Speeding Up Non-archimedean Numerical Computations Using AVX-512 SIMD Instructions

Lorenzo Fiaschi, Federico Rossi(✉), Marco Cococcioni, and Sergio Saponara

University of Pisa, Largo Lucio Lazzarino 1, 56122 Pisa, Italy
{lorenzo.fiaschi,federico.rossi}@ing.unipi.it,
{marco.cococcioni,sergio.saponara}@unipi.it

Abstract. This work presents the acceleration of a Bounded Algorithmic Number (BAN) library exploiting vector instructions in general-purpose processors. With the use of this encoding, it is possible to represent non-Archimedean numbers that are not only finite (like real numbers) but also infinite or infinitesimal. The tremendous growth in non-Archimedean numerical computations over the past 20 years and the resulting applications spurred this study's development. Enabling acceleration of BANs processing can significantly increase the throughput of non-Archimedean numerical computations, enlarging the spectrum of possible applications to industrial and real-time ones.

Keywords: Non-archimedean fields · Alpha theory · Bounded Algorithmic Number (BAN) · Single Instruction Multiple Data (SIMD) · AVX-512 instruction set

1 Introduction

Numerical non-Archimedean computations have been pioneered by Sergeyev and his Grossone Methodology [1], and allow for the use of infinitely large and infinitely small numbers in machines, other than finite ones as usual. From their advent, numerous applications benefited, especially in the domain of multi-objective optimisation, e.g., linear programming [2,3], quadratic programming [4], evolutionary algorithms [5], game theory [6], artificial intelligence [7,8], etc.

The reference framework of this study is the non-Archimedean model built upon the Alpha Theory [9], which introduces the set of Euclidean numbers and their associated numerical encoding called Bounded Algorithmic Number (BAN) format. The BAN encoding is a fixed length representation, guaranteeing that any operation involving two BANs outputs a result that occupies the same memory as the operands, exactly as happens with computations between 32-bit IEEE 754 floats, where the result of addition, subtraction, addition, and division is again a 32-bit float. However, since the Euclidean numbers form a superset of the real ones, their numerical representation is heavier than the one of floats, making the processing of BANs cumbersome. CPU vectorization can be exploited to optimise computations since BANs encoding can be implemented through multiple fixed-length coefficients, thus fitting them inside vector registers and efficiently computing operations in a single clock cycle.

Using vector instructions had already proved to be a good solution for handling non-native data types that do not have hardware acceleration [10,11]. In this paper, we present an optimisation of a C++ BAN library called *BANcpp* [12]. The goal is to optimise the library to leverage CPU vectorization when dealing with BAN coefficients. In particular, we focused on BAN numbers with eight 64-bit coefficients and 512-bit vector instruction sets (e.g., Intel AVX-512). We also present a benchmark application that consists of several iterations of a non-Archimedean optimisation problem. We evaluate the goodness of automatic vectorization as a baseline model and then we enhance the automatic vectorization whenever the compiler fails to automatically optimise the code.

The paper is organised as follows: (i) Sect. 2 briefly introduces Alpha Theory and the Euclidean numbers; (ii) Sect. 3 details the BAN format; (iii) Sect. 4 explains the choices made to implement the enhanced vectorization of the library; (iv) Sect. 5 shows the application benchmark and the results obtained in terms of timing performance and throughput.

2 Alpha Theory and The Euclidean Numbers

Alpha Theory reference set of non-Archimedean numbers is indicated with the symbol $\mathbb{E}$ and it is called the set of α-Euclidean numbers, or, in brief, just Euclidean numbers. The peculiar name of the theory comes from the definition of a reference infinite value within $\mathbb{E}$ and it is indicated by the symbol α. Then, any Euclidean number can be represented as a function of α, and only functions of α are numbers in $\mathbb{E}$, which guarantees that the Euclidean numbers and the mathematical operations among them behave according to their counterparts in $\mathbb{R}$, i.e., commutative operations continue to be commutative, differentiable functions are still differentiable, etc. For instance, the following are all Euclidean numbers:

$$\alpha^3, \qquad \frac{1}{\alpha}, \qquad \frac{1}{\alpha^2} - e^{\alpha}, \qquad -\frac{1}{2^{\alpha}}, \qquad -\ln\left(\frac{1}{\alpha}\right). \tag{1}$$

As opposed to Archimedean Mathematics the concepts of infinite and infinitesimal numbers are sharply defined rather than vague concepts.

Definition 1. Given $\xi \in \mathbb{E}$, then

- ξ is infinite $\iff \forall n \in \mathbb{N}, |\xi| > n$
- ξ is finite $\iff \exists n \in \mathbb{N}\ , \frac{1}{n} < |\xi| < n$
- ξ is infinitesimal $\iff \forall n \in \mathbb{N},\ |\xi| < \frac{1}{n}$.

Therefore, in (1) the first and fifth numbers are positive and infinite, the third is negative and infinite, the second is positive and infinitesimal, while the fourth is negative and infinitesimal. A more detailed presentation of Alpha Theory and the set $\mathbb{E}$ can be found in [13].

3 The BAN Format

The BAN encoding consists in a finite length representation for Euclidean numbers; however, as for IEEE 754 floating point numbers, it cannot represent the whole set $\mathbb{E}$ because it would require infinitely many binary possible representations, i.e., an infinite computer memory. Any Euclidean number compliant with the following representation can be represented as a BAN:

$$\xi = \sum_{i=1}^{L} r_i \alpha^{p-i},$$

where $L \in \mathbb{N}$ is the encoding length, $r_i \in \mathbb{R}$ and $p \in \mathbb{Z}$. Changing perspective, one can define a BAN as a Euclidean number that can be represented by a linear combination of L subsequent integer powers of α.

From the very first glimpse, one may notice that the BAN representation of a Euclidean number is very similar to the one of a polynomial, suggesting how cumbersome can be to execute algebraic operations between BANs. An example of addition and multiplication between BANs with $L = 3$ follows:

$$(3.2\alpha^2 - 0.5\alpha + 1.4) + (0.2\alpha + 1 - 1.5\alpha^{-1}) = 3.2\alpha^2 - 0.3\alpha + 2.4 - 1.5\alpha^{-1}$$

$$(3.2\alpha^2 - 0.5\alpha + 1.4) \times (0.2\alpha + 1 - 1.5\alpha^{-1}) = 0.64\alpha^3 + 3.1\alpha^2 - 5.02\alpha - 2.15 - 2.1\alpha^{-1}$$

Both computations output a result that is not a BAN since it requires more than three consecutive powers of α to be represented. Therefore, there is the need to approximate them considering only the first three highest powers of α, the most significant ones somehow, that is executing a truncation of the result. Below we report the numerical execution of the previous two operations, along with the division, realized by a software simulator of a BAN Processing Unit (BPU, whose hardware design has been recently proposed in [12]). The latter manipulates and outputs BANs in the normal form [13], i.e., in the standardized format which guarantees the uniqueness of the representation.

```
Operands: α^2(3.2 - 0.5η^1 + 1.4η^2) and α^1(0.2 + 1η^1 - 1.5η^2)

Sum: α^2(3.2 - 0.3η^1 - 2.4η^2)
Product: α^3(0.64 - 3.1η^1 - 5.02η^2)
Division: α^1(16 - 82.5η^1 + 539.5η^2)
```

4 Vectorization of BANcpp Library

We vectorized the *BANcpp* library by mixing two approaches: (i) leveraging the automatic vectorization offered by the compiler; (ii) enhancing vectorization manually whenever the automatic optimisation of the compiler fails. In particular, we needed to implement manual vectorization in the following cases:

- when a for-loop contains control-flow instructions on the BAN coefficients: due to possible branches and FPU exceptions the compiler refuses to insert vector instructions for these loops. The solution is to provide vectorization manually exploiting masked instructions based on comparisons. This is the case of comparison between BANs or checks on BAN values.
- when we have an outer and inner for-loop whose indexes are depending on each other. The solution is to implement vectorization manually leveraging the "geometry" of the problem. This is the case of multiplication between two BANs, which ends up in a one-dimensional convolution.

Listing 1. Manually vectorized 8 × 8 1-D convolution for AVX512 SIMD.

```
void convmul(const double* a, const double* b, double* dst) {
    __m512 va0 = _mm512_set1_pd(a[0]);
    ...
    __m512 va7 = _mm512_set1_pd(a[7]);

    __m512 vb07 = _mm512_loadu_pd(&b[0]);
    __m512 vb17 = _mm512_mul_pd(vb07,masked1);
    ...
    __m512 vb77 = _mm512_mul_pd(vb07,masked7);

    __m512 va0b = _mm512_mul_pd(va0,vb07);
    __m512 va1b = _mm512_slide(_mm512_mul_pd(va1,vb17),1);
    __m512 va2b = _mm512_slide(_mm512_mul_pd(va2,vb27),2);
    ...
    __m512 va7b = _mm512_slide(_mm512_mul_pd(va7,vb77),7);

    __m512 accr = _mm512_add_pd(va0b,va1b);
    accr = _mm512_add_pd(accr,va2b);
    ...
    accr = _mm512_add_pd(accr,va7b);
    _mm512_storeu_pd(&dst[0],accr);
}
```

5 Benchmark Application and Results

The problem used as a benchmark in this study is one of the first ever used for testing and showing the efficacy of non-Archimedean numerical computations, namely Kite [3]. It consists of a bi-objective lexicographic linear programming problem, i.e., an optimisation problem of two linear functions, ordered by strict priority, over a linearly defined domain. To solve the problem, we adopted a Simplex-like non-Archimedean algorithm [3], precisely tailored to this type of task. To make it more realistic, we wrapped the problem within the I-Big-M framework [14], which adds a third objective to generalize the optimisation to the case of unknown starting feasible basis.

We ran the benchmark application for 10^5 steps with both the auto-vectorized (namely, *baseline*) and the enhanced-auto-vectorized (namely *enhanced*) versions of the BAN library, collecting the time spent for each iteration. We smoothed the data by a 200-steps moving average window and computed a least square fit to plot the metric trends for the average time spent during an iteration and the average throughput (in terms of iterations per second). The benchmark was run on an Intel Xeon Gold 6238R processor running at 2.2 GHz with eight 64-bit BAN coefficients. Figure 1 shows the comparison between the two versions in terms of average time spent per iteration and overall throughput (iterations per second). Mean value and standard deviation of the two are reported in Table 1.

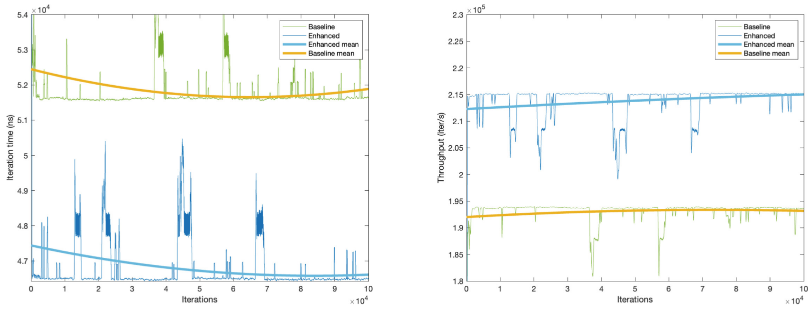

Fig. 1. Comparison between time spent for each iteration (**left**) and throughput (iterations per second, **right**) in the two different versions of the BAN library with the associated fitted curve.

Table 1. Mean value and standard deviation over 10^5 iterations of the benchmark application, 8 64-bit BAN Coefficient with AVX-512.

	Time (ns, $\times 10^4$)	Throughput (iter/s, $\times 10^5$)
Baseline (Non-vector)	9.44 ± 1.11	1.07 ± 0.87
Baseline (Vector)	5.18 ± 0.21	1.93 ± 0.04
Enhanced	4.68 ± 0.19	2.14 ± 0.05

6 Conclusions

In this work, we presented the acceleration of a C++ library for Bounded Algorithmic Numbers (BAN) exploiting vector instructions, testing it on a non-Archimedean optimisation benchmark. The results showed how manually enhancing the automatic vectorization produced by the compiler can improve the performance of such applications even without complete hardware support for BANs. We have found that the performance of compiler automatic vectorization is significantly inferior to that achieved by manually optimising which intrinsics to use (and in which order) and how to load the information (again, in which order and according to which scheme). This can be helpful for the community of compiler developers too since it means that there is room for improving the compilers for handling the specific use case tackled in this work.

Acknowledgments. Work partially supported by H2020 project TEXTAROSSA (grant no. 956831), https://textarossa.eu/, by the Italian Ministry of Education and Research (MUR), ForeLab project (Departments of Excellence), and by PNRR—M4C2—Investimento 1.3, Partenariato Esteso PE00000013—“FAIR—Future Artificial Intelligence Research”—Spoke 1 “Human-centered AI”.

References

1. Sergeyev YD (2017) Numerical infinities and infinitesimals: Methodology, applications, and repercussions on two Hilbert problems. EMS Surv Math Sci 4(2):219–320
2. De Cosmis S, De Leone R (2012) The use of grossone in mathematical programming and operations research. Appl Math Comput 218:8029–8038
3. Cococcioni M, Pappalardo M, Sergeyev YD (2018) Lexicographic multi-objective linear programming using grossone methodology: theory and algorithm. Appl Math Comput 318:298–311
4. Fiaschi L, Cococcioni M (2022) A non-archimedean interior point method and its application to the lexicographic multi-objective quadratic programming. Mathematics 10(23):4536
5. Lai L, Fiaschi L, Cococcioni M, Deb K (2021) Solving mixed pareto-lexicographic many-objective optimization problems: the case of priority levels. IEEE Trans Evol Comput 25:971–985
6. Cococcioni M, Fiaschi L, Lambertini L (2021) Non-Archimedean Zero Sum Games. J Comput Appl Math 393:113483
7. Astorino A, Fuduli A (2020) Spherical separation with infinitely far center. Soft Comput 24(23): 17 751–17 759
8. Cavoretto R, De Rossi A, Mukhametzhanov MS, Sergeyev YD (2021) On the search of the shape parameter in radial basis functions using univariate global optimization methods. J Global Optim 79(2):305–327
9. Benci V, Di Nasso M (2018) How to measure the infinite: mathematics with infinite and infinitesimal numbers. World Scientific, Singapore
10. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2020) Fast deep neural networks for image processing using posits and ARM scalable vector extension. J R-Time Image Process 17(3):759–771 Jun
11. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2021) Faster deep neural network image processing by using vectorized posit operations on a RISC-V processor. In: Kehtarnavaz N, Carlsohn MF (eds)Real-time image processing and deep learning 2021, vol 11736. International Society for Optics and Photonics. SPIE, p 1173604
12. Rossi F, Fiaschi L, Cococcioni M, Saponara S (2023) Design and FPGA synthesis of BAN processing unit for non-archimedean number crunching. In: Berta R, De Gloria A (eds) Applications in electronics pervading industry, environment and society. Springer Nature Switzerland, Cham, pp 320–325
13. Benci V, Cococcioni M, Fiaschi L (2022) Non-standard analysis revisited: an easy axiomatic presentation oriented towards numerical applications. Appl Math Comput 32(1):65–80
14. Cococcioni M, Fiaschi L (2021) The big-M method using the infinite numerical M. Optim Lett 15:2455–2468