, vec_floor(x_vd));
16         return vec_ctidu(vec_madd(vec_sub(x_vd,vec_madd(fractional_part_vd,
17         vec_st(vec_ctidu(vec_madd(vec_sub(x_vd,vec_madd(fractional_part_vd,
18                                                 vec_madd(fractional_part_vd,
19                                                 vec_madd(fractional_part_vd,
20                                                 vec_madd(fractional_part_vd,
21                                                 vec_madd(fractional_part_vd,
22                                     a_vd,b_vd),c_vd),d_vd),e_vd),f_vd))),
23                                     coeffA_vd,coeffB_vd));
24                                     coeffA_vd,coeffB_vd)), 0, &y[j]);
25     }
26 }
```

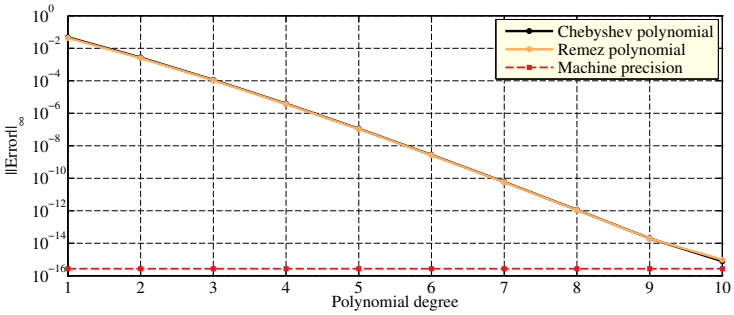
In Kernel II we provide an implementation of the exponential function algorithm tuned for the IBM® Blue Gene/Q (BG/Q). This implementation make extensive use of SIMD vector instructions. Similar implementations for IBM® POWER7 and POWER8 are straightforward, as well as those for other architectures, e.g., Intel® Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX). All the instructions, including the loads, the store, the floating-point multiply-add, the `floor`, and the `long int` to `double` conversion are SIMD vectorized. On IBM® BG/Q this allows to process four exponent at the same time, i.e., during the same CPU cycles. It can be also noted that in this implementation, Steps 3, 4, and 5 in Algorithm I have been joined in a single code line to minimizes the number of temporary variables, even though the compiler can always decide to reintroduce them.

We remark that these implementations are presented as reference in a high level fashion. In particular, assembly code is suppressed. Although modern compilers do a good job in optimizing the code, an assembler version allows a precise accountability of used instructions.

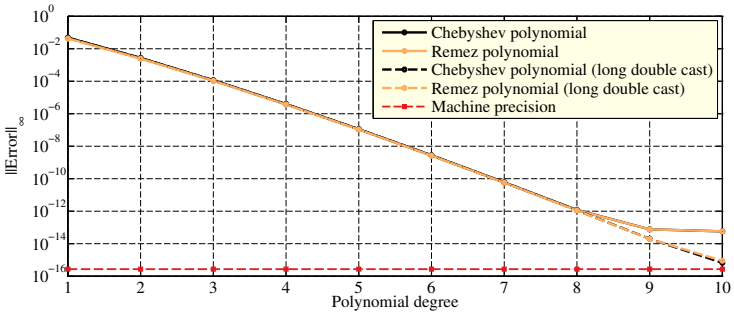
D. Accuracy

One of the main features of the presented methodology is that the accuracy of the approximated exponential function can be tuned by the user, who has the flexibility to decide the degree n (and the type) of polynomial that models the correction function in Step 3 of Algorithm I. A library with a range of implementations for $n \in [1, 10]$ can be available at compile time to provide this functionality in a general way.

In Figure 1 we show the relation between the degree n and the norm-inf of the error for two polynomials: Chebyshev of



(a)



(b)

Figure 1: Chebyshev polynomial (first kind) vs. Remez polynomial. (a) Fractional part approximation, i.e., $\|\mathcal{K}_n(x_f) - \mathcal{K}(x_f)\|$. (b) Exponential approximation with $x \in [0, \ln(2)]$. Machine precision accuracy is recovered by using a cast to `long double` for degrees 9 and 10.

the first kind, and the Remez. The latter one minimizes the infinity norm of the error. Among other information, the figure shows how a polynomial of degree 4 guarantee an accuracy of at least 6 digits, which is already more than enough for most of the scientific computing applications. Moreover, a polynomial of degree 8 reaches an accuracy of at least 12 digits, and for the most demanding scenarios the machine precision can be reached with a polynomial of degree 10 and an additional cast to `long double` (see yellow line in Kernel I).

IV. Results

To benchmark the potential of our algorithm we tested it on two architectures, namely the IBM® Blue Gene/Q supercomputer [13] and the IBM® Power 755 (P755) server [14]. Both architectures have 4 hardware threads per core and are equipped with multiple cache levels. As a reference, we compare only against the IBM® MASS implementation, since we verified that IBM® MASS performance is always as good or better than the `std::exp` performance.

In Figures 2 and 3 we show the time and percentual reduction with respect to the IBM® MASS library for both architectures. In particular, Figure 2 targets L1 cache, with a small vector of 2^7 exponents, while Figure 3 targets DDR memory, with a long vector of 2^{22} exponents. For both cases and architectures we notice that our approach (labeled as "Polynomial") is better than the corresponding implementation provided by the IBM® MASS library. Note that IBM® MASS library offers three implementations: a scalar version (labeled as "Mass" and compared with Kernel I), a SIMD vector version with standard interface (labeled as "Mass V." and compared with Kernel II, yellow lines), and a SIMD vector version with a machine-dependent interface (labeled as "Mass V4D" on IBM® BG/Q and "Mass V2D" on IBM® P755, and compared with Kernel II, green lines). In all the presented cases, our algorithm perform better than the reference IBM® MASS implementation. Moreover, we can observe how performances can be further increased by reducing the degree n of the interpolating polynomial.

Finally, as shown in Kernel II, the algorithm can be extended easily to incorporate threads (see blue line) and split up the work even further for large vectors. As shown in Figure 4, the OpenMP implementation with one thread, pays a small overhead with respect to the non-threaded version. However this effect disappears when using two or more threads, leading to increased performance, expecially for large vectors.

V. Conclusions

We presented an algorithm to compute the exponential of a value (or of a list of values) with an augmented vectorization friendliness that allows us to speed up the computation by a large factor. The main advantages or our approach are:

1) The user has control over accuracy of the exponential, by selecting an appropriate degree of the polynomial approximating the fractional part 2^{x_f}. This can have high impact on the final application, as the user knows best which accuracy is reccommended to solve the problem at hand.
2) The algorithm is designed to enable a pure SIMD implementation. This has been demonstrated on IBM® BG/Q (with QPX vectorization) and on IBM POWER7 (with VSX vectorization). Similar implementations on other architectures, e.g., Intel® are straighforward.
3) The algorithm provides a huge reduction in the time-to-solution. This has been quantified in up to 96 % on IBM® BG/Q and POWER7 architectures. Similar performances are expected on other architectures (e.g., IBM® POWER8 and Intel®).
4) Additionally, the scalar versions, where only one exponential is evaluated at each call and thus SIMD instructions are not applicable, provides also a sensible reduction in time-to solution (between 10 % and 50 % on IBM® BG/Q, and between 75 % and 90 % on IBM® POWER7).
5) OpenMP helps to further improve time-to-solution in the case of large vectors (e.g., vectors that do not fit the lower caches levels.
6) In case of a vector size that is not divisible by the SIMD factor (i.e., 4 on BGQ and 2 on POWER7) the performances does not diminish significantly.
7) Also energy-to-solution is reduced, mainly as a consequence of the diminished computational time.

The last two points have been verified on both architectures, even if results are not presented here.

Acknowledgments

The project Exa2Green (under grant agreement n°318793) acknowledges the financial support of the Future and Emerging

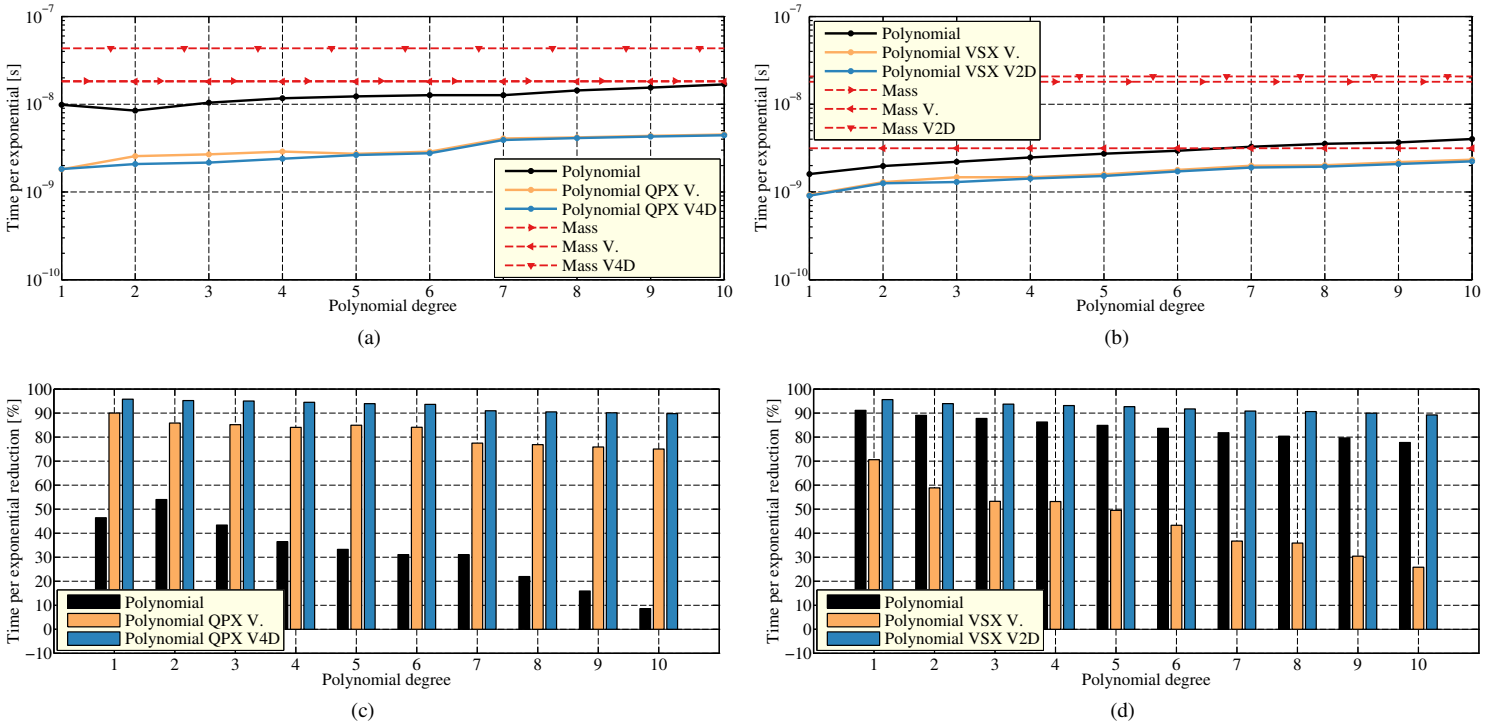


Figure 2: Time per exponential (top row) and corresponding percentual reduction with respect to the IBM® MASS library (bottom row) for a vector of size 2^7. LEFT: IBM® BG/Q, RIGHT: IBM® P755.

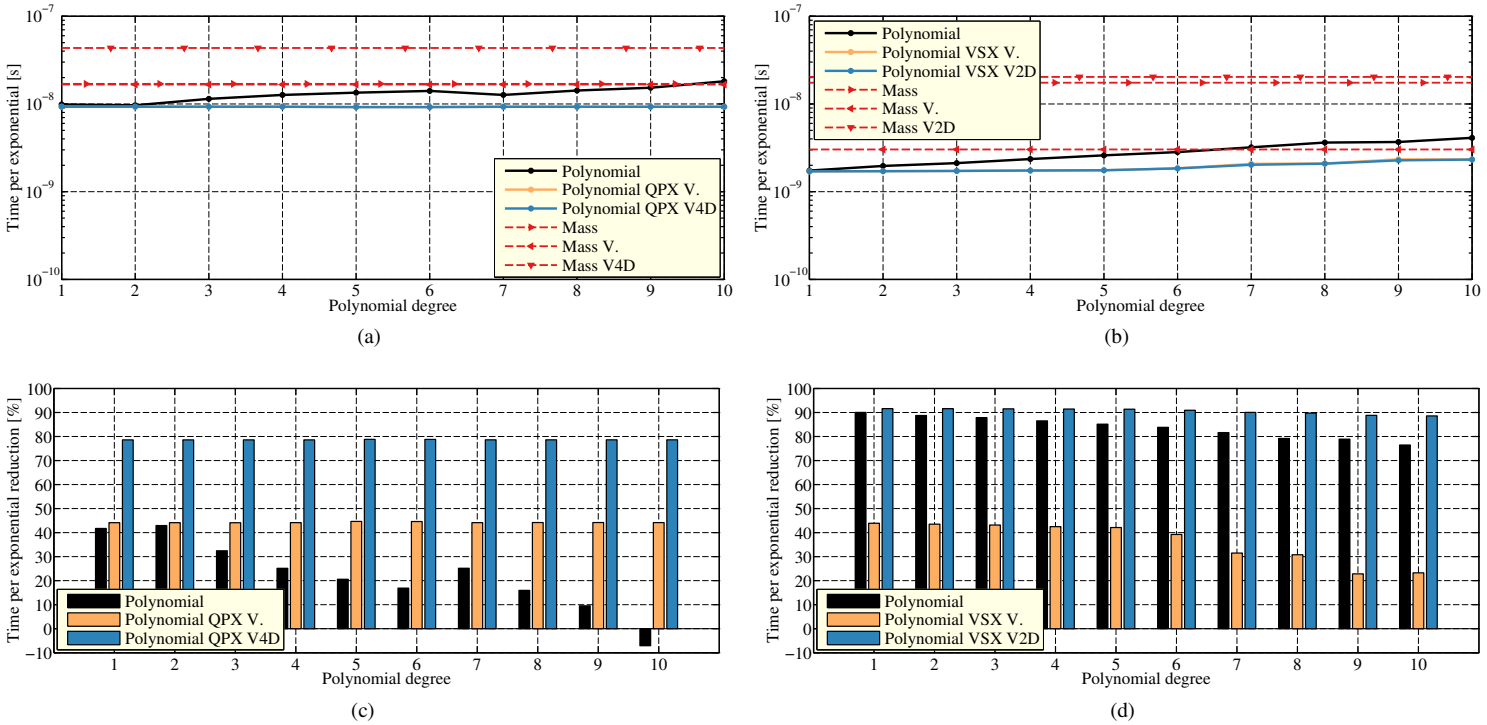


Figure 3: Time per exponential (top row) and corresponding percentual reduction with respect to the IBM® MASS library (bottom row) for a vector of size 2^{22}. LEFT: IBM® BG/Q, RIGHT: IBM® P755.

Technologies (FET) programme within the ICT theme of the Seventh Framework Programme for Research (FP7/2007-2013) of

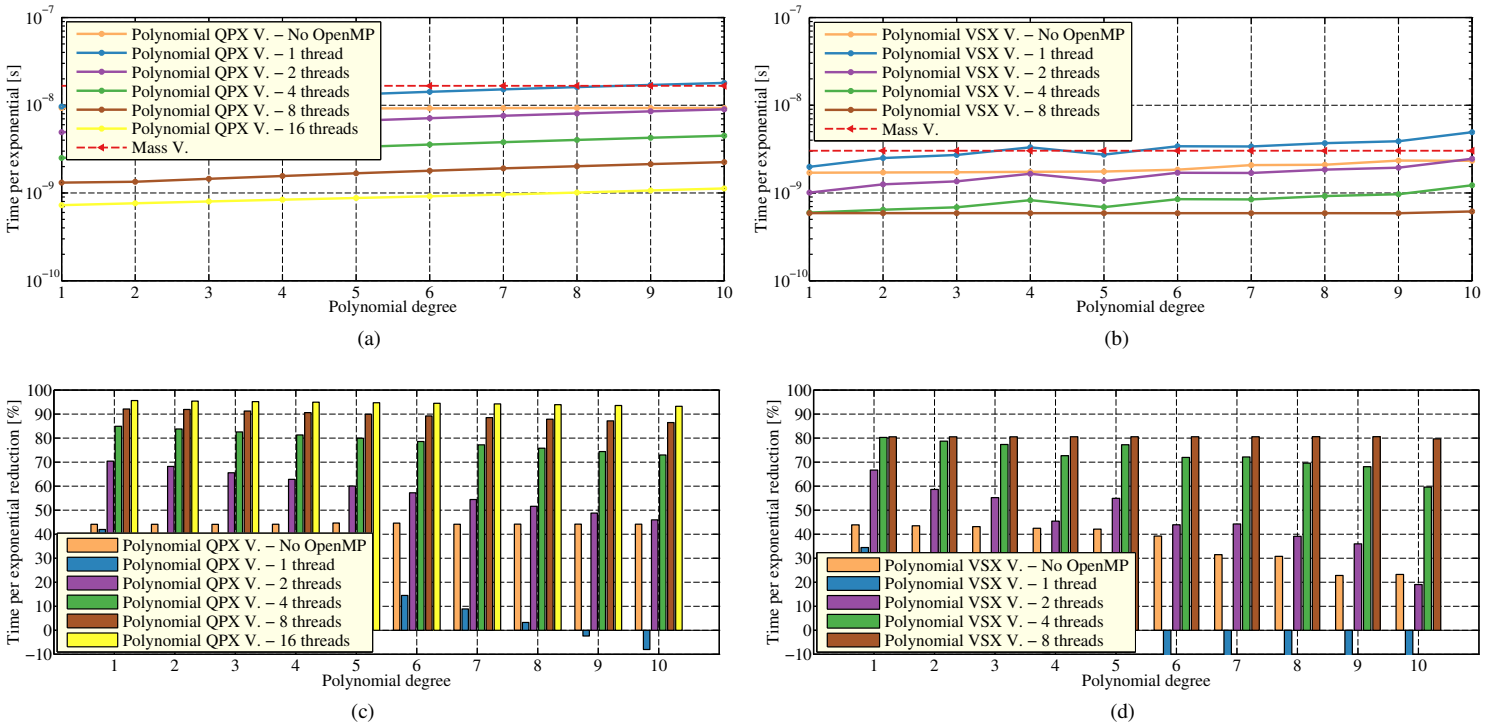


Figure 4: Time per exponential (top row) and corresponding percentual reduction with respect to the IBM® MASS library (bottom row) with and without OpenMP directives (always one thread per core) for a vector of size 2^{22}. LEFT: IBM® BG/Q, RIGHT: IBM® P755.

the European Commission.

REFERENCES

[1] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. Bower, M. Diesmann, A. Morrison, P. Goodman, F. C. Harris, and et al., "Simulation of networks of spiking neurons: A review of tools and strategies," *Journal of Computational Neuroscience*, vol. 23, no. 3, pp. 349–398, 2007.

[2] P. Reymond, F. Merenda, F. Perren, D. Rüfenacht, and N. Stergiopulos, "Validation of a one-dimensional model of the systemic arterial tree." *American journal of physiology. Heart and circulatory physiology*, vol. 297, no. 1, pp. H208–22, 2009.

[3] A. C. I. Malossi, P. J. Blanco, and S. Deparis, "A two-level time step technique for the partitioned solution of one-dimensional arterial networks," *Computer Methods in Applied Mechanics and Engineering*, vol. 237-240, pp. 212–226, 2012.

[4] P. Auger and J.-C. Poggiale, "Emergence of population growth models: fast migration and slow growth," *Journal of Theoretical Biology*, vol. 182, no. 2, pp. 99–108, 1996.

[5] C. Baumann, "A simple and fast look-up table method to compute the exp(x) and ln(x) functions," http://www.convict.lu/Jeunes/ultimate_stuff/exp_ln_2.htm, July 2004.

[6] H. Ainsworth, "Fast pow() with adjustable accuracy," http://www.hxa.name/articles/content/fast-pow-adjustable_hxa7241_2007.html, November 2007.

[7] X. Yan, T. Tang, Y. Deng, J. Du, and X. Yang, "Evaluation of transcendental functions on imagine architecture," in *Parallel Processing, 2007. ICPP 2007. International Conference on*, Sept 2007, pp. 53–53.

[8] Z. Hussain, "Exponent processing systems and methods," Patent Grant US7 912 883, 3 22, 2011. [Online]. Available: http://www.google.com/patents/US7912883

[9] K. Azadet, J. G. Chen, S. Hijazi, and J. Williams, "Digital signal processor having instruction set with an exponential function using reduced look-up table," Patent Application US20 100 198 894, 08 5, 2010. [Online]. Available: http://www.google.com/patents/US20100198894

[10] IEEE Computer Society, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[11] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Comput.*, vol. 11, no. 4, pp. 853–862, 1999.

[12] G. C. Cawley, "On a fast, compact approximation of the exponential function," *Neural Comput.*, vol. 12, no. 9, pp. 2009–2012, 2000.

[13] J. Milano and P. Lembke, "IBM system Blue Gene solution: Blue Gene/Q hardware overview and installation planning," IBM, Tech. Rep. SG24-7872-01, May 2013.

[14] S. Vetter, G. Anselmi, B. Blanchard, Y. Cho, C. Hales, and M. Quezada, "IBM Power 750 and 755 (8233-E8B, 8236-E8C) technical overview and introduction," IBM, Tech. Rep. REDP-4638-00, April 2012.