On the usage of the Arm C Language Extensions for a High-Order Finite-Element Kernel

Sylvain Jubertie, Guillaume Quintin, Fabrice Dupros

On the usage of the Arm C Language Extensions for a High-Order Finite-Element Kernel

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

Guillaume Quintin
Agenium Scale
France
guillaume.quintin@agenium.com

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

***Abstract*—Physics-based three-dimensional numerical simulations are becoming more predictive and have already become essential. As an example 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. Consequently, the diversity of HPC architectures available (heterogeneity, high core counts or depth of the memory hierarchy) leads to increasing concerns regarding the portability of applicative performances. In this paper, we discuss the implementation of the classical spectral finite-elements method using the Arm C Language Extensions for both Neon and SVE SIMD units.**

I. Introduction

Explicit parallel elastodynamics application usually exhibits very good weak and strong scaling up to several tens of thousands of cores ([1], [2], [3]). Regardless of the numerical method involved (Finite-Differences, Finite-Elements or Spectral-Element Method (SEM)), this class of application benefits from quite dense numerical kernel and limited amount of point-to-point communications between neighboring subdomains. Significant works have been made to extend this parallel results on heterogeneous and low-power processor ([4], [5]). Due to their numerical efficiency, high-order finite-elements methods are routinely used for seismic simulations (e.g. foundational work of SpecFEM3D software package is described here ([6]). Several challenges are well known for the optimization of this algorithm. As an example, it is admitted that the summation of the element contributions (assembly phase) represents a major bottleneck of such approaches to scale out. This is coming both from the shared values between neighboring elements and the inherent indirection for data accesses.

Additionally, the computation of the internal forces can represent as much as 80% of the total elapsed time. Unfortunately, so far only hand-tuned implementations have been able to benefit from advanced SIMD units available on modern architectures ([7], [8]). The SVE (Scalable Vector Extension) SIMD instruction set brings a Vector-Length Agnostic (VLA) programming model that could address these limitations.

In this short paper, we discuss the implementation of a spectral-elements method kernel using the Arm C Language Extensions. We underline key aspect of both Neon or SVE implementations and show performances obtained on two Neon-based processors (the AWS Graviton2 and the Marvell ThunderX2). Results from the Arm Instruction Emulator (ArmIE) with varying SVE vector lengths are also presented.

The remainder of this paper is organized as follows. Section II provides the numerical background regarding seismic wave propagation. A detailed description of our implementations using the Arm C Language Extension is provided in Section III. Finally, we discuss the results obtained in Sections IV.

II. Spectral Element numerical kernel

A. Governing equations

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 ρ 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 τ_{ij} with respect to x_j; f_i is the i-th component of the body force.

The weak form of eq. 1 is expressed in eq. 2, 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 Ω and Γ 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 Γ. Superscript T denotes the transpose, and a colon denotes the contracted tensor product.

B. Characterization of the numerical kernel

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

Gathering points for each element consists in copying values from a global-to-local operation. An indirection array is implemented as the same grid point can be shared by up to eight

elements (corners). 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 in 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 following the same indirection pattern as for the gathering step.

III. Vectorization

A. Neon implementation

To the best of our knowledge, the most efficient strategy to vectorize the kernel previously described is to target the outer most loop (i.e. see here for details [7]). In this case, porting the kernel to Neon is 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 in 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 local[ 5 * 5 * 5 * 9 ];
```

becomes:

```
float32x4_t local[ 5 * 5 * 5 * 9 ];
```

2) Gather/scatter: Non-contiguous loads required by the vectorization and the data storage pattern had 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: The same coefficient has to be applied to all the values in the vector, thus requiring to explicitly call the `vdupq_n_f32` intrinsics to duplicate the coefficient:

```
auto vcoeff = vdupq_n_f32( coeff );
```

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 scalar and vectors to remove some calls to the `vdupq_n_f32` intrinsics, like the one in the following expression:

```
auto tau=(rhovp2-vdupq_n_f32(2.0f)*rhovs2)
        *(duxdx+duydy+duzdz);
```

B. SVE implementation

The SVE approach is quite different since the vector length is only at runtime through the use of the `svcntw` intrinsics which returns the number of 32-bit words in one SVE register. Obviously, one of the key aspect of this feature is to generate a vector length agnostic code portable across SVE-enabled hardwares. 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, the implementation of 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 they require a mask, thus increasing the verbosity of the code. 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

	ThunderX2	Graviton2
frequencies	2.2-2.5GHz	2.5GHz
cache(L1/L2/L3)	32kB/256kB/32MB	64kB/1MB/32MB
memory	8x DDR4-2667	8x DDR4-3200

TABLE I: Arm-based architectures used for this study.

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.

IV. Experiments and Results

A. Setup

In this section, we discuss both implementations of the seismic kernel : 1) scalar version with autovectorization, 2) usage of the Arm C Langauge Extensions for Neon and SVE. For these experiments, we use GCC (v10.1) and Armclang (v20.1) LLVM-based compilers. We use the `-Ofast` optimization flag which enables automatic vectorization. To generate SVE versions, we need to explicitly activate its support with the `-march=armv8-a+sve` flag. Results presented in this section are from a serie of exeuctions carried out 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 I. For the SVE analysis, the Arm Instruction Emulator (ArmIE) is used as this tool can execute unsupported instructions on AArch64 architecture by converting them into native ones. Dynamic binary instrumentation (through DynamoRIO) is used to extract additional metrics such as the memory traces([9]).

B. Timing results on Neon enabled hardware

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 (2.3x). With the 128-bit Neon unit, the theoretical maximum speedup is four (single precision), but the irregularity of the memory access pattern clearly hurts the performances. On the Graviton2 architecture, the situation is rather different since the standard implementation of the kernel is significantly faster compared with the same benchmark on the ThunderX2 processor (2.7x). As the vectorization level is poor for this baseline version, these results are mainly due to the improved micro-architecture. The usage of intrinsics do not bring the same level of acceleration on this second platform (1.3x). In this case, the pressure on the memory hierarchy may explain these results. Overall, both GNU and Arm Compiler for Linux deliver similar level of performances and performance numbers are within a margin of 5%.

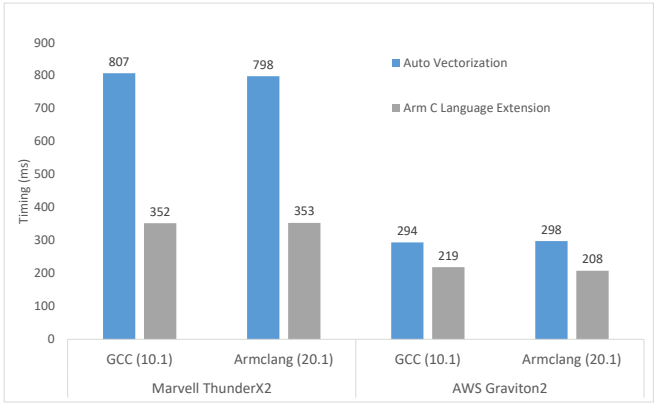

Fig. 1: Timing results on two different Arm-based architectures with GNU and Arm Compiler for Linux compilers.

C. SVE study

Dynamic instruction execution traces at each SVE vector length (from 128 to 1024 bits) are represented in Figure 2 using the same binary. We represent both the native and the emulated instructions counts. As expected, the number of instructions is divided by a factor of almost two every time we double the length of the SVE vector. This demonstrates both the high ratio of vectorization for this kernel (an average of 70% for all the configurations) but also the efficacy of our implementation.

The percentage of utilization of the vector lanes is a very good metric to discuss this last point. SVE is predicate-centric architecture, it is therefore key to collect metrics related to the

exploitation of the vector lanes. In our case, the usage of the SIMD lanes appears to be optimal (nearly 100%) for all the vector lengths evaluated.

As the introduction of native scatter and gather operations is

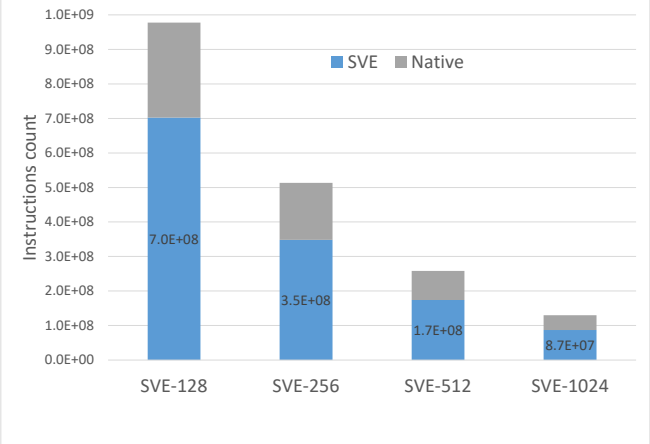

Fig. 2: Dynamic instructions count (GNU compiler) for the Spectral Finite-element kernel

one of the key feature of SVE, the breakdown of the memory operations is an interesting metric. Figure 3 summarizes the corresponding instructions for the high-order finite-elements kernel. As expected, a significant portion of the memory accesses (20%) are implemented with $gather/scatter$ instructions.

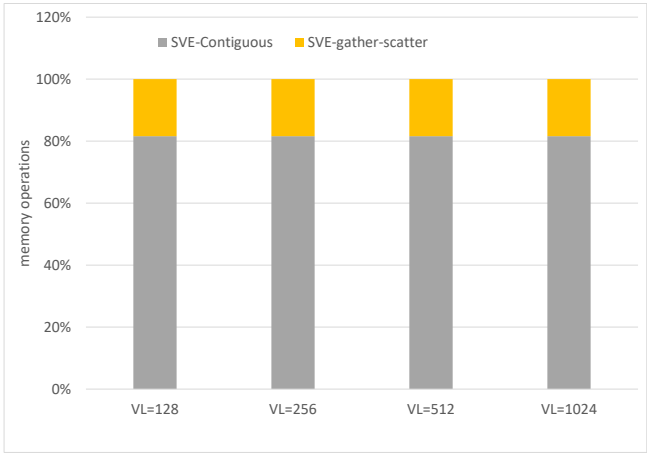

Fig. 3: Memory operations (GNU compiler) for the Spectral Finite-elements method.

V. Conclusion

In this paper, we study the explicit vectorization of a Seismic Spectral-Element kernel using Neon and SVE intrinsics to evaluate the complexity of both fixed length and VLA programming models. The results from AWS Graviton2 and Marvell ThunderX2 processors show the impact of explicit vectorization with significant speedups on both platforms. The SVE analysis demonstrates the efficacy of our implementation despite the complexity of the key loops and the irregular memory accesses. Obviously these findings should be compared with insights from real hardware and we expect to refine these analysis in future works. Additionally, several higher level abstractions over intrinsics like Vc[10], Boost.SIMD[11] are available to provide both performance and portability. We expect to explore these options to further improve our implementations.

References

[1] 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.

[2] 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.

[3] 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.

[4] D. Göddeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, and A. Ramírez, "Energy efficiency vs. performance of the numerical solution of pdes: An application study on a low-power arm-based cluster," *J. Comput. Physics*, vol. 237, pp. 132–150, 2013.

[5] M. Castro, E. Francesquini, F. Dupros, H. Aochi, P. O. A. Navaux, and J. Méhaut, "Seismic wave propagation simulations on low-power and performance-centric manycores," *Parallel Computing*, vol. 54, pp. 108–120, 2016.

[6] 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.

[7] 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

[8] 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.

[9] M. T. Cruz, D. Ruiz, and R. Rusitoru, "Asvie: A timing-agnostic sve optimization methodology," in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, Nov 2019, pp. 9–16.

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

[11] 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.