# Portability across Arm NEON and SVE vector instruction sets using the NSIMD library: a case study on a Seismic Spectral-Element kernel

Sylvain Jubertie, Kenny Peou, Guillaume Quintin, Fabrice Dupros

# Portability across Arm NEON and SVE vector instruction sets using the NSIMD library: a case study on a Seismic Spectral-Element kernel

Sylvain Jubertie
*Univ. d'Orléans, INSA CVL*
*LIFO EA 4022*
Orléans, France
sylvain.jubertie@univ-orleans.fr

Kenny Peou
*Agenium Scale*
Gif-sur-Yvette, France
kenny.peou@agenium.com

Guillaume Quintin
*Agenium Scale*
Gif-sur-Yvette, France
guillaume.quintin@agenium.com

Fabrice Dupros
*Arm*
Sophia-Antipolis, France
fabrice.dupros@arm.com

***Abstract*—The SIMD paradigm (Single Instruction on Multiple Data) is implemented in almost all today's processors in the form of units with large registers and arithmetic execution units operating on them. Traditional SIMD instruction sets like SSE, AVX and AVX-512 on x86 architectures or NEON on Arm architectures have fixed size register widths (or vector lengths): 128-bit for SSE and NEON, 256-bit for AVX, and 512-bit for AVX-512. To vectorize a code, the easiest solution is to rely on automatic vectorization provided by compilers. Porting a code to a different SIMD unit only requires to recompile it. However, in many cases, like the Seismic Spectral-Element kernel considered in this study, compilers are not able to efficiently vectorize the code and developers may consider using compiler intrinsics for explicit vectorization. In this case, the developer has to rewrite his code for each targeted SIMD instruction set, which hinders maintainability and portability.**

**The Arm SVE (Scalable Vector Extension) SIMD instruction set brings a different programming model called Vector-Length Agnostic (VLA) to address these limitations. It allows to decouple the vectorized code from the underlying SVE implementation. An SVE implementation may have 128-bit or n$\times$128-bit registers, up to 2048-bit. The same SVE instruction may operate on a register independently of its length. Thus the same code will run on all SVE architectures without recompilation.**

**In this paper, we study the vectorization of a Seismic Spectral-Element kernel using NEON and SVE intrinsics to illustrate the differences and the difficulties of porting a code to the VLA programming model. Several libraries address the problem of portability between fixed-size SIMD instruction sets but depend on the knowledge of the vector length at compile time and thus do not support VLA instruction sets like SVE. We propose to use the NSIMD library, which supports both paradigms, to create a single vectorized code for both instruction sets.**

***Keywords*—*SIMD;NEON/SVE;SEM.***

## I. Introduction

SIMD units are available on almost all current processors under several different names: SSE, AVX and AVX-512 on x86 architecture, NEON and SVE [1] on Arm, VMX and VSX on Power. Each of these units has different instruction sets operating on different numbers of registers with different vector lengths. SSE, NEON, VMX and VSX have 128-bit registers, but SSE has only 16 registers on x86-64 (only 8 on 32-bit x86), NEON 32 registers on AARCH64 (only 16 on AARCH32), VMX 32 and VSX 64 registers. On the same architecture, processors with a given SIMD instruction set are backward compatible with previous SIMD instruction sets. For example, a processor with AVX-512 also supports SSE and AVX instruction sets. Thus, a binary compiled for a given SIMD instruction set is not able to take advantage of a more recent one. The SVE instruction set is designed to solve this portability problem across Arm architectures. The main idea is to decouple the instruction set from the vector length. This approach is called VLA which stands for Vector-Length Agnostic. Depending on the performance requirement for a given platform, for example embedded or HPC, it is possible to implement SVE with 128-bit registers, or a multiple of 128 up to 2048-bit registers. The instruction set and the number of registers (32 general + 16 predicate registers) remains the same. Note that the vector length may be also decoupled from the length of the execution units (ALUs and FPUs). For example, a processor implementing a 512-bit SVE unit will have 512-bit registers but may only have 128-bit execution units operating sequentially on each quarter of the register. Using 128-bit SIMD execution units, we may expect a theoretical speedup of 4 when operating on 32-bit values over a scalar code. Of course, to benefit the most from these units, codes have to exhibit SIMD-friendly patterns and also be compute bound.

To take advantage of these SIMD units without having to deal with all the different flavors of SIMD units, the simplest solution for the developer may be to rely on the compiler to automatically vectorize his code. It is also possible to help the compiler identify vectorization opportunities by adding some OpenMP pragmas (since OpenMP 4.0) in front of loops. However, in most cases compilers are not able to vectorize or to efficiently vectorize codes. In [2], [3], we study some codes like image processing algorithms, vector normalization, nbody simulation, stencil computation and show that even if some compilers are able to perform automatic vectorization, the performance is below the one obtained from explicit vectorization using intrinsics, which are specific functions automatically translated to SIMD instructions by compilers.

Another highlighted problem is the lack of portability across architectures: the same version of a compiler may be able to vectorize a code for AVX but not for NEON. Thus, to ensure the best performance level on each architecture, the best option is to use intrinsics but at the price of losing portability and a higher development effort.

Several higher level abstractions over intrinsics were developed, like Vc [4], VCL [5], Boost.SIMD [6], to provide both performance and portability to vectorized codes. These libraries are further detailed in the Related Work section. While they enable support of various SIMD instruction sets on different platforms, these abstractions require to set the vector length when writing the code or at compile time. For SVE, the vector length is only known at runtime which complexifies the building of a common abstraction for all architectures.

In this paper, we study the explicit vectorization of a Seismic Spectral-Element kernel, known not to be efficiently vectorized by compilers, using NEON and SVE to evaluate the complexity of both fixed length and VLA programming models. We also consider a version based on the NSIMD library, which provides a common abstraction for both paradigms, to validate its approach and its efficiency compared with previous NEON and SVE versions. Since SVE architectures are not currently widely available, we compare the generated assembly codes to determine the achieved level of performance.

## II. Related work

Automatic vectorization is an active research domain since the advent of the first vector processors in Cray computers and the need to vectorize Fortran codes [7], [8]. Today, compilers contain some optimization passes to automatically vectorize codes by analyzing instructions and data access patterns. For example, the Clang compiler contains two vectorizers: the Loop Vectorizer and the SLP [9](Superword-Level Parallelism) Vectorizer. Loop vectorization works by analyzing dependencies between iterations and fuse scalar instructions into vector ones. It requires well formed loop structures and a simple control flow. SLP vectorization is more general, in that it is not limited to loops, and consequently more complex. It analyzes instructions in the compiler's intermediate representation code and tries to gather them into vector ones. This approach may work well on simple codes but strongly depends on the way the code is written and is not able to scale as the code complexity grows. Some improvements are proposed to enhance loop and SLP vectorization [10], [11], often at the price of increased compilation times. In [10], the authors show that a speedup of two is achievable over the performance of the Clang vectorizer, but on very few benchmarks, with a geometric mean speedup of 1.06 on all benchmarks performed, which may be not significant enough compared to performance of hand-vectorized versions of the codes. Previous evaluation of vectorizing compilers [12]–[14], on several benchmarks, show that compilers are not able to bring as much performance as hand-vectorized codes. In [2], on different benchmarks, several architectures and up to date compilers, we also observe the same results. More importantly, we observe that using the same compiler version on different architectures may vectorize some codes on one but not on the others and thus does not guarantee portability of the vectorization.

Using SIMD intrinsics or even assembly codes may guarantee the best level of vectorization but requires writing specific codes for each targeted SIMD instruction set. Several highly optimized libraries like OpenBLAS, FFTW, MKL, use this approach. In [15], we already have used AVX2 and AVX-512 intrinsics to show the performance benefit of such an approach. However, programming using SIMD intrinsics is tedious, not portable, and the amount of code to maintain increases as new SIMD instruction sets arise, thus motivating this present work. Another aspect motivating our work is the impact of vectorization on energy consumption. In [16], we show that using NEON units on ARM cores demands more instantaneous power, but this increase is not significant compared to the gain obtained by reducing the execution time.

A common approach to alleviate the writing of hand-vectorized codes is to hide the diversity of intrinsics dialects inside an abstraction library providing a unique interface for all SIMD instruction sets. Many SIMD abstraction libraries are available, some of them are presented in table I. At the time of writing this paper, the NSIMD library is the only one to support the SVE instruction set when used in combination with the GCC 10 or Armclang compilers which are the only ones to provide access to SVE intrinsics. A limitation for the Eigen, VCL, simdpp and T-SIMD libraries is the requirement of setting the vector length explicitly when writing the code. This model is not compatible with the VLA model promoted by the SVE instruction set. However, the GCC compiler allows to set the SVE vector length at compile time with the flag `-msve-vector-bits`, but the generated code is no more portable across SVE architectures with different vector lengths. The main limitation for all these libraries is that they are based on encapsulating low level vector types (like `__m128` for SSE registers containing float values) into structures. The SVE vector types are indeed sizeless since their length is not known at compile time, which is not permitted by C and C++ standards but is supported as an extension by the Armclang compiler. Thus, SVE vector types cannot be used as structure or class members, except when using this specific compiler. NSIMD is compatible with this extension when compiling with Armclang or allows to set the length of SVE vectors at compile time when using GCC.

## III. The NSIMD library

NSIMD is a vectorization library that abstracts SIMD programming. It was designed to ease the programming of SIMD units found in all commercial processors, but often underutilized, and thus to bring better performance at a lower development cost.

### A. Towards zero-cost abstraction

To achieve maximum performance, NSIMD mainly relies on the inline optimization pass of compilers which is activated by default with optimization flags like `-O2` and above. Therefore,

TABLE I: List of several SIMD abstraction libraries and their specificities.

| General Information | | | Instruction Set | | | | Data Type | | | | | | | Features | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | | License | AVX-512 512-bit | NEON 128-bit | SVE∗ | Altivec 128-bit | Float 64 | Float 32 | Float 16 | Integer 64 | Integer 32 | Integer 16 | Integer 8 | C++ Technique | Register Size |
| Eigen | [17] | MPL2 | Y | Y | N | Y | Y | Y | Y | Y | Y | Y | Y | Op. Overload | Explicit |
| MIPP | [18] | MIT | Y | Y | N | N | Y | Y | N | Y | Y | Y | Y | Op. Overload | Implicit |
| VCL | [5] | Apache-2 | Y | N | N | N | Y | Y | N | Y | Y | Y | Y | Op. Overload | Explicit |
| simdpp | [19] | Boost | Y | Y | N | Y | Y | Y | N | Y | Y | Y | Y | Exp. Template | Explicit |
| T-SIMD | [20] | Open-Source | N | Y | N | N | N | Y | N | N | Y | Y | Y | Op. Overload | Explicit |
| Vc | [4] | BSD-3-Clause | N | N | N | N | Y | Y | N | Y | Y | Y | N | Op. Overload | Implicit |
| boost.SIMD | [6] | Boost | P | N | N | N | Y | Y | N | Y | Y | Y | Y | Exp. Template | Implicit |
| bsimd | [21] | Non-free | P | Y | N | Y | Y | Y | N | Y | Y | Y | Y | Exp. Template | Implicit |
| xsimd | [22] | BSD-3-Clause | Y | Y | N | N | Y | Y | N | Y | Y | N | N | Op. Overload | Implicit |
| NSIMD | [23] | MIT | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Op. Overload | Implicit |

∗ 128-bit multiples up to 2048
† Implementation dependent

using any mainstream compiler such as GCC, Clang, MSVC, XL C/C++, ICC and others with NSIMD gives a zero-cost SIMD abstraction library. To allow inlining, most of the code is placed into header files. Small functions such as addition, multiplication, square root, etc, are all present in header files whereas bigger functions such as I/O are put in source files that are compiled as a dynamic library.

### B. Programming APIs

NSIMD provides C89, C++98, C++11 and C++14 APIs. These different APIs are proposed to ease the code modernization of old industrial codes relying on previous C and C++ standards and compilers, as well as for developing modern codes based on the latest standards and compilers. All APIs allow writing generic code. For the C API this is achieved through a thin layer of macros, while for the C++ APIs it is achieved by using templates and function overloading. The C++ API is split in two. The first part is a C-like API with only function calls and direct type definitions for SIMD types, while the second one is more advanced and provides operator overloading, higher level type definitions that allows unrolling. C++11, C++14 APIs add for instance templated type definitions and templated constants. We show below the same short code example, consisting in the addition of two vectors of 8-bit integers, using the three APIs.

*1) C API:* The C API provides short aliases for all supported data types of the form `{i,u,f}{8,16,32,64}`(f8 is not a valid combination). It then provides aliases for all corresponding SIMD registers by prepending a `v` to the type, or `vl` if the register contains logical values. Functions take the form `nsimd_[Function]_[Extension]_[type]`. This leads to the following code:

```
vi8 r1 = nsimd_loadu_avx2_i8( &p1[i] );
r1 = nsimd_add_avx2_i8( r1 , r1 );
nsimd_storeu_avx2_i8( &p2[i] , r1 );
```

Note that, the SIMD extension and the type need to be included in the function name since the C language prior C11 does not provide function overloading mechanism.

*2) C++ API:* The C++ API relies on C types, but uses tag dispatching to simplify function calls. Functions take the form `nsimd::[Function]( [args] , type() )`. The last parameter is used by the compiler to determine which function to call. This leads to the following code:

```
vi8 r1 = nsimd::loadu( &p1[i] , i8() );
r1 = nsimd::add( r1 , r1 , i8() );
nsimd::storeu( &p2[i] , r1 , i8() );
```

The SIMD instruction set is defined at compile time by a specific flag.

*3) C++ Advanced API:* The C++ API introduces the `pack` type, which takes the form `nsimd::pack<[Type]>`. Template type deduction allows functions to be completely transparent (except for loading data). The `pack` implements operator overloading to further simplify development. This leads to the following code:

```
nsimd::pack<int8_t> r1 = nsimd::loadu<
    nsimd::pack<int8_t>>( &p1[i] );
r1 = r1 + r1;
nsimd::storeu( &p2[i] , r1 );
```

In addition, the `pack` also allows for automatic loop unrolling. When declaring it as `nsimd::Pack<[Type],[Factor]>`, it will automatically unroll loops by `Factor`.

## IV. Seismic wave numerical kernel

Physics-based three-dimensional numerical simulations are becoming more predictive and have already become essential in geosciences. In geophysics, simulations at scale with a very fine resolution, including uncertainty quantification procedures are crucial to provide the relevant physical parameters for forward modeling of seismic wave propagation. The forward wave propagation problem is governed by the elastodynamic equations of motion:

$$\rho \ddot{u}_i = f_i + \tau_{ij,j}, \tag{1}$$

where $\rho$ is the material density; $\ddot{u}_i$ is the $i$-th component of the second time-derivative of the displacement $u_i$; $\tau_{ij,j}$ is the spatial derivative of the stress tensor component $\tau_{ij}$ with respect to $x_j$; $f_i$ is the $i$-th component of the body force.
From the numerical point of view, several methods have been

successfully used for the simulation of elastic wave propagation in three-dimensional domains. Finite-difference methods (FDM), classical or spectral finite-element methods (FEM and SEM) have been introduced last few years for this class of problems. Interested readers could refer to [24] for further details on these approaches. Due to its numerical efficiency, the spectral finite-element method (SEM) is routinely used for seismic simulations ( [25]). As an example, SPECFEM3D is a major software package dedicated to seismic wave modeling. The code implements the SEM leading to breakthroughs in computational geophysics on multi-petascale systems ( [26]). The weak form of eq. 1 is expressed in eq. 2, and is solved in its discretized form by using this method.

$$\int_\Omega \rho \mathbf{v}^T \cdot \ddot{\mathbf{u}}\, d\Omega = \int_\Omega \boldsymbol{\epsilon}(\mathbf{v})^T : \boldsymbol{\tau}\, d\Omega - \int_\Omega \mathbf{v}^T \cdot \mathbf{f}\, d\Omega - \int_\Gamma \mathbf{v}^T \cdot \mathbf{T}\, d\Gamma \tag{2}$$

where $\Omega$ and $\Gamma$ are the volume and the surface area of the domain under study, respectively; $\boldsymbol{\epsilon}$ is the virtual strain tensor related to the virtual displacement vector $\mathbf{v}$; $\mathbf{f}$ is the body force vector and $\mathbf{T}$ is the traction vector acting on $\Gamma$. Superscript $T$ denotes the transpose, and a colon denotes the contracted tensor product.

As reported in [26]–[28], explicit parallel elastodynamics application usually exhibits very good weak and strong scaling up to several tens of thousands of cores. But if we focus on SIMD-level optimizations, the compiler performance plays a major role and recent studies have underlined the limited impact of automatic vectorization [3], [29]. Most of the time, the complexity of the computational workflows limit the efficacy of automatic optimization mechanisms. For the Spectral-element method, it is admitted that the summation of the element contributions (assembly phase) represents a major bottleneck. This is coming both from the shared values between neighboring elements and the inherent indirection for data accesses induced by this approach.

The complexity of the internal loop might also be a blocker for the compiler. Despite representing as much as 80% of the total elapsed time, only hand-tuned implementations have been able to fully vectorize the computation of internal forces.

## V. Vectorization

In this section, we first present the scalar version of our kernel. Then, we describe the approach we use to vectorize the code independently of the targeted SIMD instruction set. We give details on our NEON and SVE implementations using intrinsics and discuss the differences between them. Finally, we present our implementation based on the NSIMD library.

### A. Scalar version

The kernel is composed of four loop levels. The outermost one iterates over elements. For each element, we perform the three following steps: 1) gathering points of the element, 2) computing internal forces, 3) assembly step: element contribution is updated. Each of these steps consists of three nested loops to iterate along the three directions of each element.

Gathering points for each element consists in copying values from a global array to a local array following an indirection array. These indirections are required since element borders overlap: one point may be shared by up to eight elements (corners). Points are not replicated in order to reduce memory footprint and avoid multiple updates at step 3. The local array is relatively small and is likely to fit and stay into the L1 cache along the two other steps. For example, at order 4, the element consists of a block of 5x5x5 points, each point containing three floating point values, thus a total of 1.5kB per element.

Internal forces are computed by traversing the element along the three directions, multiplying point values by Lagrangian coefficients and accumulating the results. Thus, it essentially consists of additions and multiplications which are likely to be merged into FMA (Fused Multiply Add) instructions.

Finally, results are to be stored back to the global array with the same indirection pattern as for the gathering step.

### B. Vectorization approach

A common approach suited for vectorizing most algorithms is to vectorize the innermost loop. For our kernel, the number of iterations for the three inner loops depends on the considered order. At order 4, the innermost loop only performs five iterations, which is enough to feed 128-bit NEON units, but not enough to feed larger 512-bit SVE units. It is possible to fuse the inner loops to bring more vectorization opportunities. However, this approach has several drawbacks:

1) it does not exactly match the register length in the general case and thus still requires some scalar (NEON) or masked (SVE) instructions
2) it requires irregular accesses to data since data in the same register may not be contiguous following the element structure
3) it breaks the code organization making it harder for the developer to read/modify/maintain
4) it is not possible to write a single generic code independent from the considered order and from the SIMD unit (SVE) width

A much simpler approach consists of vectorizing the outermost loop, thus applying the same computation over several elements at the same time. It allows us to solve the above problems. The SIMD unit width determines the number of elements considered at the same time. With the 128-bit NEON instruction set, we can store four single precision floating point values into a single register, thus we can process four elements at the same time. With the vector length agnostic SVE instruction set, we can process `len` elements, where `len` is known dynamically and may be a multiple of 4 between 4 (128-bit) and 64 (2048-bit). Only a few remaining elements may be computed the scalar way with NEON (up to three), thus requiring to keep a scalar version of the code after the vectorized one. With masked instructions provided by SVE, the remaining elements are processed with the same code. Another advantage of this approach is that the code organization stays the same. Instead of considering scalar floating point values, we consider vectors of floating point

values, and arithmetic instructions are exactly the same but are applied on these vectors.

This approach is implemented using NEON and SVE as described hereafter. However, these two instructions sets rely on two different programming models which leads to two very different implementations.

### C. Implementation using NEON intrinsics

Following our approach, porting the kernel to NEON is relatively straightforward. First, we replace the `float` data type by the `float32x4_t` one for the definition of the local array. For the outer loop, we compute four elements at the same time, thus we increment the iterator by four instead of one. Since no gather instruction is available in NEON for gathering a vector of data from the global array and the indirection array, the gathering step just consists of filling the local array using scalar loads but for four times more elements. The vectorization of the computation of internal forces is quite straightforward thanks to the automatic type inference provided by the C++ `auto` keyword and to the overload of arithmetic operators for NEON data types. Thus, the code organization for the NEON version is almost identical to the scalar version minus few modifications described hereafter.

*1) Local variables:* The `float` type of local variables needs to be replaced by the `float32x4_t` type, for example the local array definition at order 4:

```
float rl_displacement_gll[125*3]
```

becomes:

```
float32x4_t rl_displacement_gll[125*3];
```

Most local variables are used to store temporary results, so we can use the `auto` keyword to infer their types.

*2) Gather/scatter:* Non-contiguous loads required by the vectorization and the data storage pattern have to be replaced by four scalar loads, for example:

```
float32x4_t dxidx;
dxidx[ 0 ] = rg_hexa_gll_dxidx[ id0 ];
dxidx[ 1 ] = rg_hexa_gll_dxidx[ id1 ];
dxidx[ 2 ] = rg_hexa_gll_dxidx[ id2 ];
dxidx[ 3 ] = rg_hexa_gll_dxidx[ id3 ];
```

*3) Broadcast:* When the same coefficient has to be applied to all the values in the vector, we use the `vdupq_n_f32` intrinsics to explicitly duplicate the coefficient. For example, the following scalar code:

```
auto tauxx=trace_tau+2.0f*rhovs2*duxdx;
```

is replaced by:

```
auto tauxx=trace_tau
          +vdupq_n_f32(2.0f)*rhovs2*duxdx;
```

Arithmetic operators between vectors are overloaded by compilers but not between vectors and scalar values.

*4) Arithmetic instructions:* In most cases, arithmetic expressions are not modified, for example, the expression:

```
auto duxdx = duxdxi*dxidx
           +duxdet*detdx
           +duxdze*dzedx;
```

is automatically converted by the compiler to its vectorized version since variables are vectors instead of floats and arithmetic operators are overloaded by default. It is also possible to overload operators between scalars and vectors to remove explicit calls to the `vdupq_n_f32` intrinsics. Note that, Fused Multiply Add (FMA) intrinsics are available in NEON but it is not mandatory to explicitly use them since compilers are able to generate them by analyzing arithmetic expressions. This allows to keep the syntax of the vectorized code as close as possible to the original one.

### D. SVE implementation

The SVE approach for vectorization is quite different since the vector length is not known at compile time but available at runtime through the use of the `svcntw` intrinsics which returns the number of 32-bit words in one SVE register. It has the advantage of generating a vector length agnostic code which does not require to recompile it for different SVE architectures. Other advantages over NEON are the availability of gather/scatter instructions and of masked load/-store instructions with a supplemental mask parameter which allows to partially fill/store vectors. However, implementing the SVE version is not as straightforward as expected for the following reasons: 1) it is not possible to create local arrays of vectors since their size is not known at compile time, 2) arithmetic operators are not overloaded by default since intrinsics require a mask, thus increasing the code verbosity. Our SVE implementation requires to modify our scalar code as described below.

*1) Local variables and arrays:* The definition of the local array becomes:

```
float rl_displacement_gll[125*3*svcntw()]
```

where the `svnctw` intrinsics returns the number of float values in SVE vectors. Indeed, both Armclang and GCC compilers do not support defining an array of vectors of type svfloat32_t on the stack since the size of this type is not known at compile time. It is possible to specify the vector length with a specific flag for the GCC compiler but it limits the code portability.

*2) Loop management:* The main loop increment depends on the vector length:

```
for(iel=elt_start;...;iel+=svcntw())
```

Since the number of elements may not be a multiple of the SVE vector length, we need to use a mask to enable only active lanes.

*3) Masking:* We define the mask for SVE intrinsics according to the number of elements for each iteration as:

```
auto mask=svwhilelt_b32_u32(iel,end);
```

In our case, the mask enables all the vector lanes for all iterations except the last one if the number of elements is not a multiple of the vector length. Note that, in our case, the mask is only mandatory for gather/scatter intrinsics to avoid segmentation faults caused by out of bound memory accesses. Arithmetic intrinsics may operate on all vector lanes since values in each lane are computed independently i.e. computation on one lane does not require to access values from other lanes. Thus, for these intrinsics we have the choice between passing the mask we previously defined or using the `svptrue_b32()` intrinsics which enables all lanes.

*4) Gather/scatter:* Unlike NEON, SVE provides a set of gather/scatter intrinsics to retrieve indexed data. For the gathering step of our kernel, we first need to gather indices of the first point of each element considered. At order 4, each element is composed of 125 points. If we consider the first elements processed at the first iteration, the index of the first point of the first element is at position 0, the one for the second element is at position 125, for the third element at position 250 and so on. SVE provides the `svindex_u32()` intrinsics to fill a vector with multiples of a given value. In our case, we use the following code:

```
auto vstrides = svindex_u32(0u, 125u);
```

At iteration `i`, the first element of the vector is at position `i*svcntw()*125` and we need to duplicate it and add the `vstrides` values to obtain the indices of the first point of each element in the vector:

```
auto base=svadd_z(mask,
                  vstrides,
                  iel*125u));
```

The indices in this resulting vector will serve as the base for accessing the following 124 points of these elements.

*5) Arithmetic instructions:* Since the mask parameter is not optional for arithmetic intrinsics, there is no overloaded operator available by default, which increases the code verbosity. In our case, we have the choice between using the mask computed at each iteration and already used for gather/scatter instructions or we can use the `svptrue_b32()` intrinsics which activates all vector lanes.

### E. NSIMD implementation

Deriving the kernel for NSIMD is quite straightforward from NEON and SVE intrinsics versions.

*1) Retrieving vector length:* Since the VLA paradigm imposes to obtain the vector length at runtime, we need to explicitly use the `nsimd::len` function which in turn calls the `svcntw` intrinsics to dynamically retrieve the vector length. When compiled for a fixed length vector architecture, the call is replaced by the vector length at compile time. The vector length is then used to allocate local arrays and determine the increment for the loop iterating over elements.

*2) Gather/scatter:* NSIMD gather and scatter functions take the same arguments as the corresponding SVE intrinsics and basically wrap them. Note that this is also the case for their AVX-512 equivalents. When compiled for a fixed size architecture which does not support these intrinsics, like NEON, calls are replaced by scalar loads and stores.

*3) Arithmetic operators:* NSIMD provides overloaded operators for logical and arithmetic instructions. Since SVE instructions require a mask parameter, it is set by default to `svptrue_b32()`, for instructions operating on 32-bit data types, to apply instructions on all the vector lanes.

*4) Compiler limitations/future improvements:* The main limitation to improve the abstraction and further simplify the writing of NSIMD codes is the missing support for creating local arrays of SVE vectors in current compilers. The following declaration is indeed not valid:

```
svfloat32_t array[10];
```

and must be replaced by:

```
float array[10*svcntw()];
```

This prevent the use of the = operator for transparently loading/storing vectors from/to memory. When using the = operator with fixed length instruction sets, we observe that compilers are able to automatically generate both load and store instructions. We believe that this limitation will be removed in future compilers since the first declaration may be automatically replaced by the second one. The GCC compiler allows to set the SVE vector length at compile time, thus allowing to use the first declaration, but requires the code to be recompiled in order to run on different SVE vector lengths. If developers target a specific SVE vector length this is currently the only option to further simplify the NSIMD version.

## VI. Experimental results

In this section, we discuss the following versions of our kernel: 1) scalar with autovectorization, 2) intrinsics NEON and SVE, 3) NSIMD. Note that, since SVE architectures are not widely available, we are only able to discuss the performance of NEON versions. For SVE versions, we propose to evaluate the ratio of vectorized instructions between the autovectorized, the intrinsics and the NSIMD versions. To verify that NSIMD offers similar performance as using intrinsics, we propose to compare the generated assembly codes.

### A. Experimental setup

For these experiments, we use GCC (v10.0.1) and Armclang (v20.0) LLVM-based compilers. We use the `-O3` optimization flag which enables automatic vectorization. Targeted architectures are using the Arm AARCH64 instruction set which includes the NEON instruction set by default. To generate SVE versions, we need to explicitly activate its support with the `-march=armv8.2-a+sve` flag. Results presented in this section have been obtained on two Arm-based platforms:

- a Marvell ThunderX2 dual-socket with 2x32 cores,
- a 64-cores AWS Graviton2 processor.

Processor characteristics are detailed in table II. Our kernel runs on a single core to study the effect of vectorization

TABLE II: Details of architectures.

| | ThunderX2 | Graviton2 |
|---|---|---|
| architecture | Broadcom Vulcan | Arm Neoverse N1 |
| ISA | ARMv8.1 | ARMv8.2 |
| frequencies | 2.2-2.5Ghz | 2.5Ghz |
| cache(L1/L2/L3) | 32kB/256kB/32MB | 64kB/1MB/32MB |
| memory | 8x DDR4-2667 | 8x DDR4-3200 |

exclusively. Additionally, we consider dynamic binary instrumentation and the Arm Instruction Emulator tool to validate SVE-based implementations and to further analyze instructions breakdown. Note that, using NSIMD has no measurable impact on the compilation time compared to using intrinsics.

### B. NEON performance

Figure 1 shows the timing results. As expected, we observe the impact of using intrinsics to enable vectorization with a two-fold speedup on the ThunderX2 (2.26x). With the 128-bit NEON unit, the theoretical maximum speedup is four in this case (single precision). However, with the irregular memory access pattern in our code we are clearly in a memory bound scenario. On the Graviton2 architecture, the situation is rather different since the scalar implementation of the kernel is significantly faster compared with the same benchmark on the ThunderX2 processor (2.8x). These results are due in part to the slightly higher bandwidth but mainly to the improved micro-architecture, since it is more recent. To confirm this, we have tested the binary obtained for the ThunderX2 on the Graviton2 and we observe the same speedup. Unfortunately, the usage of intrinsics does not bring the same level of acceleration on this second platform (1.4x). These encouraging results are still under investigation, but may be explained by the memory bandwidth which is theoretically only 20% higher on the Graviton2 thanks to a higher frequency (see table II). Thus, the micro-architecture improvement may put more pressure on the memory subsystem.

Regarding the NSIMD results, we obtain the same level of performance than the intrinsics version when using both compilers, on both platforms. Assembly codes for both versions are similar, we only observe small differences, mostly instructions not organized in the same order. Results show that it slightly impacts performance, and depends on the compiler. On the ThunderX2, the NSIMD version with GCC is around 10% slower than the intrinsics one, but on the Graviton2 the NSIMD version with Armclang is 14% faster. Note that, these differences are not significant if we consider the speedup brought by explicit vectorization over autovectorized versions.

### C. SVE: NSIMD vs intrinsics

We compare the number of SVE instructions produced by using intrinsics and NSIMD when using both GCC and Armclang with a 512-bit vector length, provided in table III. We observe that the total number of instructions are very different between each versions, especially those compiled with Armclang. By analyzing the assembly codes, we observe very important differences. It is not the case for the NEON

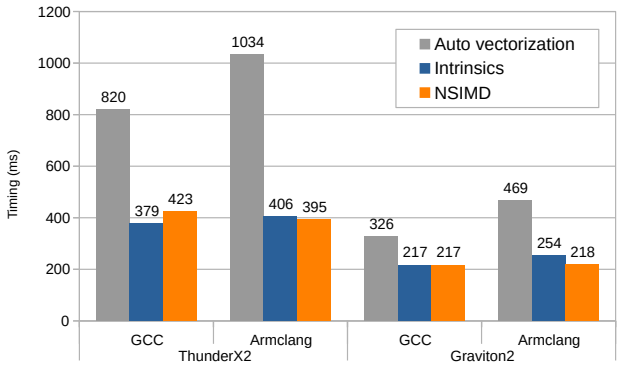


Fig. 1: Timing results on various architectures (automatic vs explicit vectorization for a Spectral Finite-element kernel)

TABLE III: Ratio SVE/total # of instructions (millions)

| intr.+GCC | intr.+Armclang | NSIMD+GCC | NSIMD+Armclang |
|---|---|---|---|
| 174/258=67% | 209/373=56% | 179/256=70% | 169/284=60% |

versions but we believe that this is due to a higher maturity of the NEON support in current compilers. We need to further investigate these differences. We expect to reach the same level of performance between intrinsics and NSIMD versions with GCC since the total number of instructions (around 256M) and the ratio of vectorized instructions (around 70%) are similar. In this case, we may expect very good performance compared to autovectorized versions on SVE platforms. We now consider analysing the evolution of this ratio for the NSIMD version with GCC, depending on the vector length.

### D. SVE intrinsics analysis

We use ArmIE which translates unsupported SVE instructions, to evaluate the impact of the Scalable Vector Extension. Consequently, we can validate and further analyze SVE-based implementations. The analysis of the dynamic instruction execution traces at each SVE vector length (from 128 to 1024 bits) are represented in figure 2. The Region-of-interest feature (start and stop macros added in the source code) is implemented to capture the behavior of key loops and limit the amount of runtime data collected.

First of all, we can visually see the portion of the SVE instructions generated (blue bars). This ratio is close to 70% for all our experiments and re-emphases the inherent vector-length agnostic characteristic of our optimized implementation. Secondly, this plot also underlines the linear decrease of the SVE instructions count with the theoretical size of the vector. This was expected due to the design of the underlying algorithm but this comes with an additional benefit. In our case, we also observe a decrease of the native AArch64 operations (orange bars) mainly related to the control flow (*bcond, orr, ubfm* instructions are dominant).

## VII. Conclusion

Our contributions are twofold: 1) we show that explicit vectorization is mandatory to take further advantage of SIMD

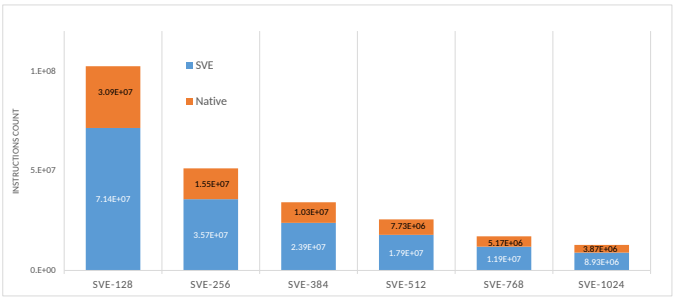


Fig. 2: Instructions count for the SVE instrinsics version with GCC. Varying vector lengths are considered with the Arm Instruction Emulator.

units present in all commercial processors today, 2) while using intrinsics is tedious and not portable, we demonstrate that it is possible to build high level abstractions, like NSIMD, for both fixed-length and VLA programming models, and reach the same level of performance for NEON with both Armclang and GCC compilers. We expect the same results for SVE architectures but we still need to evaluate the performance on upcoming SVE architectures like the Fujitsu A64FX to confirm our hypothesis. Further work also includes optimizing the memory access pattern of our kernel to reduce the stress on the memory subsystem.

## References


[1] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[2] S. Jubertie, I. Masliah, and J. Falcou, "Data layout and SIMD abstraction layers: Decoupling interfaces from implementations," in *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*. IEEE, 2018, pp. 531–538. [Online]. Available: https://doi.org/10.1109/HPCS.2018.00089

[3] G. Sornet, F. Dupros, and S. Jubertie, "A multi-level optimization strategy to improve the performance of stencil computation," *Procedia Computer Science*, vol. 108, pp. 1083 – 1092, 2017, international Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.

[4] M. Kretz and V. Lindenstruth, "Vc: A c++ library for explicit vectorization," *Software: Practice and Experience*, vol. 42, 11 2012.

[5] A. Fog, "VCL: C++ vector class library," pp. 1–110, 2016. [Online]. Available: http://www.agner.org/optimize/#vectorclass

[6] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté, "Boost.simd: generic programming for portable simdization," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM, 2014, pp. 1–8.

[7] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: Association for Computing Machinery, 1983, p. 177–189. [Online]. Available: https://doi.org/10.1145/567067.567085

[8] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 4, p. 491–542, Oct. 1987. [Online]. Available: https://doi.org/10.1145/29873.29875

[9] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 145–156. [Online]. Available: https://doi.org/10.1145/349299.349320

[10] R. Rocha, V. Porpodas, P. Petoumenos, L. Góes, Z. Wang, M. Cole, and H. Leather, "Vectorization-aware loop unrolling with seed forwarding," 02 2020.

[11] C. Mendis and S. Amarasinghe, "Goslp: Globally optimized superword level parallelism framework," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276480

[12] S. Maleki, Y. Gao, M. J. Garzar'n, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 372–382.

[13] M. Alvanos and P. Trancoso, "Video simdbench: Benchmarking the compiler vectorization for multimedia applications," in *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 168–175.

[14] S. Siso, W. Armour, and J. Thiyagalingam, "Evaluating auto-vectorizing compilers through objective withdrawal of useful information," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3356842

[15] S. Jubertie, F. Dupros, and F. D. Martin, "Vectorization of a spectral finite-element numerical kernel," in *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, J. Eitzinger and J. C. Brodman, Eds. ACM, 2018, pp. 8:1–8:7. [Online]. Available: https://doi.org/10.1145/3178433.3178441

[16] S. Jubertie, E. Melin, N. Raliravaka, E. Bodèle, and P. E. Bocanegra, "Impact of vectorization and multithreading on performance and energy consumption on jetson boards," in *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*. IEEE, 2018, pp. 276–283. [Online]. Available: https://doi.org/10.1109/HPCS.2018.00055

[17] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[18] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, and C. Jégo, "MIPP: A portable C++ SIMD wrapper and its use for error correction coding in 5G standard," *WPMVP 2018 - Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing, Co-located with PPoPP 2018*, vol. 2018-March, no. February, 2018.

[19] P. Kanapickas, "libsimdpp," https://github.com/p12tic/libsimdpp, 2017.

[20] R. Möller, "Design of a low-level C++ template SIMD library," 2016. [Online]. Available: www.ti.uni-bielefeld.de/html/people/moeller/tsimd_warpingsimd.html

[21] "bsimd: Introduction," https://developer.numscale.com/bsimd, numscale, 2017, accessed 2017-09-15.

[22] Quantstack, "xsimd," https://github.com/xtensor-stack/xsimd, 2018.

[23] A. Scale, "nsimd," https://github.com/agenium-scale/nsimd, 2019.

[24] P. Moczo, J. Robertsson, and L. Eisner, "The finite-difference time-domain method for modeling of seismic wave propagation," in *Advances in Wave Propagation in Heterogeneous Media*, ser. Advances in Geophysics. Elsevier - Academic Press, 2007, vol. 48, ch. 8, pp. 421–516.

[25] D. Komatitsch, "Méthodes spectrales et éléments spectraux pour l'équation de l'élastodynamique 2D et 3D en milieu hétérogène (Spectral and spectral-element methods for the 2D and 3D elastodynamics equations in heterogeneous media)," Ph.D. dissertation, Institut de Physique du Globe, Paris, France, May 1997, 187 pages.

[26] S. Tsuboi, K. Ando, T. Miyoshi, D. Peter, D. Komatitsch, and J. Tromp, "A 1.8 trillion degrees-of-freedom, 1.24 petaflops global seismic wave simulation on the K computer," *IJHPCA*, vol. 30, no. 4, pp. 411–422, 2016.

[27] D. Roten, Y. Cui, K. B. Olsen, S. M. Day, K. Withers, W. H. Savran, P. Wang, and D. Mu, "High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016, pp. 957–968.

[28] A. Breuer, A. Heinecke, and M. Bader, "Petascale local time stepping for the ADER-DG finite element method," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 854–863.

[29] F. Krużel and K. Banaś, "Vectorized opencl implementation of numerical integration for higher order finite elements," *Computers & Mathematics with Applications*, vol. 66, no. 10, pp. 2030 – 2044, 2013, iCNC-FSKD 2012.