Università di Pisa

Master Of Science in Computer Engineering

artificial intelligence and data engineering

BAN arithmetic using CPU vectorization SVE and NEC AURORA

Project members:
Massimo Valentino Caroti
Simone Landi

Contents

1 Introduction

This study is focusing on the non-Archimedean model based on Alpha Theory, introducing the concept of Euclidean numbers and their numerical encoding known as Bounded Algorithmic Numbers (BAN). The key advantage of the BAN format is its fixed-length representation. Unlike other numerical representations, it allows operations between BANs to occupy the same memory space as the operands, facilitating streamlined computations. However, it brings its challenges, primarily due to the heavier numerical representation of Euclidean numbers compared to traditional floating-point formats.

Optimizing computations with BANs is at the core of our study. To achieve this, we are utilizing CPU vectorization, implementing BANs through multiple fixed-length coefficients. This approach aims to maximize the utilization of vector registers, enabling efficient operations in a single clock cycle. The complexity lies in effectively translating the mathematical structures of Alpha Theory into a computational environment. Processing BANs can be cumbersome due to the intricacies of the Euclidean numbers and their encoding. This necessitates careful design and optimization, ensuring that the operations are both accurate and efficient.

Our work is concentrating on the bancpp, a specialized C++ library that has been developed to handle BAN coefficients. This library is integral to our research, and optimizing it is central to unlocking the full potential of BAN processing. The purpose of this work is to extend the translations made in the previous study, where the focus was on accelerating the BAN library using Intel's AVX512 architecture. The challenge is to extend bancpp for two specific computing systems, NEC Aurora and ARM SVE architectures. Each of these systems is offering unique possibilities, but they also come with their limitations and specific requirements.

We are examining BAN numbers with variable 64 and 32-bit coefficients tailored for both NEC Aurora and ARM SVE. Investigating different coefficient lengths is providing insights into performance variations and potential bottlenecks. The nuances associated with different bit- lengths and how they interact with the underlying architecture is a key aspect of our exploration. A benchmark application forms a critical part of our study. It consists of numerous iterations of a non-Archimedean optimization problem, allowing us to gauge the effectiveness of our methods. We are first evaluating the effectiveness of automatic vectorization, treating it as a baseline. This is a vital step, as it provides an initial understanding of how the compiler is handling the complexity of BAN computations. However, our study is revealing areas where the compiler's automatic optimization falls short, requiring us to manually intervene to enhance performance.

1.1 Alpha Theory

Non-Archimedean analysis is a mathematical branch that deals with fields lacking of the Archimedean property. We will focus on the Euclidean numbers and their axiomatization called Alpha Theory. This axiomatization is composed by just three axioms:

1. Existence: $\exists$ an ordered field $E \supset R$ whose numbers are called Euclidean numbers.
2. Numerosity of alpha: that introduces the infinite number alpha and states that it can be manipulated as any other real number using field properties such as commutativity, associativity, etc.
3. Alpha-limit: every real function f : R —> R can be extended to an E. function f*: E —>E

1.2 Algorithmic Numbers

Algorithmic numbers (ANs) constitute a subset of E, which can be better standard- ized in order to being easily used for computations on a machine.

A number $\xi \in E$ is called algorithmic if it can be represented as a finite sum of monosemia:

$$\xi = \sum_{k=1}^{l} r_k \alpha^{s_k}, \text{ with } r_k \in \mathbb{R},\ s_k \in \mathbb{Q},\ s_k > s_{k+1}.$$

An algorithmic number can always represented in a form, called normal form, such that:

$$\xi = \alpha^p P\left(\eta^{m_1}\right),$$

where:

$\eta := \alpha^{-1}$, $p \in Q$, $m \in N$ and $P(x)$ is a polynomial with real coefficients such that $P(0) \neq 0$.

The problems with the Algorithmic numbers as defined is that this subset is not closed with respect to inversion, meaning that when computing the inverse of an AN the result is not always an AN and this representation has a variable length coding. To solve both these issues, we need to use the truncation operation for algorithmic numbers, in particular for the polynomial P() of the normal form for ANs.

$$tr_n[P(x)] := \begin{cases} P(x) & \text{if } n \geq m, \\ p_0 x^{z_0} + \ldots + p_n x^{z_n} & \text{if } n < m. \end{cases}$$

This operation gives us a finite length approach called Bounded Algorithmic Numbers.

1.3 Bounded Algorithmic Numbers

The Bounded Algorithmic Numbers (BANs) are a particular subset of ANs suitable for computer computations.

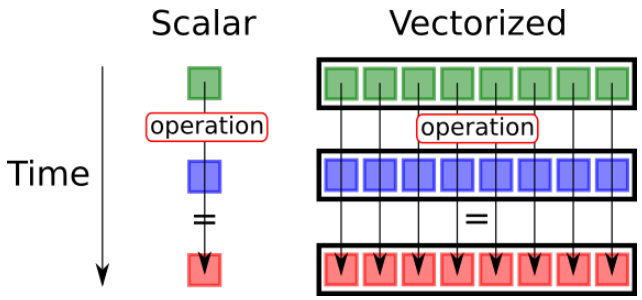
$$\xi = \sum_{i=1}^{L} r_i \alpha^{p-i},$$

Where $L \in N$ is the encoding length, $r_i \in R$, and $p \in Z$. 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.

In order to handle BAN operations in a well-defined manner, even with limited length representations, while respecting the standardized BAN format, the truncation process is applied to retain only the most significant terms in the result, thus maintaining numerical stability and consistency of the BAN representation during calculations.

2 Vectorization of bancpp library

The conventional method of programming follows a pattern known as SISD, or single instruction, single data stream. In this approach, if we want to carry out a particular operation on a vector, we go through each element one by one, applying the same operation to each. This method, although straightforward, is not the most efficient in terms of performance. Even compilers recognize this inefficiency and often attempt to make improvements.



Vectorization offers a solution to this problem by introducing a different paradigm called SIMD, or single instruction, multiple data stream. Rather than processing one element at a time, multiple elements of a vector are processed simultaneously, each by a dedicated functional unit performing the same instruction. This concurrent execution enables much more efficient computations, transforming the way we handle vector operations. Our approach has been to use the autovectorization offered by the compiler, going on to improve it manually whenever the compiler's automatic optimization fails.

We noticed that both compilers had similar difficulties in vectorizing for-loops containing control- flow instructions on BAN coefficients and to the presence of an outer and inner for-loop whose indices depend on each other. The techniques used to determine the parts of code not vectorized by the compilers and the functions implemented by us to have manual vectorization will be presented in subsequent architecture-specific chapters.

3 SVE

3.1 Overview

The Scalable Vector Extension (SVE) is an extension of the Armv8-A Architecture, available from Armv8.2-A. SVE is designed to improve integer and floating-point performance of Arm processors through enhanced vectorization compared to the Arm's existing Advanced SIMD instruction set.

SVE is a vector length agnostic architecture. Instead of mandating a particular vector length, it allows implementations to choose a length up to the architectural maximum of 2048 bits. The actual length of the vector is therefore not a compile-time constant and can in principle change at runtime. The SVE design guarantees that the same application can run on different implementations that support SVE, without the need to recompile the code. SVE improves the suitability of the architecture for High Performance Computing (HPC) and Machine Learning (ML) applications, which require very large quantities of data processing.

3.2 SVE Intrinsics

The SVE intrinsics are a set of functions and macros defined in the `arm_sve.h` header file. The definitions of these functions are known to the compiler, and there is a close correspondence between the Instruction Set Architecture (ISA) and the intrinsics. Using intrinsics gives you control over SVE code generation, without the need for assembly programming.

Arm recommends using intrinsics over assembly because code with intrinsics is easier to port and maintain.In order to use intrinsic functions, the following flag must be passed to the compiler: armclang++ -march=armv8.2-a+sve

3.2.1 Types

Several sizeless types are defined by the ACLE specification. Sizeless types are necessary because the length of the SVE vector registers are not known to the compiler. The arithmetic types have the following pattern:

```
sv<type>_t
```

For example, a scalable vector type of 16-bit signed integers is: `svint16_t`. There is also the type `svbool_t` which is used to represent predicates.

3.2.2 Function

The SVE intrinsics use the following general naming convention:

```
svbase [disambiguator][_type0] [_type1]...[_predication]
```

Where:

- base is the name of an instruction. For example,`svmla_n_f32_m()` corresponds to the MLA instruction.

- `_disambiguator` indicates any special behavior of the function. For example, `svld1_gather_s32_offset_u32()` indicates a gather load LD1.
- `_type0, _type1` and so on indicate the types of the vectors being operated on. For example, `svld1_f32()` loads 32-bit floats.
- `_predication` indicates a zeroing, z, or merging, m, operation. Predicates control which lanes in a vector are active, and which are inactive. Zeroing operations set inactive elements to zero in an operation. Merging operations leave the existing inactive elements unchanged.

3.3 Auto-vectorization

Auto-vectorization is the process of allowing the compiler to automatically identify opportunities in your code to vectorize. Auto-vectorization includes the following compilation techniques:

- Loop vectorization – Unrolling loops to reduce the number of iterations, while performing more operations in each iteration
- Superword-Level Parallelism (SLP) vectorization – Bundling scalar operations together to use full width Helium instructions.

Auto-vectorization can be specified with the optimization level. Arm Compiler provides a wide range of optimization levels, selected with the -O option.

To have autovectorization disabled, simply select the optimization level to 0.

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -march=armv8.2-a+sve -O0")
```

Instead, to enable autovectorization, simply select the optimization level at 3.

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -march=armv8.2-a+sve -O3")
```

The #pragma clang loop directive allows loop vectorization hints to be specified for the subsequent for, while or do-while. The directive allows vectorization to be enabled or disabled.

```
bool Ban::operator==(const Ban &b) const{
    if(p != b.p)
        return false;

    bool res = true;
    #pragma clang loop vectorize(enable) interleave(enable)
    for(unsigned i=0; i<SIZE; ++i)
        if(res && (num[i] != b.num[i])) // res == true
            res = false;

    return res;
}
```

During the compilation phase, flags can be set to identify and diagnose loops that are skipped by the loop vectorizer. Optimization observations are enabled with:

`-Rpass=loop-vectorize`

identifies loops that have been successfully vectorized.

`-Rpass-missed=loop-vectorize`

identifies loops that have not been vectorized and indicates whether vectorization was specified.

`-Rpass-analysis=loop-vectorize`

identifies the instructions that caused the vectorization to fail. This information was used to figure out which functions we have to write with SVE intrinsics.

```
/home/marco/R3/src/ban.cpp:78:20: remark: Cannot SLP vectorize list: vectorization was impossible with available vectorization facto
rs [-Rpass-missed=slp-vectorizer]
        if(res && (num[i] != b.num[i])) // res == true
                  ^
```

3.4 Vector length agnostic

As mentioned earlier SVE is a vector length agnostic architecture. To ensure that the same application can run on different implementations that support SVE, without the need to recompile the code, we used the code below.

```
inline void function(const double* a, const double* b) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                          // # double nel registro
    int32_t N  = (int32_t)SIZE;                              // # double nel vettore

    for (int32_t i = 0; i < N; i += VL) {
        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                      // prende elementi da 0 a VL
        }

        /* corpo della funzione */

        pg = svwhilelt_b64(i + VL, N);                      // Aggiorna per la prossima iterazione
    }
    return true;
}
```

Initializes an SVE predicate pg as true for 64-bit elements. Retrieves the maximum vector length supported by the SVE implementation and stores it in the VL variable. Sets the variable N to the value of the number of elements in a vector.

The function enters a loop that iterates over the elements of the block vector of size VL. Within the loop, there is an if statement that checks whether the current block (i + VL) goes beyond the end of the vector (N). If it does, it updates VL to make sure that the last block is handled correctly and also updates the pg predicate to account for the adjusted vector length. After the loop body, there is another statement that updates the pg predicate to properly handle the next iteration.

3.5 banintrinSVE.h

Through the setting of flags to enable diagnostic methods, we were able to define the parts of code that the compiler was unable to vectorize. These portions of code were replaced by code that implies the use of intrinsic functions to achieve manual vectorization. The following are the functions we implemented contained in the "banintrinSVE.h" file.

3.5.1 Denoise

```
inline void control(const double*a,int tol, double* dst){
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                         // # double nel registro
    int32_t N  = (int32_t)SIZE;                             // # double nel vettore

    for (int32_t i = 0; i < N; i +=VL) {
        if (i + VL > N) {
            VL = N - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                       // Aggiorna il predicato per coprire solo gli elementi validi
        }
        svfloat64_t va = svld1_f64(pg, &a[i]);
        svfloat64_t vb = svdup_f64(tol);
        svfloat64_t vc = svdup_f64(-tol);

        svbool_t mask_gt = svcmpgt(pg, va, vc);              // gt[i]=true se a>b
        svbool_t mask_lt = svcmplt(pg, va, vb);

        svfloat64_t v0 = svdup_f64(0);
        svfloat64_t vgt = svsel_f64(mask_gt, v0, va);        //0 se va[i] > -tol
        svfloat64_t vlt = svsel_f64(mask_lt, v0, va);        //0 se vai[i] < tol

        svfloat64_t res = svadd_f64_m(pg, vgt, vlt);
        svst1_f64(pg, &dst[i], res);

        pg = svwhilelt_b64(i + VL, N);                       // Aggiorna il predicato per la prossima iterazione
    }
}
```

The loading of the elements of vector a within the SIMD SVE va register is performed. Two SVE constant vectors vb and vc containing the value of tol and its negative opposite (-tol), respectively, are created. Two SVE predicates `mask_gt` and `mask_lt` are created containing true at elements of va greater than -tol and less than tol, respectively. A constant SVE vector v0 containing the value zero is created. Two new SVE vectors vgt and vlt are created that contain the value zero if they are greater than -tol and if they are less than tol, respectively, otherwise they contain the value of va[i]. A new SVE vector res is created containing the element-by-element sum of vgt and vlt. The vector res is stored in the vector dst.

```
inline void control(const float*a, int tol, float* dst){
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                         // # float nel registro
    int32_t N  = (int32_t)SIZE;                             // # float nel vettore

    for (int32_t i = 0; i < N; i +=VL) {
        if (i + VL > N) {
            VL = N - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                       // Aggiorna il predicato per coprire solo gli elementi validi
        }
        svfloat32_t va = svld1_f32(pg, &a[i]);
        svfloat32_t vb = svdup_f32(tol);
        svfloat32_t vc = svdup_f32(-tol);

        svbool_t mask_gt = svcmpgt(pg, va, vc);              // gt[i]=true se a>b
        svbool_t mask_lt = svcmplt(pg, va, vb);

        svfloat32_t v0 = svdup_f32(0);
        svfloat32_t vgt = svsel_f32(mask_gt, v0, va);        //0 se va[i] > -tol
        svfloat32_t vlt = svsel_f32(mask_lt, v0, va);        //0 se vai[i] < tol

        svfloat32_t res = svadd_f32_m(pg, vgt, vlt);
        svst1_f32(pg, &dst[i], res);

        pg = svwhilelt_b32(i + VL, N);                       // Aggiorna il predicato per la prossima iterazione
    }
}
```

The function performs the same operations as the previous one, but goes to work with float elements. The main difference is in the type of vectors on which it operates (32 instead of 64).

3.5.2 Sum

Loading of the elements of vector a within the SIMD SVE va register is performed. Loading of the elements of vector b within the SIMD SVE register vb is performed. A new SVE vector res containing the element-by-element sum of va and vb is created. The vector res is stored in the vector dst.

```
inline void sumvet(const double* a, const double* b, int p, double* dst) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                          // # double nel registro

    for (int32_t i = 0; i < p; i +=VL) {
        if (i + VL > p) {                                    // Calcola la lunghezza del vettore per questa iterazione
            VL = p - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                       // Aggiorna il predicato per coprire solo gli elementi validi
        }
        svfloat64_t va = svld1_f64(pg, &a[i]);
        svfloat64_t vb = svld1_f64(pg, &b[i]);
        svfloat64_t res = svadd_f64_m(pg, va, vb);
        svst1_f64(pg, &dst[i], res);

        pg = svwhilelt_b64(i + VL, p);
    }
}
```

The function performs the same operations as the previous one, but goes to work with float elements. The main difference is in the type of vectors on which it operates (32 instead of 64).

```
inline void sumvet(const float* a, const float* b, int p, float* dst) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                          // # float nel registro

    for (int32_t i = 0; i < p; i +=VL) {
        if (i + VL > p) {                                    // Calcola la lunghezza del vettore per questa iterazione
            VL = p - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                       // Aggiorna il predicato per coprire solo gli elementi validi
        }
        svfloat32_t va = svld1_f32(pg, &a[i]);
        svfloat32_t vb = svld1_f32(pg, &b[i]);
        svfloat32_t res = svadd_f32_m(pg, va, vb);
        svst1_f32(pg, &dst[i], res);

        pg = svwhilelt_b32(i + VL, p);
    }
}
```

3.5.3 All zeros

Loading of the elements of vector a within the SIMD SVE va register is performed. A new SVE vector v0 containing the value zero is created. An SVE result predicate is created that contains true at the elements of va that are equal to zero. A new SVE predicate `result_rev` is created that contains the logical complement of result, thus true at the elements of va that are non-zero. A check is made using `svptest_any` to see if there is at least one element in `result_rev` (i.e., a non- zero element). If at least one non-zero element is present, the function returns false, otherwise it continues with the next iteration of the loop. If the loop ends without having returned false, it means that all elements in the vector are equal to zero, so the function returns true.

```
inline bool allzeros(const double* a) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                         // # double nel registro
    int32_t N  = (int32_t)SIZE;                             // # double nel vettore

    for (int32_t i = 0; i < N; i += VL) {
        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                      // prende elementi da 0 a VL
        }
        svfloat64_t va = svld1_f64(pg, &a[i]);
        svfloat64_t v0 = svdup_f64(0.0);
        svbool_t result = svcmpeq_f64(pg, va, v0);          // result[i] = true se va[i] = 0

        svbool_t result_rev = svnot_b_z(pg, result);        // result[i] = false se va[i] = 0
        if (svptest_any(pg, result_rev) != false) {
            return false;                                   // elemento != 0 -> false
        }

        pg = svwhilelt_b64(i + VL, N);                      // Aggiorna per la prossima iterazione
    }
    return true;
}
```

The function performs the same operations as the previous one, but goes to work with float elements. The main difference is in the type of vectors on which it operates (32 instead of 64).

```
inline bool allzeros(const float* a) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                         // # float nel registro
    int32_t N  = (int32_t)SIZE;                             // # float nel vettore

    for (int32_t i = 0; i < N; i += VL) {
        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                      // prende elementi da 0 a VL
        }
        svfloat32_t va = svld1(pg, &a[i]);
        svfloat32_t v0 = svdup_f32(0.0f);
        svbool_t result = svcmpeq(pg, va, v0);              // result[i] = true se va[i] = 0

        svbool_t result_rev = svnot_b_z(pg, result);        // result[i] = false se va[i] = 0
        if (svptest_any(pg, result_rev) != false) {
            return false;                                   // elemento != 0 -> false
        }

        pg = svwhilelt_b32(i + VL, N);                      // Aggiorna per la prossima iterazione
    }
    return true;
}
```

3.5.4 All equal

Loading of the elements of vector a within the SIMD SVE va register is performed. Loading of the elements of vector b within the SIMD SVE register vb is performed. An SVE result predicate is created that contains true at the elements of va and vb that are equal to each other. A new SVE predicate `result_rev` is created that contains the logical complement of result, thus true at the elements of va and vb that are different from each other. A check is made using `svptest_any` to see if there is at least one element in `result_rev` (i.e., a different element). If at least one different element is present, the function returns false, otherwise it continues with the next iteration of the loop. The predicate pg is updated to properly handle the next iteration of the loop. If the loop ends without having returned false, it means that all elements of vectors a and b are equal to each other, so the function returns true.

```
inline bool allequal(const double* a, const double* b) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                          // # double nel registro
    int32_t N  = (int32_t)SIZE;                              // # double nel vettore

    for (int32_t i = 0; i < N; i += VL) {
        if (i + VL > N) {
            VL = N - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                       // prende elementi da 0 a VL
        }
        svfloat64_t va = svld1(pg, &a[i]);
        svfloat64_t vb = svld1(pg, &b[i]);
        svbool_t result = svcmpeq_f64(pg, va, vb);           // result[i] = true se va[i] = vb[i]

        svbool_t result_rev = svnot_b_z(pg, result);         // result[i] = false se va[i] = vb[i]
        if (svptest_any(pg, result_rev) != false) {
            return false;                                    // elemento != 0 -> false
        }

        pg = svwhilelt_b64(i + VL, N);                       // Aggiorna per la prossima iterazione
    }
    return true;
}
```

The function performs the same operations as the previous one, but goes to work with float elements. The main difference is in the type of vectors on which it operates (32 instead of 64).

```
inline bool allequal(const float* a, const float* b) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                          // # float nel registro
    int32_t N  = (int32_t)SIZE;                              // # float nel vettore

    for (int32_t i = 0; i < N; i += VL) {
        if (i + VL > N) {
            VL = N - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                       // prende elementi da 0 a VL
        }
        svfloat32_t va = svld1(pg, &a[i]);
        svfloat32_t vb = svld1(pg, &b[i]);
        svbool_t result = svcmpeq(pg, va, vb);               // result[i] = true se va[i] = vb[i]

        svbool_t result_rev = svnot_b_z(pg, result);         // result[i] = false se va[i] = vb[i]
        if (svptest_any(pg, result_rev) != false) {
            return false;                                    // elemento != 0 -> false
        }

        pg = svwhilelt_b32(i + VL, N);                       // Aggiorna per la prossima iterazione
    }
    return true;
}
```

3.5.5 Compare all

```
inline int cmpall(const double* a, const double* b) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                         // # double nel registro
    int32_t N  = (int32_t)SIZE;                             // # double nel vettore

    for (int32_t i = 0; i < N; i += VL) {                   // Lunghezza del vettore per questa iterazione

        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                      // prende elementi da 0 a VL
        }

        svfloat64_t va = svld1_f64(pg, &a[i]);
        svfloat64_t vb = svld1_f64(pg, &b[i]);

        svbool_t mask_gt = svcmpgt(pg, va, vb);             // gt[i]=true se a>b
        svbool_t mask_lt = svcmplt(pg, va, vb);             // gt[i]=true se a<b

        svuint64_t indices = svindex_u64(0, 1);
        uint32_t gt_index = svminv_u64(pg, svsel_u64(mask_gt, indices, svdup_u64(UINT64_MAX)));
        uint32_t lt_index = svminv_u64(pg, svsel_u64(mask_lt, indices, svdup_u64(UINT64_MAX)));

        if (gt_index < lt_index) return 1;                  // a > b
        if (lt_index < gt_index) return -1;                 // a < b

        pg = svwhilelt_b64(i + VL, N);                      // Aggiorna per la prossima iterazione
    }
    return 0;                                               // a = b
}
```

Loading of the elements of vector a within the SIMD SVE va register is performed.

Loading of the elements of vector b within the SIMD SVE vb register is performed. Two SVE predicates `mask_gt` and `mask_lt` are created that contain true at the elements of va greater than and less than the corresponding elements in vb, respectively. An SVE indices vector is created containing the values of indices 0, 1, 2, ... up to VL-1.

Two registers are created having the index value matching true in `mask_gt` and `mask_lt`, otherwise they are set to `UINT64_MAX`. For each of the two registers, the smallest value is selected using `svminv_u64`. A comparison is made between the two selected values. If the value of the first element (`gt_index`) is less than the value of the second element (`lt_index`), the function returns 1 ($a > b$). If the value of the first element (`gt_index`) is greater than the value of the second element (`lt_index`), the function returns -1 ($a < b$). The predicate pg is updated to properly handle the next iteration of the loop. If the loop ends without returning 1 or -1, it means that all elements of vectors a and b are equal to each other, so the function returns 0 ($a = b$).

```
inline int cmpall(const float* a, const float* b) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                         // # float nel registro
    int32_t N  = (int32_t)SIZE;                             // # float nel vettore

    for (int32_t i = 0; i < N; i += VL) {                   // Lunghezza del vettore per questa iterazione
        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                      // prende elementi da 0 a VL
        }
        svfloat32_t va = svld1(pg, &a[i]);
        svfloat32_t vb = svld1(pg, &b[i]);

        svbool_t mask_gt = svcmpgt(pg, va, vb);             // gt[i]=true se a>b
        svbool_t mask_lt = svcmplt(pg, va, vb);             // gt[i]=true se a<b

        svuint32_t indices = svindex_u32(0, 1);
        uint32_t gt_index = svminv_u32(pg, svsel_u32(mask_gt, indices, svdup_u32(UINT32_MAX)));
        uint32_t lt_index = svminv_u32(pg, svsel_u32(mask_lt, indices, svdup_u32(UINT32_MAX)));

        if (gt_index < lt_index) return 1;                  // a > b
        if (lt_index < gt_index) return -1;                 // a < b

        pg = svwhilelt_b32(i + VL, N);
    }
    return 0;                                               // a = b
}
```

3.5.6 Compare zero

```
inline int cmp0(const double* a) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                         // # double nel registro
    int32_t N  = (int32_t)SIZE;                             // # double nel vettore

    for (int32_t i = 0; i < N; i += VL) {                   // Lunghezza del vettore per questa iterazione

        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                      // prende elementi da 0 a VL
        }

        svfloat64_t va = svld1_f64(pg, &a[i]);
        svfloat64_t vb = svdup_f64(0.0);

        svbool_t mask_gt = svcmpgt(pg, va, vb);             // gt[i]=true se a>b
        svbool_t mask_lt = svcmplt(pg, va, vb);             // gt[i]=true se a<b

        svuint64_t indices = svindex_u64(0, 1);
        uint32_t gt_index = svminv_u64(pg, svsel_u64(mask_gt, indices, svdup_u64(UINT64_MAX)));
        uint32_t lt_index = svminv_u64(pg, svsel_u64(mask_lt, indices, svdup_u64(UINT64_MAX)));

        if (gt_index < lt_index) return 1;                  // a > 0
        if (lt_index < gt_index) return -1;                 // a < 0

        pg = svwhilelt_b64(i + VL, N);                      // Aggiorna per la prossima iterazione
    }
    return 0;                                               // a = 0
}
```

The loading of the elements of vector a within the SIMD SVE va register is performed. An SVE vector vb containing the value zero is created. Two SVE predicates `mask_gt` and `mask_lt` are created containing true at the elements of va greater than and less than zero, respectively. An SVE vector indices is created containing the values of indices 0, 1, 2, ... up to VL-1. Two registers are created having the index value matching true in `mask_gt` and `mask_lt`, otherwise they are set to `UINT64_MAX`. For each of the two registers, the smallest value is selected using `svminv_u64`. A comparison is made between the two selected values. If the value of the first element (`gt_index`) is less than the value of the second element (`lt_index`), the function returns 1 ($a > 0$). If the value of the first element (`gt_index`) is greater than the value of the second element (`lt_index`), the function returns -1 ($a < 0$). The predicate pg is updated to properly handle the next iteration of the loop. If the loop ends without returning 1 or -1, it means that all elements of vectors va and vb are equal to each other, so the function returns 0 ($a = 0$).

```
inline int cmp0(const float* a) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                         // # float nel registro
    int32_t N  = (int32_t)SIZE;                             // # float nel vettore

    for (int32_t i = 0; i < N; i += VL) {                   // Lunghezza del vettore per questa iterazione
        if (i + VL > N) {
            VL = N - i;                                     // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                      // prende elementi da 0 a VL
        }

        svfloat32_t va = svld1(pg, &a[i]);
        svfloat32_t vb = svdup_f32(0);

        svbool_t mask_gt = svcmpgt(pg, va, vb);             // gt[i]=true se a>b
        svbool_t mask_lt = svcmplt(pg, va, vb);             // gt[i]=true se a<b

        svuint32_t indices = svindex_u32(0, 1);
        uint32_t gt_index = svminv_u32(pg, svsel_u32(mask_gt, indices, svdup_u32(UINT32_MAX)));
        uint32_t lt_index = svminv_u32(pg, svsel_u32(mask_lt, indices, svdup_u32(UINT32_MAX)));

        if (gt_index < lt_index) return 1;                  // a > 0
        if (lt_index < gt_index) return -1;                 // a < 0

        pg = svwhilelt_b32(i + VL, N);                      // Aggiorna per la prossima iterazione
    }
    return 0;                                               // a = 0
}
```

3.5.7 Find first non zero

```
inline int findFirstNonZero(const double* a) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                          // # double nel registro
    int32_t N  = (int32_t)SIZE;                              // # double nel vettore

    for (int32_t i = 0; i < N; i += VL) {                    // Lunghezza del vettore per questa iterazione
        if (i + VL > N) {
            VL = N - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                       // prende elementi da 0 a VL
        }
        svfloat64_t va = svld1(pg, &a[i]);
        svfloat64_t vb = svdup_f64(0.0);

        svbool_t mask = svcmpne(pg, va, vb);                 // mask[i] = true se va[i] != 0
        svuint64_t indices = svindex_u64(0, 1);
        uint64_t min_index = svminv_u64(pg, svsel(mask, indices, svdup_u64(UINT64_MAX)));

        if( min_index != UINT64_MAX)
            return i + min_index;                            // 0 passati + clz relativo

        pg = svwhilelt_b64(i + VL, N);                       // Aggiorna per la prossima iterazione
    }
    return N;                                                // tutti elementi a 0
}
```

The loading of the elements of vector a within the SIMD SVE va register is performed. An SVE vb vector containing the value zero is created.

An SVE mask predicate is created that contains true at the non-zero elements of va. An SVE indices vector is created containing the values of indices 0, 1, 2, ... up to VL-1. The index of the first nonzero element is computed using `svminv_u64`. If there is no nonzero element (i.e., all elements are zero), the index is set to `UINT64_MAX`. A check is made to see if a nonzero element was found. If a nonzero element is present, the function returns the index calculated as i + `min_index`, which represents the absolute index in the vector a. The predicate pg is updated to properly handle the next iteration of the loop. If the loop ends without having returned an index, it means that all elements in vector a are zero, so the function returns N.

```
inline int findFirstNonZero(const float* a) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                          // # flaot nel reigstro
    int32_t N  = (int32_t)SIZE;                              // # flaot nel vettore

    for (int32_t i = 0; i < N; i += VL) {                    // Lunghezza del vettore per questa iterazione
        if (i + VL > N) {
            VL = N - i;                                      // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                       // prende elementi da 0 a VL
        }
        svfloat32_t va = svld1(pg, &a[i]);
        svfloat32_t vb = svdup_f32(0);

        svbool_t mask = svcmpne(pg, va, vb);                 // mask[i] = true se va[i] != 0
        svuint32_t indices = svindex_u32(0, 1);
        uint32_t min_index = svminv_u32(pg, svsel(mask, indices, svdup_u32(UINT32_MAX)));

        if( min_index != UINT32_MAX)
            return i + min_index;                            // 0 passati + clz relativo

        pg = svwhilelt_b32(i + VL, N);                       // Aggiorna per la prossima iterazione
    }
    return N;                                                // tutti elementi a 0
}
```

The function performs the same operations as the previous one, but goes to work with float elements. The main difference is in the type of vectors on which it operates (32 instead of 64).

3.5.8 Convolution

For the convolution operation, we started by writing a possible version that did not take into account the size of the registers. The code that performs this operation is as follows.

```
inline int convmul(const double* a, const double* b, double* dst) {
    svbool_t pg = svwhilelt_b64(0, SIZE);
    svfloat64_t va0;
    svfloat64_t vb07  = svld1_f64 (pg, &b[0]);          //serve caricarlo la prima volta
    svfloat64_t va0b = svdup_f64(0);
    svfloat64_t accr = svdup_f64(0);

    for (int i = 0 ; i < SIZE ; i++){                   // per ogni riga
        va0  = svdup_f64(a[i]);
        if ( i != 0 ){
            vb07  = svinsr_n_f64(vb07, 0);              // inserisce 0 in testa con shift a dx
        }
        va0b  = svmul_f64_m (pg, va0,  vb07);
        accr = svadd_f64_m(pg, accr, va0b);
    }

    svst1_f64(pg, &dst[0], accr);
    return 0;
}
```

Initialize an SVE pg predicate as true using `svwhilelt_b64` to cover the entire vector of size SIZE. This predicate defines the active part of the vector to work on during function execution.

Load the first element of vector b into the SIMD SVE vb register using `svld1_f64`. This value will be used to multiply it with the elements of vector a during convolution. Create three SIMD SVE registers va, vab, and accr initialized to zero using `svdup_f64(0)`. va will contain the elements of vector a, vab will contain the result of the multiplication between va and vb, and accr will contain the accumulation of all multiplications performed so far. Enter a for loop that iterates for each element of vector a. Within the loop, loads the current element of vector a into the SIMD SVE va register using `svdup_f64(a[i])`. It checks whether the index i is non-zero. If i is nonzero, it performs a "shift to the right" (`svinsr_n_f64`) of vb by inserting a zero value at the register head position. This corresponds to a "shift" to the right of vb. It performs multiplication between va and vb, storing the result in the SIMD SVE register vab. It adds the resulting value vab to the accumulation register accr using `svadd_f64_m`. It continues with the next iteration of the loop for the next element of the vector a. After the loop completes, stores the accumulated accr result in the dst array using `svst1_f64` to save the final convolution result. It returns 0 to indicate the completion of the function. The result of this code for vectors of 8 elements is the following operation:

a[o]*b[o]	a[o]*b[1]	a[0]*b[2]	a[0]*b[3]	a[0]*b[4]	a[0]*b[5]	a[0]*b[6]	a[0]*b[7]
0	a[1]*b[0]	a[1]*b[1]	a[1]*b[2]	a[1]*b[3]	a[1]*b[4]	a[1]*b[5]	a[1]*b[6]
0	0	a[2]*b[0]	a[2]*b[1]	a[2]*b[2]	a[2]*b[3]	a[2]*b[4]	a[2]*b[5]
0	0	0	a[3]*b[0]	a[3]*b[1]	a[3]*b[2]	a[3]*b[3]	a[3]*b[4]
0	0	0	0	a[4]*b[0]	a[4]*b[1]	a[4]*b[2]	a[4]*b[3]
0	0	0	0	0	a[5]*b[0]	a[5]*b[1]	a[5]*b[2]
0	0	0	0	0	0	a[6]*b[0]	a[6]*b[1]
0	0	0	0	0	0	0	a[7]*b[0]
dst[0]	dst[1]	dst[2]	dst[3]	dst[4]	dst[5]	dst[6]	dst[7]

In order to make this function vector length agnostic, it was necessary to handle backup registers with two for loops to ensure that the shift operation was performed correctly. We also inserted an additional check in order to avoid multiplications by zero, thus going to reduce the number of multiplications needed. The final code for the convolution operation is the following:

```
inline int convmul(const double* a, const double* b, double* dst) {
    svbool_t pg = svptrue_b64();
    int32_t VL = (int32_t)svcntd();                                 // # double nel registro
    int32_t N  = (int32_t)SIZE;                                     // # double nel vettore
    svfloat64_t va, vb, vaxb, vb_app, res_add;

    int index_p, fast_f;                                            // index_p = indice per caricare il vettore precedente ; fast_f = stop condition per convoluzione sotto la diagonale
    int VL_fix = (int32_t)svcntd();
    svbool_t pg_fix = svptrue_b64();

    for (int32_t i = 0; i < N; i += VL) {
        index_p= i ;

        if (i + VL > N) {                                           // Lunghezza del vettore per questa iterazione
            VL = N - i;                                             // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b64(0, VL);                              // prende elementi da 0 a VL
        }

        vb = svld1_f64(pg, &b[i]);                                  // carico b ora perche lo aggiorno alla fine del for
        res_add = svdup_f64(0);

        for (int j = 0 ; j < SIZE ; j++){                           // per ogni riga
            va  = svdup_f64(a[j]);
            vaxb = svmul_f64_m(pg, va,  vb);
            res_add = svadd_f64_m(pg, res_add, vaxb);

            if (j % VL_fix == 0) {                                  // momento di aggiornare il registro di appoggio, se i=j siamo sulla diagnonale
                index_p = (j == i) ? index_p : index_p - VL_fix;                    // aggiorno index blocco da prendere
                vb_app = (j == i) ? svdup_f64(0) : svld1_f64(pg_fix, &b[index_p]);  // aggiorno il registro di appoggio
                fast_f = (j == i) ? 4 : UINT64_MAX;                                 // aggiorno stop condition se siamo sulla diagnoale ci mancano 4 operazioni prima del braek;
            }
            fast_f --;

            vb = svinsr_n_f64(vb, vb_app[VL_fix-(j%VL_fix)-1]);     // prendo i valori partendo dal fondo del blocco
            if ( fast_f == 0)                                       // alla fine della diagonale stop condition
                break;
        }
        svst1_f64(pg, &dst[i], res_add);

        pg = svwhilelt_b64(i + VL, N);                              // Aggiorna per la prossima iterazione
    }
    return 0;
}
```

It creates va, vb, vaxb, `vb_app`, `res_add`, `index_p`, and `fast_f` variables for operands, intermediate results, and control variables. A second SVE predicate `pg_fix` is created as true using `svptrue_b64()` and a variable `VL_fix` representing the block size that will be used to update vb at each internal iteration of the loop. The first for loop iterates over the elements of the VL block-sized vector. Within the outer loop, `index_p` is initialized with i, which represents the starting index to load vector b in each inner iteration of the loop. The first elements of vector b are loaded into the SIMD SVE vb register using `svld1_f64`. A SIMD SVE register `res_add` initialized to zero is created, which will be used to accumulate the intermediate results of the multiplications. The second for loop iterates for each element of vector a. Within the inner loop, the current element of vector a is loaded into the SIMD SVE va register using `svdup_f64(a[j])`. The product between va and vb is made and stored in the SIMD SVE register vaxb. The accumulation register `res_add` is updated by summing vaxb using `svadd_f64_m`. It is checked whether it is time to update the support register `vb_app` using vb to get a new block of elements from b. This is done when index j reaches a multiple of `VL_fix`. It is updated vb with `vb_app` by taking values starting at the bottom of the block. A `fast_f` count is performed to determine when to stop the convolution. When `fast_f` reaches zero, the convolution is stopped as the diagonal is completed. The accumulated result `res_add` in the dst array using `svst1_f64` is saved. The predicate pg is updated to properly handle the next iteration of the outer loop.

```
inline int convmul(const float* a, const float* b, float* dst) {
    svbool_t pg = svptrue_b32();
    int32_t VL = (int32_t)svcntw();                                 // # flaot nel reigstro
    int32_t N  = (int32_t)SIZE;                                     // # flaot nel vettore
    svfloat32_t va, vb, vaxb, vb_app, res_add;

    int index_p, fast_f;                                            // index_p = indice per caricare il vettore precedente ; fast_f = stop condition per convoluzione sotto la diagonale
    int VL_fix = (int32_t)svcntd();
    svbool_t pg_fix = svptrue_b32();

    for (int32_t i = 0; i < N; i += VL) {
        index_p= i ;

        if (i + VL > N) {                                           // Lunghezza del vettore per questa iterazione
            VL = N - i;                                             // Aggiusta la lunghezza se supera la fine del vettore
            pg = svwhilelt_b32(0, VL);                              // prende elementi da 0 a VL
        }

        vb = svld1_f32(pg, &b[i]);                                  // carico b ora perche lo aggiorno alla fine del for
        res_add = svdup_f32(0);

        for (int j = 0 ; j < SIZE ; j++){                           // per ogni riga
            va  = svdup_f32(a[j]);
            vaxb = svmul_f32_m(pg, va,  vb);
            res_add = svadd_f32_m(pg, res_add, vaxb);

            if (j % VL_fix == 0) {                                  // momento di aggiornare il registro di appoggio, se i=j siamo sulla diagnonale
                index_p = (j == i) ? index_p : index_p - VL_fix;                    // aggiorno index blocco da prendere
                vb_app  = (j == i) ? svdup_f32(0) : svld1_f32(pg_fix, &b[index_p]);     // aggiorno il registro di appoggio
                fast_f  = (j == i) ? 4 : UINT32_MAX;                                // aggiorno stop condition se siamo sulla diagnoale ci mancano 4 operazioni prima del braek;
            }
            fast_f --;

            vb = svinsr_n_f32(vb, vb_app[VL_fix-(j%VL_fix)-1]);     // prendo i valori partendo dal fondo del blocco
            if ( fast_f == 0)                                       // alla fine della diagonale stop condition
                break;
        }
        svst1_f32(pg, &dst[i], res_add);

        pg = svwhilelt_b32(i + VL, N);                              // Aggiorna per la prossima iterazione
    }
    return 0;
}
```

The function performs the same operations as the previous one, but goes to work with float elements. The main difference is in the type of vectors on which it operates (32 instead of 64).

4 NEC Aurora

4.1 Overview

NEC's innovative platform, SX-Aurora TSUBASA, is specifically designed to address complex mathematical, scientific, and engineering problems. NEC's proprietary computing systems offer comprehensive solutions for various requirements by integrating a rich array of x86-based products and storage appliances.

A key component of this platform is the NEC SX-Aurora Vector Engine (VE), which leverages the proven vector computing technique from NEC's long history in supercomputing. With its vector computation mechanism, large memory bandwidth, and a small number of powerful cores, this architecture provides a robust foundation for achieving high sustained performance. This supercomputing platform represents a vital resource for tackling the most complex and cutting-edge challenges in scientific and technological research.

4.2 VE Intrinsics

In order to use the intrinsic functions for NEC Aurora contained in the velintrin.h header, LLVM-VE must be installed. LLVM-VE is a community-supported open source compiler and is not included in the official SX-Aurora TSUBASA SDK. Once LLVM-VE is installed, the compiler can be invoked as follows:

/home/tesi1/c/rel/bin/clang++ -target ve-linux

4.2.1 Vector Type

VE Intrinsics introduce vector type `__vr` and functions that operate on `__vr` type variables. Typical code with intrinsics starts with loading data into `__vr` type variables, then works with them, and finally stores to memory. `__vr` is defined as below. It can hold 256 x 64bit values (2048KB in total).

```
typedef double __vr __attribute__(( vector_size__(2048))
```

It is important to know that VE has 64 vector registers. Using too many `__vr` variables results in vector register spills.

4.2.2 Intrinsic Function

Function format is:

```
_vel_<asm>_<suffix>
```

<asm> is an instruction mnemonic in the assembly manual. Some assembly instructions have mnemonic suffix to give additional means to an instruction. For example, vfaddd instruction has d suffix to show that operands are handled as fp64.

<suffix> is a list of types of return value and arguments in a single character:

- v: vector

- s: scalar
- m and M: vector mask for 256 elements and 512 elements
- l: vector length

4.2.3 Vector length

Vector length argument shows number of elements updated by the function. For example:

```
va = _vel_vfaddd_vvvl(vb, vc, 100)
```

updates first 100 elements of va with the result of vb + vc. Elements after vector length, becomes undefined.

4.2.4 Vector Mask

VE has eight 256 bit vector mask registers. A vector mask register is given to a vector instruction to mask elements of its output vector. When an element is masked, i.e. a bit is zero, such element is not updated by the instruction. VE intrinsics introduce `__vm256` type to hold a vector mask. Intrinsic functions has variants that accept `__vm256` variable as an argument.

4.3 Auto-vectorization

LLVM has two vectorizers: The Loop Vectorizer, which operates on Loops, and the SLP Vectorizer. These vectorizers focus on different optimization opportunities and use different techniques. The SLP vectorizer merges multiple scalars that are found in the code into vectors while the Loop Vectorizer widens instructions in loops to operate on multiple consecutive iterations. The Loop Vectorizer is enabled by default, but it can be disabled through clang using the command line flag:

```
$ clang ... -fno-vectorize  file.c
```

Also the SLP Vectorizer is enabled by default, but it can be disabled through clang using the command line flag:

```
$ clang -fno-slp-vectorize file.c
```

The #pragma clang loop directive allows loop vectorization hints to be specified for the subsequent for, while or do-while. The directive allows vectorization to be enabled or disabled.

```
void Ban::_mul_conv(const T num_a[SIZE], const T num_b[SIZE], T aux[(SIZE<<1)-1]){

    #pragma clang loop vectorize(enable) interleave(enable)
    for(unsigned i=0; i<(SIZE<<1)-1; ++i){
        aux[i] = 0;
        for(unsigned j=0; j<SIZE; ++j)
            if((i-j)>=0 && (i-j)<SIZE)
                aux[i] += num_a[i-j]*num_b[j];
    }
}
```

During the compilation phase, flags can be set to identify and diagnose loops that are skipped by the loop vectorizer. Optimization observations are enabled with:

-Rpass=loop-vectorize

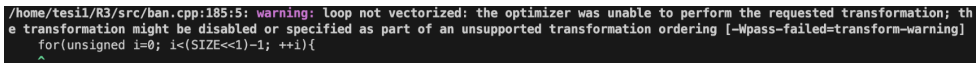
identifies loops that have been successfully vectorized.

-Rpass-missed=loop-vectorize

identifies loops that have not been vectorized and indicates whether vectorization was specified.

-Rpass-analysis=loop-vectorize

identifies the instructions that caused the vectorization to fail.
This information was used to figure out which functions we have to write with VE intrinsics.



In order to allow VE intrinsics functions to properly handle the loading and saving of data from arrays to vector registers, it was necessary to use data aligned according to the data type. For example, in the case of BANs composed of elements in double format, we modified the code in the ban.h file as follows:

```
typedef double T;

#pragma pack(push, 8)
class Ban{
    T num[SIZE];
    int p;
};
#pragma pack(pop)
```

4.4 banintrinNEC.h

Through the setting of flags to enable diagnostic methods, we were able to define the parts of code that the compiler was unable to vectorize. These portions of code were replaced by code that implies the use of intrinsic functions to achieve manual vectorization. The following are the functions we implemented contained in the "banintrinNEC.h" file.

4.4.1 Vectorial Sum

```
inline void vecsum(const double* a, const double* b, double* dst) {

    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vb = _vel_vld_vssl(8, b, SIZE);

    __vr accr = _vel_vfaddd_vvvl(va, vb, SIZE);

    _vel_vst_vssl(accr, 8, dst, SIZE);
}
```

Three constant pointers to doubles `a`, `b`, and `dst` are declared, representing the input vectors and the sum output. The function begins by loading the memory portions of vectors `a` and `b` into the vector registers `va` and `vb`. The `_vel_vld_vssl` method is used to perform the loading, specifying the size of the elements (8 bytes per double) and the total size of the vectors (`SIZE`) to be loaded. A vector sum of the values contained in the `va` and `vb` registers is performed, and the result is stored in the `accr` vector register. The `_vel_vfaddd_vvvl` function sums the vector elements present in `va` and `vb` and stores the result in `accr`. Finally, the result obtained in `accr` is written to the destination vector memory `dst`. The `_vel_vst_vssl` function does the saving, specifying the size of the elements (8 bytes per double) and the total size of the vectors (`SIZE`) to be written.

```
inline void vecsum(const float* a, const float* b, float* dst) {

    __vr va = _vel_vldu_vssl(4, a, SIZE);
    __vr vb = _vel_vldu_vssl(4, b, SIZE);

    __vr accr = _vel_vfadds_vvvl(va, vb, SIZE);

    _vel_vstu_vssl(accr, 4, dst, SIZE);

}
```

The function performs the same operations as the previous one, but it works with float elements. The main differences are in the type of the methods such as `_vel_vldu_vssl`, `_vel_vfadds_vvvl` and `_vel_vstu_vssl` which specify the size of the elements as 4 bytes.

4.4.2 Multiplication

```
inline void mul(const double* a, double n, double* dst) {

    __vr va = _vel_vld_vssl(8, a, SIZE);

    __vr accr = _vel_vfmuld_vsvl(n, va, SIZE);

    _vel_vst_vssl(accr, 8, dst, SIZE);
}
```

Two pointers are declared: a and dst, both doubles, and n, which represents the scalar with which to multiply the vector a. The function begins by loading the memory elements of vector a into the vector va register. Vector multiplication of the scalar value n with the values contained in the va vector register is performed, and the result is stored in the accr vector register. The `_vel_vfmuld_vsvl` function multiplies the scalar n with the vector elements in va and stores the result in accr. Finally, the result obtained in accr is written to the memory of the destination vector dst. The `_vel_vst_vssl` function performs the storage.

```
inline void mul(const float* a, float n, float* dst) {

    __vr va = _vel_vldu_vssl(4, a, SIZE);

    __vr accr = _vel_vfmuls_vsvl(n, va, SIZE);

    _vel_vstu_vssl(accr, 4, dst, SIZE);
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vfmuls_vsvl` and `_vel_vstu_vssl` specifying the size of the elements at 4 bytes.

4.4.3 Division

```
inline void div(const double* a, double n, double* dst) {

    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vb = _vel_vbrdd_vsl(n, SIZE);

    __vr accr = _vel_vfdivd_vvvl(va, vb, SIZE);

    _vel_vst_vssl(accr, 8, dst, SIZE);
}
```

Two pointers are declared: `a` and `dst`, both to double numbers, and `n`, which represents the scalar with which to divide the vector `a`. The function begins by loading the memory elements of vector `a` into the vector register `va`. The scalar value `n` is loaded into the vector register `vb` using the `_vel_vbrdd_vsl` function. This function copies the scalar value to all elements of the vector register. Vector division of the values contained in the `va` vector register by the scalar value `n` is performed, and the result is stored in the `accr` vector register. The function `_vel_vfdivd_vvvl` performs the division between the vector elements in `va` and the corresponding elements of the vector register `vb` containing the scalar `n`, storing the result in `accr`. Finally, the result obtained in `accr` is written to the memory of the destination vector `dst`.

```
inline void div(const float* a, float n, float* dst) {

    __vr va = _vel_vldu_vssl(4, a, SIZE);
    __vr vb = _vel_vbrds_vsl(n, SIZE);

    __vr accr = _vel_vfdivs_vvvl(va, vb, SIZE);

    _vel_vstu_vssl(accr, 4, dst, SIZE);
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vbrds_vsl`, `_vel_vfdivs_vvvl` and `_vel_vstu_vssl` specifying the size of the elements at 4 bytes.

4.4.4 Sum

```
inline void sum(const double* a, const double* b, double* dst, int diff) {

    __vr va = _vel_vld_vssl(8, a, SIZE-diff);
    __vr vb = _vel_vld_vssl(8, b, SIZE-diff);

    __vr accr = _vel_vfaddd_vvvl(va, vb, SIZE-diff);

    _vel_vst_vssl(accr, 8, dst, SIZE-diff);
}
```

Three double pointers `a`, `b`, and `dst` are declared, representing the input vectors and the output of the sum. In addition, there is also an integer parameter `diff` indicating the size to be excluded from the sum. The function begins by loading the memory portions of vectors `a` and `b` into the special vector registers `va` and `vb`, specifying the total size of the vectors (SIZE - `diff`) to be loaded. A vector sum of the values contained in the `va` and `vb` registers is performed, and the result is stored in the vector register `accr`. Finally, the result obtained in `accr` is written to the memory of the destination vector `dst`. The `_vel_vst_vssl` function performs the writing, specifying the total size of the vector (SIZE - `diff`) to be written.

```
inline void sum(const float* a, const float* b, float* dst, int diff) {

    __vr va = _vel_vldu_vssl(4, a, SIZE-diff);
    __vr vb = _vel_vldu_vssl(4, b, SIZE-diff);

    __vr accr = _vel_vfadds_vvvl(va, vb, SIZE-diff);

    _vel_vstu_vssl(accr, 4, dst, SIZE-diff);
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vfadds_vvvl` and `_vel_vstu_vssl` specifying the size of the elements at 4 bytes.

4.4.5 Denoise

```
inline void control(const double* a, double tol, double* dst) {

    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vb = _vel_vbrdd_vsl(tol, SIZE);
    __vr vc = _vel_vbrdd_vsl(-tol, SIZE);
    __vr vzero = _vel_vbrdd_vsl(0, SIZE);
    __vr vone = _vel_vbrdd_vsl(1, SIZE);

    __vr vgt = _vel_vfcmpd_vvvl(va, vc, SIZE);                    // vgt[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vr vlt = _vel_vfcmpd_vvvl(va, vb, SIZE);                    // vlt[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vr vmul = _vel_vfmuld_vvvl(vgt, vlt, SIZE);                 // vmul[i] < 0 se a[i] < tol && a[i] > -tol

    __vm256 mask = _vel_vfmkdge_mvl(vmul, SIZE);                  // 1 se vsum[i] >= 0

    __vr accr = _vel_vfmuld_vvvmvl(vone, va, mask, vzero, SIZE);  //accr[i] = mask[i] ? vone[i] * va[i] : vzero[i]

    _vel_vst_vssl(accr, 8, dst, SIZE);
}
```

Two double pointers `a` and `dst` are declared, representing the input vector and output of the function, and a tolerance value `tol`. The function begins by loading the memory elements of vector `a` into the vector register `va`. Four vector registers are created: `vb` with all elements set to the value `tol`, `vc` with all elements set to the negative value of `tol`, `vzero` with all elements set to 0, and `vone` with all elements set to 1. Two vector comparison operations are performed using the functions `_vel_vfcmpd_vvvl`.

The first compares values in `va` with values in `vc`, while the second compares values in `va` with values in `vb`. These comparison operations generate two new vector registers, `vgt` and `vlt`, where `vgt`[i] will be equal to a positive value if `a`[i] $>$ `-tol`, 0 if `a`[i] $==$ `-tol`, and negative if `a`[i] $<$ `-tol`, and `vlt`[i] will be equal to a positive value if `a`[i] $>$ `tol`, 0 if `a`[i] $==$ `tol`, and negative if `a`[i] $<$ `tol`.

A vector multiplication between `vgt` and `vlt` is performed using the function `_vel_vfmuld_vvvl`. This operation will produce a new vector register `vmul`, where `vmul`[i] will be negative only if `a`[i] is between `-tol` and `tol`.

Vector masks (`__vm256`) are used to obtain the final result of the control operation. The `_vel_vfmkdge_mvl` function creates a mask where elements are set to 1 only if `vmul`[i] $>=$ 0, otherwise they are set to 0.

A vector multiplication is performed between `vone`, `va`, and the mask `mask` using the function `_vel_vfmuld_vvvmvl`. This operation will produce a new vector register `accr`, where `accr`[i] will be equal to `a`[i] only if `mask`[i] is set to 1, otherwise it will be equal to 0.

Finally, the result obtained in `accr`is written to the destination vector memory dst.

```
inline void control(const float* a, float tol, float* dst) {

    __vr va = _vel_vldu_vssl(4, a, SIZE);
    __vr vb = _vel_vbrds_vsl(tol, SIZE);
    __vr vc = _vel_vbrds_vsl(-tol, SIZE);
    __vr vzero = _vel_vbrds_vsl(0, SIZE);
    __vr vone = _vel_vbrds_vsl(1, SIZE);

    __vr vgt = _vel_vfcmps_vvvl(va, vc, SIZE);                    // vgt[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vr vlt = _vel_vfcmps_vvvl(va, vb, SIZE);                    // vlt[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vr vmul = _vel_vfmuls_vvvl(vgt, vlt, SIZE);                 // vmul[i] < 0 se a[i] < tol && a[i] > -tol

    __vm256 mask = _vel_vfmksge_mvl(vmul, SIZE);                  // 1 se vsum[i] >= 0

    __vr accr = _vel_vfmuls_vvvmvl(vone, va, mask, vzero, SIZE);  //accr[i] = mask[i] ? vone[i] * va[i] : vzero[i]

    _vel_vstu_vssl(accr, 4, dst, SIZE);
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vbrds_vsl`, `_vel_vfcmps_vvvl`, `_vel_vfmuls_vvvl`, `_vel_vfmksge_mvl`, `_vel_vfmuls_vvvmvl` and `_vel_vstu_vssl` specifying the size of the elements at 4 bytes."

4.4.6 All zeros

```
inline bool allzeros( const double* a) {
    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vm256 mask = _vel_vfmkdeq_mvl(va, SIZE);                // 1 se va[i]= 0
    unsigned long int ris = _vel_pcvm_sml(mask, SIZE);        // count degli 1
    return ris == SIZE;
}
```

A double pointer `a` is declared, representing the input vector for the function. The function begins by loading the memory elements of vector `a` into the vector register `va`. A vector mask (`__vm256`) created with the `_vel_vfmkdeq_mvl` function is used to check whether each element in `va` is equal to zero. The mask `mask` will have an element set to 1 if the corresponding element in `va` is zero, otherwise it will be set to 0. The `_vel_pcvm_sml` function is used to count the number of elements set to 1 in the mask. This corresponds to the count of zeros present in the `va` vector. The function returns true if the number of zeros in the `va` vector is equal to the total length of the vector, otherwise it returns false.

```
inline bool allzeros( const float* a) {
    __vr va  = _vel_vldu_vssl(4, a, SIZE);
    __vm256 mask = _vel_vfmkseq_mvl(va, SIZE);                // 1 se va[i]= 0
    unsigned long int ris = _vel_pcvm_sml(mask, SIZE);        // count degli 1
    return ris == SIZE;
}
```

The function performs the same operations as the previous one, but it goes to work with float

elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl` and `_vel_vfmkseq_mvl` specifying the size of the elements at 4 bytes.

4.4.7 All equal

```
inline bool allequal(const double* a,const double* b) {
    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vb = _vel_vld_vssl(8, b, SIZE);
    __vr cmp = _vel_vcmpsl_vvvl(va, vb, SIZE);              // cmp[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vm256 mask = _vel_vfmkdeq_mvl(cmp, SIZE);             // 1 se va[i]= 0
    unsigned long int ris = _vel_pcvm_sml(mask, SIZE);      // count 1
    return ris == SIZE;
}
```

Two double pointers `a` and `b` are declared, representing the input vectors for the function. The function begins by loading the memory elements of vectors `a` and `b` into the vector registers `va` and `vb`. A vector comparison is performed between the elements of `va` and `vb` using the `_vel_vcmpsl_vvvl` function. A vector mask (`__vm256`) created with the function `_vel_vfmkdeq_mvl` is used to check whether each element in `cmp` is equal to zero. The function `_vel_pcvm_sml` is used to count the number of elements set to 1 in the mask. This corresponds to the element count where `a`[i] == `b`[i]. The function returns true if the number of elements where `a`[i] == `b`[i] is equal to the total length of the vectors, otherwise it returns false.

```
inline bool allequal(const float* a,const float* b) {
    __vr va  = _vel_vldu_vssl(4, a, SIZE);
    __vr vb  = _vel_vldu_vssl(4, b, SIZE);
    __vr cmp = _vel_vfcmps_vvvl(va, vb, SIZE);              // cmp[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vm256 mask = _vel_vfmkseq_mvl(cmp, SIZE);             // 1 se va[i]= 0
    unsigned long int ris = _vel_pcvm_sml(mask, SIZE);      // count 1
    return ris == SIZE;
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vfcmps_vvvl` and `_vel_vfmkseq_mvl` specifying the size of the elements at 4 bytes.

4.4.8 Compare all

```
inline int cmpall(const double  * a, const double  * b){
    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vb = _vel_vld_vssl(8, b, SIZE);

    __vr cmp = _vel_vcmpsl_vvvl(va, vb, SIZE);                  // cmp[i] = p se a>b ; 0 se a=b ; neg se a<b
    __vm256 mask_g = _vel_vfmkdgt_mvl(cmp, SIZE);               // 1 se va[i] > vb[i]
    __vm256 mask_l = _vel_vfmkdlt_mvl(cmp, SIZE);               // 1 se va[i] < vb[i]

    unsigned long int count_g = _vel_lzvm_sml(mask_g,SIZE);     // conta gli 0 da sinistra
    unsigned long int count_l = _vel_lzvm_sml(mask_l,SIZE);     // conta gli 0 da sinistra

    if(count_g < count_l) return 1;                            // a>b
    if(count_g > count_l) return -1;                           // a<b
    return 0;                                                  // a=b
}
```

Two double pointers `a` and `b` are declared, representing the input vectors for the function. The function begins by loading the memory elements of vectors `a` and `b` into the vector registers `va` and `vb`. A vector comparison is performed between the elements of `va` and `vb` using the `_vel_vcmpsl_vvvl` function. Two vector masks (`__vm256`) are created: `mask_g` to identify elements where `va`[i] $>$ `vb`[i] and `mask_l` to identify elements where `va`[i] $<$ `vb`[i]. Both masks have elements set to 0 where the comparison is false.

The function `_vel_lzvm_sml` is used to count the number of elements set to 0 from the left in the masks `mask_g` and `mask_l`.

The function compares the resulting counts: if `count_g` $<$ `count_l`, it returns 1 indicating that `a` is greater than `b`; if `count_g` $>$ `count_l`, it returns -1 indicating that `a` is less than `b`; if `count_g` $==$ `count_l`, it returns 0 indicating that `a` is equal to `b`.

```
inline int cmpall( const float * a,  const float * b) {
    __vr va = _vel_vldu_vssl(4, a, SIZE);
    __vr vb = _vel_vldu_vssl(4, b, SIZE);

    __vr cmp = _vel_vfcmps_vvvl(va, vb, SIZE);                  // cmp[i] = p se a>b ; 0 se a=b ; neg se a<b

    __vm256 mask_g = _vel_vfmksgt_mvl(cmp, SIZE);               // 1 se va[i] > vb[i]
    __vm256 mask_l = _vel_vfmkslt_mvl(cmp, SIZE);               // 1 se va[i] < vb[i]

    unsigned long int count_g = _vel_lzvm_sml(mask_g,SIZE);     // conta gli 0 da sinistra
    unsigned long int count_l = _vel_lzvm_sml(mask_l,SIZE);     // conta gli 0 da sinistra

    if(count_g < count_l) return 1;
    if(count_g > count_l) return -1;
    return 0;
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vfcmps_vvvl`, `_vel_vfmksgt_mvl` and `_vel_vfmkslt_mvl` specifying the size of the elements at 4 bytes.

4.4.9 Compare zero

```
inline int cmp0(const double* a) {
    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vb = _vel_vbrdd_vsl(0, SIZE);

    __vr cmp = _vel_vcmpsl_vvvl(va, vb, SIZE);                  // cmp[i] = p se a>b ; 0 se a=b ; neg se a<b

    __vm256 mask_g = _vel_vfmkdgt_mvl(cmp, SIZE);               // 0 se va[i] > 0
    __vm256 mask_l = _vel_vfmkdlt_mvl(cmp, SIZE);               // 0 se va[i] < 0

    unsigned long int count_g = _vel_lzvm_sml(mask_g,SIZE);     // conta gli 0 da sinistra
    unsigned long int count_l = _vel_lzvm_sml(mask_l,SIZE);     // conta gli 0 da sinistra

    if(count_g < count_l) return 1;                            // a>0
    if(count_g > count_l) return -1;                           // a<0
    return 0;                                                  // a=0
}
```

A double pointer `a` is declared, representing the input vector for the function. The function begins by loading the memory elements of vector `a` into the vector `va` register. A vector register `vb` is created with all elements set to the value zero using the `_vel_vbrdd_vsl` function. A vector comparison is performed between the elements of `va` and the zero value contained in `vb` using the `_vel_vcmpsl_vvvl` function. Two vector masks (`__vm256`) are created: `mask_g` to identify elements where `va[i] > 0` and `mask_l` to identify elements where `va[i] < 0`. Both masks have elements set to 0 where the comparison is false. The function `_vel_lzvm_sml` is used to count the number of elements set to 0 from the left in masks `mask_g` and `mask_l`. The function compares the counts obtained: if `count_g < count_l`, it returns 1 indicating that `a` is greater than zero; if `count_g > count_l`, it returns -1 indicating that `a` is less than zero; if `count_g == count_l`, it returns 0 indicating that `a` is equal to 0.

```
inline int cmp0(const float* a) {

    __vr va = _vel_vldu_vssl(4, a, SIZE);
    __vr vb = _vel_vbrds_vsl(0, SIZE);

    __vr cmp = _vel_vfcmps_vvvl(va, vb, SIZE);                  // cmp[i] = p se a>b ; 0 se a=b ; neg se a<b

    __vm256 mask_g = _vel_vfmksgt_mvl(cmp, SIZE);               // 0 se va[i] > 0
    __vm256 mask_l = _vel_vfmkslt_mvl(cmp, SIZE);               // 0 se va[i] < 0

    unsigned long int count_g = _vel_lzvm_sml(mask_g,SIZE);     // conta gli 0 da sinistra
    unsigned long int count_l = _vel_lzvm_sml(mask_l,SIZE);     // conta gli 0 da sinistra

    if(count_g < count_l) return 1;                            // a>0
    if(count_g > count_l) return -1;                           // a<0
    return 0;                                                  // a=0
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vbrds_vsl`, `_vel_vfcmps_vvvl`, `_vel_vfmksgt_mvl`, and `_vel_vfmkslt_mvl` specifying the size of the elements at 4 bytes.

4.4.10 Find first non zero

```
inline int findFirstNonZero(const double* a) {
    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vm256 mask = _vel_vfmkdne_mvl(va, SIZE);                  // 0 se va[i] = 0
    return _vel_lzvm_sml(mask,SIZE);                            // conta gli 0 da sinistra
}
```

A double pointer `a` is declared, representing the input vector for the function. The function begins by loading the memory elements of vector `a` into the vector `va` register. A vector mask (`__vm256`) is created using the `_vel_vfmkdne_mvl` function. The mask `mask` will have an element set to 0 if the corresponding element in `va` is zero; otherwise, it will be set to 1. The `_vel_lzvm_sml` function is used to count the number of elements set to 0 from the left in the mask. This corresponds to the position of the leftmost nonzero element in the vector. If all elements in the vector are zero, the function will return the total length of the vector.

```
inline int findFirstNonZero(const float* a) {
    __vr va = _vel_vldu_vssl(4, a, SIZE);
    __vm256 mask = _vel_vfmksne_mvl(va, SIZE);                  // 0 se va[i] = 0
    return _vel_lzvm_sml(mask,SIZE);                            // conta gli 0 da sinistra
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl` and `_vel_vfmksne_mvl`, specifying the size of the elements at 4 bytes.

4.4.11 Convolution

```
inline int convmul(const double* a,const double* b, double* dst) {
    __vr vb = _vel_vld_vssl(8, b, SIZE);
    __vr va = _vel_vld_vssl(8, a, SIZE);
    __vr vab;
    __vr accr = _vel_vbrdd_vsl(0, SIZE);

    for(int i = 0; i < SIZE; i++){
        vab = _vel_vfmuld_vsvl(va[i], vb, SIZE );
        accr = _vel_vfaddd_vvvl(accr, vab, SIZE);
        vb = _vel_vmv_vsvl(-1, vb, SIZE);
    }
    _vel_vst_vssl(accr, 8, dst, SIZE);

    return 0;
}
```

Three double pointers are declared: `a`, `b`, and `dst`. `a` and `b` represent the input vectors for the convolution operation, while `dst` represents the vector in which the result will be saved. The function begins by loading the memory elements of vectors `a` and `b` into the vector registers `va` and `vb`. An accr vector register is created with all elements set to the value zero using the `_vel_vbrdd_vsl` function. This vector register `accr` will be used to accumulate the partial result of the convolution operation. A `for` loop running from `i = 0` to `i < SIZE` is executed. Within the loop, the following operations are performed:

- A multiplication operation is performed between the `va[i]` element of the `va` vector register and the `vb` vector register using the `_vel_vfmuld_vsvl` function. The result of the multiplication is saved in the `vab` vector register.
- The result of the multiplication, contained in the vector register `vab`, is added to the vector register `accr` using the function `_vel_vffaddd_vvvl`. This operation accumulates the partial result of the convolution operation.
- A right-shift operation is performed in the vector register `vb` using the function `_vel_vmv_vsvl`, with a value of -1. This operation has the effect of "shifting" all the elements of `vb` to the right, filling the new position that has become free to the left with zeros.

When the `for` loop is finished, the final result of the convolution operation is contained in the `accr` vector register. The result is then saved in the `dst` vector using the `_vel_vst_vssl` function.

```
inline int convmul(const float* a,const float* b, float* dst) {

    __vr vb = _vel_vldu_vssl(4, b, SIZE);
    __vr vab;
    __vr accr = _vel_vbrds_vsl(0, SIZE);

    for(int i = 0; i < SIZE; i++){
        vab = _vel_vfmuls_vsvl(a[i], vb, SIZE );
        accr = _vel_vfadds_vvvl(accr, vab, SIZE);
        vb = _vel_vmv_vsvl( -1  , vb , SIZE);
    }
    _vel_vstu_vssl(accr, 4, dst, SIZE);

    return 0;
}
```

The function performs the same operations as the previous one, but it goes to work with float elements. The main differences stay in the type of the methods such as `_vel_vldu_vssl`, `_vel_vbrds_vsl`, `_vel_vfmuls_vsvl`, `_vel_vfadds_vvvl`, and `_vel_vstu_vssl`, specifying the size of the elements at 4 bytes.

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. It consists of a bi-objective lexicographic linear programming problem, i.e., an optimization 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, precisely tailored to this type of task.

To make it more realistic, we wrapped the problem within the I-Big-M framework, which adds a third objective to generalize the optimization to the case of unknown starting feasible basis.

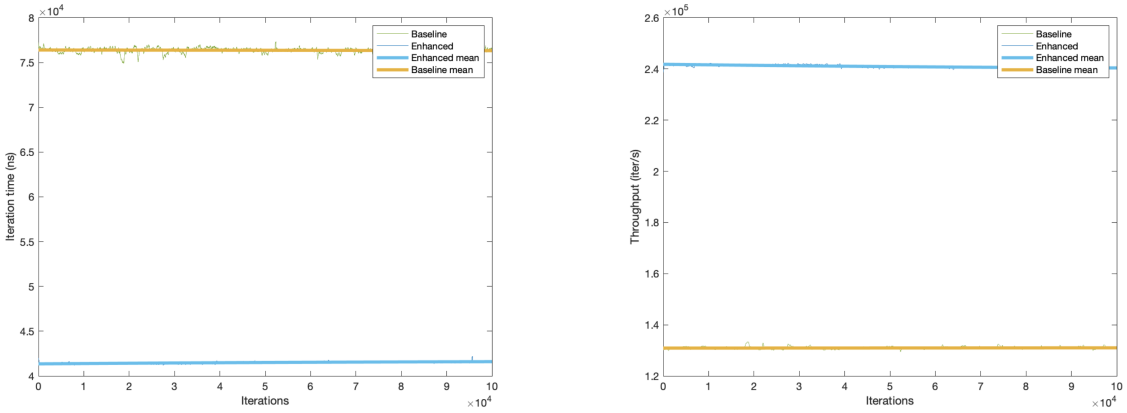


Figure 1: SVE Benchmark

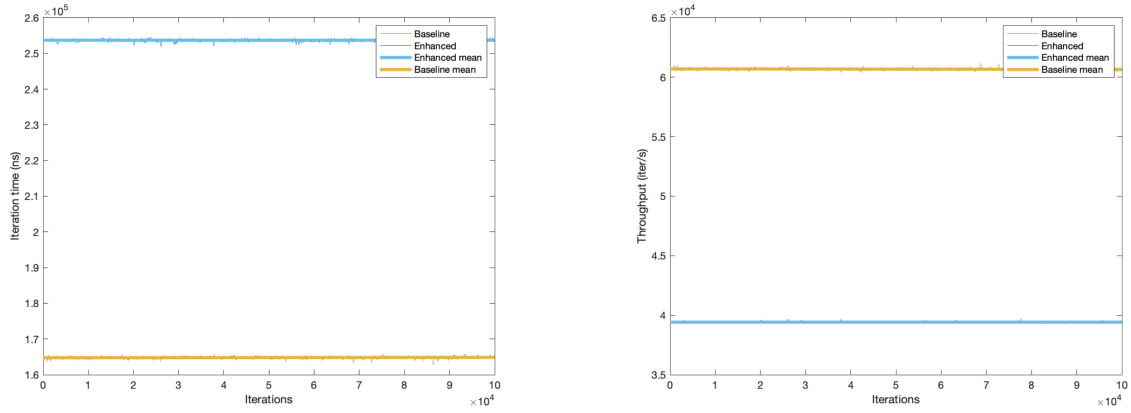


Figure 2: NEC AURORA Benchmark

We ran the benchmark application for 105 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).

Table 1: SVE

	TIME (ns, x10^4)	THROUGHPUT (iter/s, x10^5)
Baseline (non-vector)	70.0841 ± 0.024	0.1411 ± 0.0034
Baseline (vector)	7.6370 ± 0.2413	1.3094 ± 0.0386
Enhanced	4.1496 ± 0.0971	2.4099 ± 0.0506

Table 2: NEC aurora

	TIME (ns, x10^5)	THROUGHPUT iter/s, x10^4)
Baseline (non-vector)	10.0585 ± 0.0380	0.9447 ± 0.0312
Baseline (vector)	1.6480 ± 0.0290	6.0679 ± 0.0386
Enhanced	2.5369 ± 0.0309	3.9418 ± 0.101

The benchmark was run on an SVE and NEC Aurora processor with 8 64-bit BAN coefficients. Figure shows the comparison between the two versions in terms of average time taken per iteration and overall throughput (iterations per second). The mean value and standard deviation of the two metrics are shown in Table.

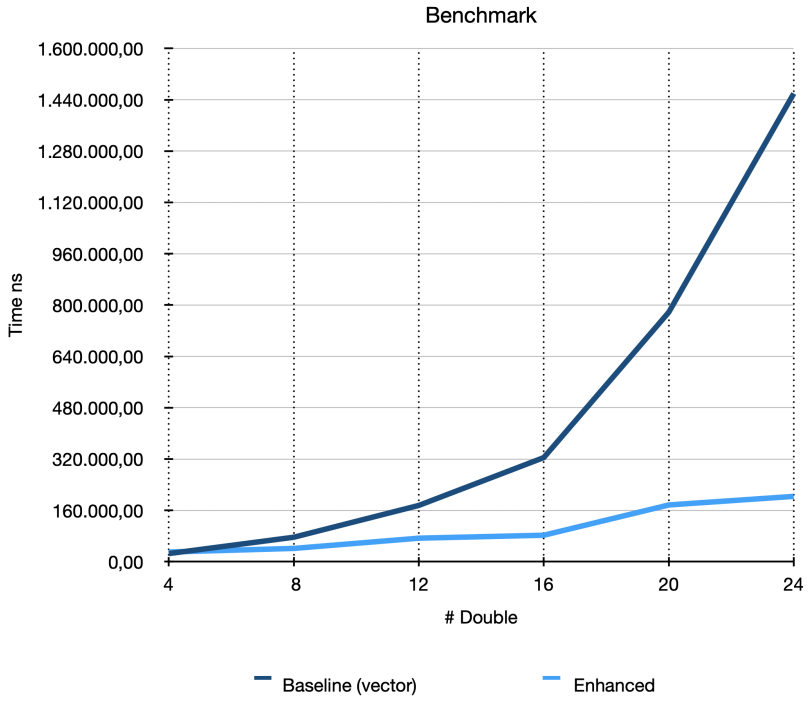
```
α^0(910 + 940η^1 + 0η^2 + 0η^3 + 0η^4 + 0η^5 + 0η^6 + 0η^7
0
```

The correctness of the evaluated operations was done by comparing the result at the end of execution with the result shown above.

5.1 SVE: Comparison as the number of BAN coefficients increases

An additional test was performed to evaluate the trend of the execution time of the algorithms when the number of doubles per BAN changes.

double	Baseline (vector)	**Enhanced**
4	25395.12	30209.49
8	76370.64	41496.15
12	175472.78	73421.44
16	324588.19	82591.67
20	777556.11	176988.35
24	1459517.31	203698.10



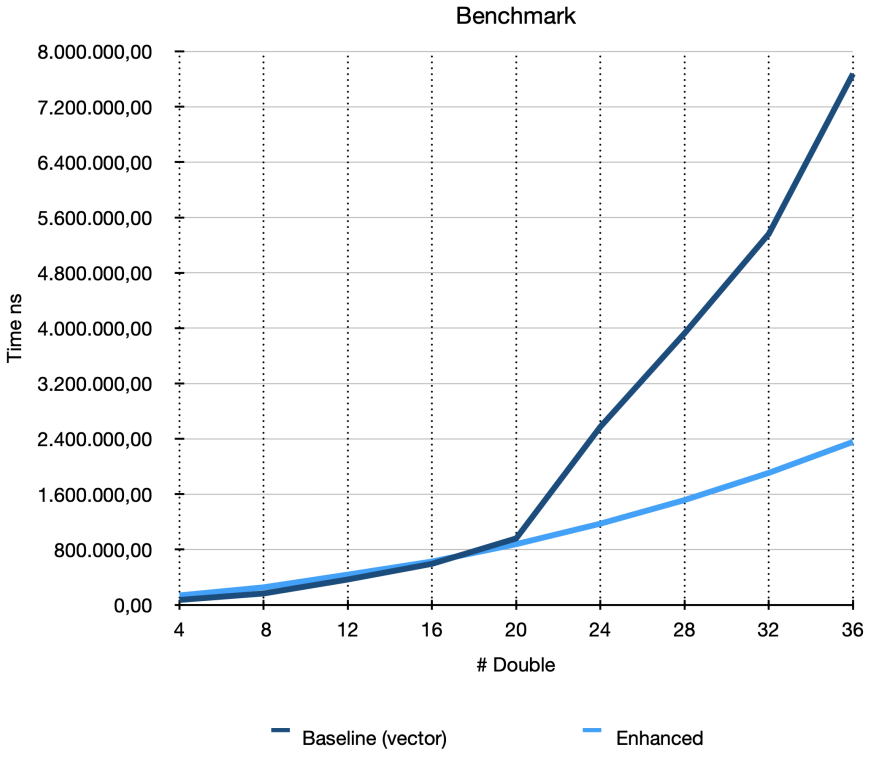
- As the number of doubles per BAN increases, both the "Baseline" and "Enhanced" algorithms show an increase in execution time. This indicates that both algorithms deal with increased computational complexity when the data size (number of doubles per BAN) increases.
- The "Baseline" algorithm has better times than the "Enhanced" algorithm only for BANs with 4 doubles. However, as BAN size increases, the "Baseline" algorithm shows an exponential increase in execution time, while the "Enhanced" algorithm has significantly better execution times.

- The "Enhanced" algorithm shows better scalability and performance as BAN size increases. This suggests that in the "Enhanced" algorithm, optimization techniques make it more efficient with larger data sizes.

5.2 NEC: Comparison as the number of BAN coefficients increases

An additional test was performed to evaluate the trend of the execution time of the algorithms when the number of doubles per BAN changes.

double	Baseline (vector)	**Enhanced**
4 double	71.345,00	136.455,00
8 double	164.633,17	253.502,51
12 double	366.915,00	437.185,00
16 double	593.325,00	626.230,00
20 double	960.807,41	877.655,00
24 double	2.574.733,67	1.174.005,03
28 double	3.926.648,24	1.515.165,83
32 double	5.362.231,16	1.907.728,64
36 double	7.676.562,81	2.355.824,12



- Looking at the data in the table, we can see that for BANs with sizes less than 16 doubles, the "Baseline" algorithm performs better than the "Enhanced" algorithm. However, from BANs with 16 doubles and above, the "Enhanced" algorithm shows a performance advantage, with lower execution times than the "Baseline".
- As the size of BANs increases, the "Baseline" algorithm shows a significantly higher increase in execution time than "Enhanced" which has a slower growth.
- The reason why the "Enhanced" algorithm has an advantage from larger BANs may be related to its ability to exploit the full power of NEC Aurora. The NEC Aurora processor has very large vector registers, capable of holding up to 256 doubles (up to 2048KB total). This vector memory capacity allows the "Enhanced" algorithm to perform operations on large numbers of data in parallel, thus achieving better performance when the number of doubles per BAN reaches a size that allows the vector registers to be fully filled.
- The choice of using the "Enhanced" versus "Baseline" algorithm may depend on the size of the BANs and the ability of the processor to take advantage of the vector processing power. For larger BAN sizes, the "Enhanced" algorithm offers better performance by taking full advantage of NEC Aurora's features.

6 Conclusion

In this project, we presented the acceleration of a C++ library for Bounded Algorithmic Numbers (BAN) by exploiting vector instructions. We also presented a non-Archimedean optimization benchmark with numerous iterations, used to evaluate the goodness of vector acceleration with respect to the BAN library.

The use of intrisic functions for both architectures leads to significant performance improvement when BANs with a high number of coefficients are present.

In conclusion, the vector optimization approach presented in our work can be of great value to improve the performance of the BAN library and enable more efficient manipulation of Bounded Algorithmic Numbers in numerous vectorization applications.

7 References

References

[1] Lorenzo Fiaschi, Federico Rossi, Marco Cococcioni, and Sergio Saponara, "Speeding Non-Archimedean Numerical Computations up using AVX-512 SIMD Instructions.

[2] ARM, "Introduction to SVE",
https://developer.arm.com/documentation/

[3] NEC CORPORATION, "SX-Aurora TSUBASA Architecture Guide",
https://sxauroratsubasa.sakura.ne.jp/documents/sdk/pdfs/g2af01e-C++UsersGuide-031.pdf

[4] SX-Aurora TSUBASA Architecture,
https://sxauroratsubasa.sakura.ne.jp/documents/guide/pdfs

[5] Sx-aurora-dev, github velintrin.h,
https://sx-aurora-dev.github.io/velintrin.html

[6] Arm Instruction-sets intrinsic,
https://developer.arm.com/architectures/instruction-sets/intrinsic.

[7] SVE Optimization Guide,
https://developer.arm.com/documentation/102699/latest/

[8] Arm C Language Extensions for SVE,
https://developer.arm.com/documentation/100987/latest/

[9] Auto-vectorization and Helium,
https://developer.arm.com/documentation/102095/0101/Auto-vectorization-and-Helium