Vectorization of a spectral finite-element numerical kernel

Sylvain Jubertie, Fabrice Dupros, Florent de Martin

Vectorization of a spectral finite-element numerical kernel

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

Fabrice Dupros
BRGM
BP 6009, 45060 Orléans, France
f.dupros@brgm.fr

Florent De Martin
BRGM
BP 6009, 45060 Orléans, France
f.demartin@brgm.fr

Abstract

In this paper, we present an optimized implementation of the Finite-Element Methods numerical kernel for SIMD vectorization. A typical application is the modelling of seismic wave propagation. In this case, the computations at the element level are generally based on nested loops where the memory accesses are non-contiguous. Moreover, the back and forth from the element level to the global level (e.g., assembly phase) is a serious brake for automatic vectorization by compilers and for efficient reuse of data at the cache memory levels. This is particularly true when the problem under study relies on an unstructured mesh.
The application proxies used for our experiments were extracted from EFISPEC code that implements the spectral finite-element method to solve the elastodynamic equations. We underline that the intra-node performance may be further improved. Additionally, we show that standard compilers such as GNU GCC, Clang and Intel ICC are unable to perform automatic vectorization even when the nested loops were reorganized or when SIMD pragmas were added.
Due to the irregular memory access pattern, we introduce a dedicated strategy to squeeze the maximum performance out of the SIMD units. Experiments are carried out on Intel Broadwell and Skylake platforms that respectively offer AVX2 and AVX-512 SIMD units. We believe that our vectorization approach may be generic enough to be adapted to other codes.

Keywords FEM, vectorization, SIMD, mini-app

1 Introduction

Realistic-sized three-dimensional earthquake modeling is extremely computationally intensive. However recent advances in High Performance Computing (HPC) platforms make numerical simulation of seismic wave propagation feasible at a large scale and at high seismic frequencies [4, 6, 18, 24]. Up until recently, the improvement of the performances of the applications were mainly coming from Moore's Law. Furthermore, the flat MPI programming model was enough to express the application-level parallelism. The quest for growing performance under a strict power budget has led to the introduction of much more complex processors with multiple levels of parallelism.
Indeed a simple way to improve the performance of processors is to use Data Level Parallelism to process several data at the same time with a single instruction. This idea was first developed for supercomputers in the form of vector processors, but is now implemented in almost all processor architectures as a dedicated SIMD (Single Instruction on Multiple Data) unit.
Leveraging emerging chips with wide SIMD units requires significant transformation in the data-layout and the organization of the computation.
Several solutions are available: 1) using compilers to automatically transform a scalar code into a vectorized one, 2) using compiler intrinsics which are binded to assembly instructions, 3) using optimized libraries such as MKL, LAPACK, FFTW or Arch-R. In this paper we focus on x86 AVX2 and AVX-512 units, since AVX2 is supported by Intel Broadwell processors used in a lot of supercomputers and since AVX-512 supported by Intel Skylake processors is promoted as the upcoming standard.
Theoretically, AVX2 and AVX-512 units may allow respectively a speedup of 8 and 16 when computing 32-bit floating point values. To attain the best speedup with SIMD units, one has to respect several constraints: 1) exhibiting a SIMD computation pattern 2) accessing contiguous data, 3) correct data alignment, 4) Structure of Array (SoA) or Array of Structure of Array (AoSoA) layouts. Of course, compute-bound codes are more likely to reach the theoretical peak speedup. Depending on the considered architecture, some constraints may be more or less prevalent. The first constraint is obvious, if the same instruction cannot be applied to some data then the code is not vectorizable (except for some `addsub` instructions). A consequence of this constraint is that branches need to be replaced by masked instructions.
Then, data need to be stored contiguously since a single SIMD load instruction is able to read up to 256-bit (resp. 512-bit) of data when using the AVX2 (resp. AVX-512) unit. It is possible to access to non-contiguous data using gather and broadcast intrinsics. Data also have to be aligned in memory as required by the SIMD unit. For example, the AVX unit requires data to be aligned on 32-byte boundaries. It is of

WPMVP'18 , February 24–28, 2018, Vienna, Austria
2018. ACM ISBN 978-1-4503-5646-6/18/02…$15.00
https://doi.org/10.1145/3178433.3178441

course possible to access unaligned data, however, it implies an additional cost. Finally, data have to be organized with a suitable layout to avoid extra data reorganization inside the SIMD registers.

In this paper, we study the EFISPEC3D seismic wave propagation software package [8] based on a Spectral Finite-Element Method (SFEM). We have extracted a proxy from the full application corresponding to the computation of the internal forces (almost 90% of the elapsed time). This strategy allows us to smoothly evaluate our vectorization strategy by rewritting this intensive section of the code in C++.

The remainder of this paper is organized as follows. In the next section, the related work is described. In section 3, we discuss the fundamentals of seismic wave propagation and the EFISPEC code is introduced. Then, in section 4 we introduce the proxy application and its SIMD implementation. In section 5, we present the results obtained with the proxy application. Finally, we conclude this study and present some future work.

2 Related work

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 SFEM) have been introduced last few years for this class of problems. Interested readers could refer to [19] for further details on these approaches.

From the HPC point of view, one major challenge is to leverage the various levels of parallelism currently available. If we consider the parallel performance at large scale, several recent research papers ([2, 22, 26]) have reported very good scaling for explicit parallel elastodynamics applications (up to several tens of thousands of cores). Whatever the numerical method implemented, the rather dense numerical kernel along with MPI point-to-point communications explain these performances. Significant works have also been made to extend this parallel results on heterogeneous and low-power processor ([3, 11]).

Regarding parallel FEM assembly on shared-memory architectures, the numerical scheme requires to gather values computed at the element-level based on an indirection array. These irregular memory accesses dramatically reduce the opportunity to reuse data at the cache memory level. For instance, optimized implementations have been introduced on GPGPU [5, 16, 21]. Most of these approaches implement mesh coloring strategy and fully benefit from the memory bandwidth available on the underlying architecture. Additionally, various multicore-aware implementations have been described (for instance in [1, 10, 12]). These papers underline the need for advanced reorganization of the computation at the algorithmic level (Cuthill-McKee graph traversal [7]) or at higher level (task-based runtime systems for instance). This latter option is described in [25] with the implementation of a divide and conquer algorithm that exploits Intel Cilk multithreading library. If we focus on SIMD only optimizations, the compiler performance plays a major role in the works previously described. Unfortunately, recent studies have underlined the limited impact of automatic vectorization [14, 15, 23]. Indeed, this strategy may not be sufficient to squeeze the optimal performance out of the underlying architecture.

3 EFISPEC: Spectral finite element solver

3.1 Spectral-element method

The spectral-element method (SEM) appeared more than 20 years ago in computational fluid mechanics [9, 17, 20]. The SEM is a specific formulation of the finite-element method for which the interpolated points and the quadrature points of an element share the same location. These points are the Gauss–Lobatto–Legendre (GLL) points, which are the $p+1$ roots of $(1-\xi^2)P'_p(\xi)=0$, where P'_p denotes the derivative of the Legendre polynomial of degree p and ξ coordinate in the one-dimensional reference space $\Lambda=[-1,1]$.

The generalization to higher dimensions is done through the tensorization of the one-dimensional reference space. In three dimensions, the reference space is the cube $\square = \Lambda\times\Lambda\times\Lambda$ (see Fig. 1).

The mapping from the reference cube to a hexahedral element Ω_e is done by a regular diffeomorphism $\mathcal{F}_e : \square \rightarrow \Omega_e$. In a finite-element method, the domain of study is discretized by subdividing its volume Ω into welded non-overlapping hexahedral elements $\Omega_e, e=1,\ldots,n_e$ such that $\Omega = \cup_{e=1}^{n_e}\Omega_e$. The elements Ω_e form the mesh of the domain. On the one hand, each element Ω_e has a local numbering of the GLL points ranging from 1 to $p+1$ along each dimension of the tensorization. On the other hand, the mesh has a unique global numbering ranging from 1 to N. The mapping from the local numbering to the global numbering is the so-called "assembly" phase of all finite-element calculations.

Each GLL point of an element Ω_e is redirected to a unique global number, $\forall\Omega_e$. When multiple elements share a common face, edge or corner, the assembly phase sums the local GLL value into the global numbering system. In this article, the problem of interest is the equation of motion whose weak formulation is given by

$$\int_\Omega \rho\mathbf{w}^T\cdot\mathbf{u}\,d\Omega = \int_\Omega \boldsymbol{\nabla}\mathbf{w}:\boldsymbol{\tau}\,d\Omega - \int_\Omega \mathbf{w}^T\cdot\mathbf{f}\,d\Omega - \int_\Gamma \mathbf{w}^T\cdot\mathbf{T}\,d\Gamma$$

where Ω and Γ are the volume and the surface area of the domain under study, respectively; ρ is the material density; $\mathbf{w}$ is the test vector; $\mathbf{u}$ is the second time-derivative of the displacement $\mathbf{u}$; $\boldsymbol{\tau}$ is the stress tensor; $\mathbf{f}$ 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.

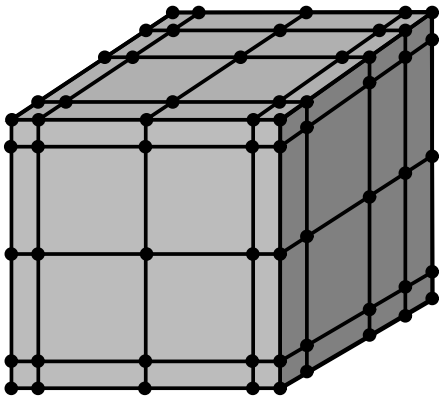

Figure 1. Referenced cube with $(4+1)^3 = 125$ GLL points.

Our study focuses on the internal forces defined by (see [13])

$$\int_{\Omega_e} \boldsymbol{\nabla}\mathbf{w} : \boldsymbol{\tau}\, d\Omega_e \approx \sum_{\alpha=1}^{p+1}\sum_{\beta=1}^{p+1}\sum_{\gamma=1}^{p+1}\sum_{i=1}^{3} \mathrm{w}_i^{\alpha\beta\gamma} \times \Bigg[\omega_\beta\omega_\gamma \sum_{\alpha'=1}^{p+1} \Bigg[\omega_{\alpha'} \mathcal{J}_e^{\alpha'\beta\gamma} \sum_{j=1}^{3} \left[\tau_{ij}^{\alpha'\beta\gamma}\, \partial_j \xi_{\alpha'}\right] \ell'_\alpha(\xi_{\alpha'}) \Bigg] + \omega_\alpha\omega_\gamma \sum_{\beta'=1}^{p+1} \Bigg[\omega_{\beta'} \mathcal{J}_e^{\alpha\beta'\gamma} \sum_{j=1}^{3} \left[\tau_{ij}^{\alpha\beta'\gamma}\, \partial_j \eta_{\beta'}\right] \ell'_\beta(\eta_{\beta'}) \Bigg] + \omega_\alpha\omega_\beta \sum_{\gamma'=1}^{p+1} \Bigg[\omega_{\gamma'} \mathcal{J}_e^{\alpha\beta\gamma'} \sum_{j=1}^{3} \left[\tau_{ij}^{\alpha\beta\gamma'}\, \partial_j \zeta_{\gamma'}\right] \ell'_\gamma(\zeta_{\gamma'}) \Bigg] \Bigg] \quad (1)$$

with $\boldsymbol{\tau}$ the stress tensor ($= \boldsymbol{c} : \nabla\mathbf{u}$); $\mathcal{J}_e^{\alpha'\beta\gamma}$ the jacobian of an element Ω_e at the GLL points $\alpha'\beta\gamma$; ω_λ integration weight at the GLL point λ; ξ, η, ζ local coordinates along the three dimensions of the reference cube; ℓ'_λ derivative of the Lagrange polynomial at the GLL point λ. $\boldsymbol{c}$ is the elastic tensor and $\nabla\mathbf{u}$ is the gradient of the displacement defined by

$$\partial_i \mathrm{u}_j\left(\xi_\alpha\eta_\beta\zeta_\gamma\right) = \left[\sum_{\sigma=1}^{p+1} u_j^{\sigma\beta\gamma}\, \ell'_\sigma\left(\xi_\alpha\right) \partial_i \xi_{\alpha\beta\gamma}\right] + \left[\sum_{\sigma=1}^{p+1} u_j^{\alpha\sigma\gamma}\, \ell'_\sigma\left(\eta_\alpha\right) \partial_i \eta_{\alpha\beta\gamma}\right] + \left[\sum_{\sigma=1}^{p+1} u_j^{\alpha\beta\sigma}\, \ell'_\sigma\left(\zeta_\alpha\right) \partial_i \zeta_{\alpha\beta\gamma}\right]$$

3.2 Implementation

3.2.1 Element storage and numberings

Since the considered meshes are unstructured with multiple refinements (see Fig. 2), the indirection from local to global numberings can not be expressed in a closed-form expression. A common approach to switch from local to global numberings is the use of an indirection array, as shown in figure 3: given the element number and the local GLL number (from 0 to $(p+1)^3 - 1$), the array returns the global GLL number. In figure 3, the first two elements are neighbours and share common global GLL numbers (4, 9, 14, ...). In addition, each global GLL number is a computational point where physical values are associated (such as, displacement in the x, y, z-directions).

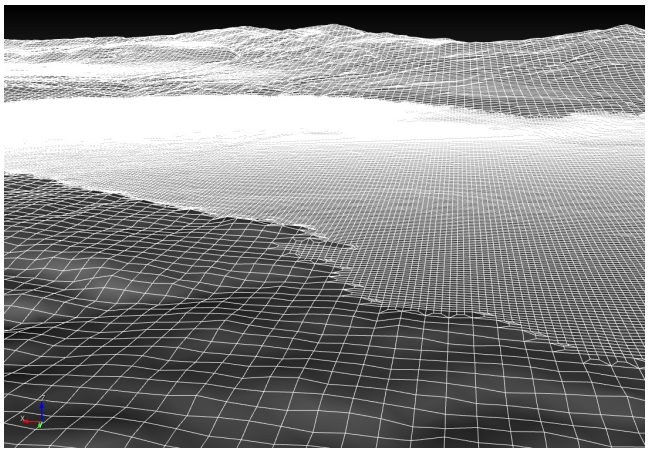

Figure 2. Example of unstructured mesh generated by CUBIT meshing tool and used by EFISPEC3D code to solve the equation of motion.

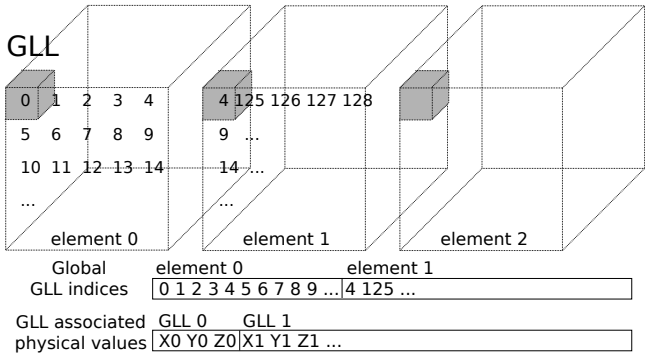

Figure 3. Storage of the elements and their GLLs.

3.2.2 Internal forces computation kernel

The main loop of the kernel iterates over elements. For each element the following three steps are performed:

1. gathering: GLL values of an element are copied into a local array,
2. internal forces computation,
3. assembly: contributions of the element are added into a global array.

Each step consists in traversing the GLLs of the element, thus it is implemented using three nested loops which iterate

over the element dimensions. The first and second steps may be merged by replacing accesses to the local array in the computation step by global accesses. However, this implementation is not as fast since arithmetic instructions have to wait for the completion of load instructions from memory. The chosen implementation on the other side stores all the GLLs into a local array which fits into the L1 cache.

The internal forces computation code is derived from equation 1. It consists in traversing each element along its three dimensions and compute each GLL contribution in a local array. It requires to access some GLL points parameters stored in different arrays like integration weights and derivatives of the Lagrange polynomial. It is composed mainly of multiplications and additions which may be translated into Fused Multiply Add (FMA) instructions by the compiler on current architectures.

The assembly step consists in adding local GLL contributions to the corresponding global array.

4 Vectorization of the EFISPEC mini-app

Prior to consider manual vectorization, several attempts were made on the Fortran code to optimize the computation of the elements and to enable automatic vectorization. SIMD pragmas were added, arrays were aligned accordingly to the SIMD instruction set considered, and loops were reorganized, but none of these modifications allowed the compiler to completely vectorize the nested loops.

4.1 Manual vectorization approach

Since the same computation is applied to all the elements, we propose to manually vectorize the external loop iterating on elements. With AVX2 it is possible to handle 8 elements at the same time. Figure 4 illustrates the proposed vectorization approach. Thus, the three internal loops iterating on GLLs are kept unmodified, only the external loop needs to iterate by the number of 32-bit floating point values available in a SIMD register. This approach has the advantage to keep the instructions organized as in the scalar version. We now present how this choice is implemented for the three different parts of the element computation.

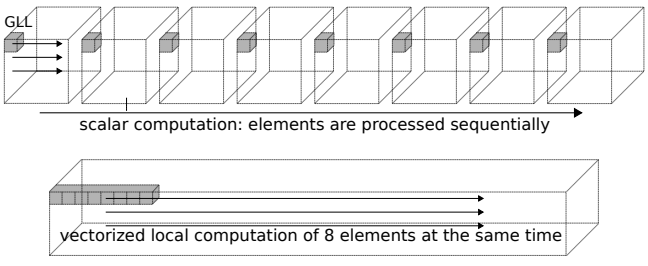

Figure 4. Scalar and vectorized approaches to process elements.

4.1.1 Gathering

As described above, each element is composed of 125 GLLs which have to be gathered into a local array before starting the local computation. The AVX2 version of this step consists in gathering 8 times more elements (resp. 16 times for AVX-512). In EFISPEC, global GLL indices of elements are stored in an array called `ig_hexa_gll_glonum`. This array starts with the 125 indices of the first element GLLs, then those of the second element, and so on. For the computation, the code needs to load in the local array the first GLL of the 8 first elements, but their indices are not stored contiguously. A solution available since SSE is to use a `set` intrinsics which gathers data of an array from the indices given as its arguments. Since AVX2, a `gather` intrinsics is available which gathers data from indices stored in a SIMD register. In this case we need to reorder the indices by interleaving them and correctly align the array. This step may be performed when the array is generated thus it may not involve an overhead.

4.1.2 Internal forces computation

As discussed in section 3.2.2, the internal forces computation requires to access some GLL parameters. For parameters which are the same for GLLs at the same position in each element, we use a `set1` intrinsics which sets all the SIMD register values to the same value. For specific GLLs parameters stored in an AoS layout, we use a `set` intrinsics which takes eight values (non contiguous in our case) as parameters and assembles them into a SIMD register. We also propose to reorder them into an SoA layout to improve the data locality so we can use `aligned load` intrinsics instead which simplifies the code and may provide better performance since contiguous data are loaded.

The computation itself is straightforward to vectorize since all the scalar arithmetic instructions used in the original code have their SIMD counterparts. Moreover, compilers provide overloaded arithmetic operators for SIMD types. Thus, the vectorized version of this step differs from the original one only by the presence of `set`, `set1` and `loads` intrinsics.

4.1.3 Assembly

The AVX2 instruction set does not provide `scatter` intrinsics to store local contributions into the corresponding global array. As a consequence, the assembly step can not be vectorized and is kept unmodified.

The AVX-512 instruction set provides a `scatter` intrinsics but, when using it, concurrent writes may occur depending on the way contributions are stored. Indeed, some elements may be arbitrarily rotated during the generation of unstructured meshes. In figure 5, the first two configurations are conflict-free since SIMD units are synchronous. The last configuration is one possible conflicting configuration which occurs when one of two contiguous element is rotated. In this case, the same GLL (in blue) may be loaded twice within

the same SIMD register. Thus, after the vectorized internal forces computation, the two contributions of the same GLL for each element may be stored in the same place. Note that, it only concerns GLLs at the corner, middle of edges or faces of contiguous elements. This configuration may be detected on the fly using conflict detection intrinsics available in the AVX-512CD subset. However, it adds a computation overhead and is not a portable solution. We propose two generic solutions to this problem. The first one consists in always storing contributions sequentially. This is the default approach when using AVX2 or older units (AVX and SSE) since no scatter intrinsics is available. However, this increases the verbosity of the code and may be not optimal when using AVX-512. The second and most general solution is to add a pre-processing step to detect this configuration, and permute or rotate elements if necessary. This is the chosen solution for our proxy application.

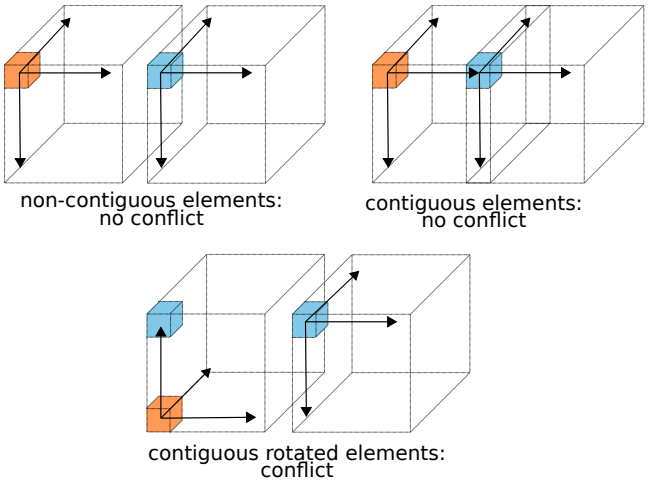

Figure 5. Conflict-free and conflicting configurations of elements.

4.2 AVX2 and AVX-512 implementations

Since most compilers provide overloaded arithmetic operators for intrinsics, it is possible to keep the syntax close to the scalar one. Only the load and store intrinsics are left visible in the code. AVX2 and AVX-512 versions are similar except for the intrinsics prefix and for the assembly part.

4.3 Proxy application versions

Different versions of the proxy application were developed to study and compare the performance brought by data-layout modifications and vectorization.

The proxy application, called `fort-local`, was first developed in Fortran to verify its correctness and behaviour with the original EFISPEC code, and to serve as a reference for further developments. This version was in turn translated into C++. It is called `c++-local`. Then two explicitly vectorized (AVX) versions were developed: the first one, `c++-local-avx`, with the local array and the second one, `c++-avx`, without the local array as discussed in section 3.2.2. It allows to measure the benefit of the local array on the performance. Finally, three different versions were proposed: `c++-local-avx-soa`, `c++-local-avx2-soa` and `c++-local-avx512-soa`, with data-layouts converted from AoS to SoA combined respectively with AVX, AVX2 and AVX-512 intrinsics. In this case we can study the impact of gather intrinsics brought by AVX2 and also the speedup brought by a larger AVX-512 unit. All the results are presented in the next section.

5 Experimentations

5.1 Experimental setup

We propose to use three different compilers: `g++` 7.2, `icpc` 18.0, `clang` 5.0, to compare their ability to optimize the codes. We use the `gfortran` and `ifort` Fortran compilers. All codes are compiled with the `-O3` optimization flag which also enables the compiler auto-vectorization process. The `-march=native` flag is also added to adapt the compilation to the architecture by enabling the support for the available SIMD unit and the FMA instructions.

	Broadwell	Skylake
Processor	Intel Xeon E5-2699 v4	Intel Xeon Gold 6148
# of cores	22	20
Base freq.	2.2 Ghz	2.4 Ghz
Turbo freq.	3.6 Ghz	3.7 Ghz
AVX freq.	1.8 Ghz (AVX)	1.6 Ghz (AVX-512)
L2	256KB	1024KB
L3	28MB	28MB

Table 1. Details on the Intel Broadwell and Skylake processors used.

Two processors are considered based on two different Intel architectures: Broadwell and Skylake. The details of both processors are given in table 1. To avoid variability of the core frequency, the turbo mode is disabled. Note that, on both architectures, the core frequency when executing AVX and AVX-512 instructions is lower than the base frequency.

Tests are performed on the same data set containing 40,000 elements.

5.2 Results

Results are shown in table 2. We discuss the results following the order in which versions were developed.

Fortran vs C++: We first observe that the Fortran and C++ versions are better optimized by `icpc` than by `g++`. An analysis of the assembly code generated by `icpc` shows that the code is partially vectorized. However, `icpc` is not able to

	g++-7.2 Bdw / Skl	**icpc-18.0** Bdw / Skl	**clang-5.0** Bdw / Skl
fort-local	755 / 629	282 / 225	-
local	682 / 559	367 / 294	716 / 654
avx	335 / 287	269 / 220	388 / 316
local-avx	233 / 195	152 / 122	251 / 203
local-avx-soa	217 / 190	141 / 119	257 / 206
local-avx2-soa	216 / 191	150 / 119	251 / 205
local-avx512-soa	— / 157	— / 99	— / 153

Table 2. Execution times (ms) for the different proxy-application versions.

optimize the C++ version to the same level of the Fortran version. The assembly code also shows some AVX instructions but it is organized very differently. The C++ code is around 10% faster with `g++`, and 30% slower with `icpc` on both architectures.

Scalar/Auto-vectorization vs AVX intrinsics: To determine the benefit brought by manual code vectorization, we compare the results for the `local` and the `local-avx` versions. Even if the `local` version may be partially vectorized by the compilers, the manual vectorization is far more efficient. We achieve a performance improvement of 65% with `g++`, 58% with `icpc` on both architectures, and of 65% on the Broadwell and 69% on the Skylake with `clang`.

Direct vs local access to GLLs: As explained in section 3.2.2, gathering GLLs in a local array prior to the computation provides better performance than accessing GLLs from the global array during the computation. We can also verify it for the AVX version by comparing the `avx` and the `local-avx` versions. With `g++`, `icpc` and `clang`, the `local-avx` version is respectively 30%, 42% and 45% faster than the `avx` version on both architectures.

AoS vs SoA: Transforming the data-layout from AoS to SoA does not improve significantly the performance. When comparing the `local-avx` and `local-avx-soa` versions, we observe at most 5% more performance with `g++` and `icpc` on the Broadwell architecture and no significant improvement on the Skylake architecture. Note that, it simplifies the code by replacing `set` intrinsics which take eight arguments (the values to put in an AVX register) by `load` intrinsics which take only one argument (the address where to load eight contiguous values).

AVX vs AVX2: Using AVX2 intrinsics for gathering GLLs values does not bring any performance advantage. However, it also highly simplifies the code by replacing the `set` intrinsics with `gather` intrinsics.

AVX2 vs AVX-512: We compare the `local-avx2-soa` and the `local-avx512-soa` versions for the Skylake architecture which supports the AVX-512 instruction set. Note that the considered processor is an Intel Xeon Gold 6148 which contains two AVX2 ports and an AVX-512 port. When an AVX-512 instruction is processed, it can be sent to the AVX-512 port or divided into two AVX2 instructions, each one sent to a different AVX2 port. This is not the case for Skylake processors from the Silver serie which do not have a dedicated AVX-512 port. Thus, with the considered Skylake processor we can expect a speedup of two when moving from AVX2 to AVX-512. However, results show that we only obtain around 17% more performance with `g++` and `icpc` and 25% with `clang`, which means that the memory bus is not able to sustain the demand of the AVX-512 unit.

Overall speedup: On both platforms, the best performance is obtained with the code compiled with the `icpc` compiler. On the Broadwell platform, a speedup of 2 is achieved by manually vectorizing the code using AVX intrinsics over the original Fortran code. On the Skylake platform, a speedup of 2.27 is achieved by using AVX-512 intrinsics. The benefit obtained with AVX-512 over AVX is far less than the theoretical speedup of two, which confirms that we have reached a memory bottleneck and that no further significant improvement may be obtained by further vectorization.

6 Conclusion and future work

The manual vectorization of our proxy application is able to provide at least a speedup of 2 on current architectures over the original Fortran version. Note that, this is not the speedup brought by the sole vectorization since the original version is already partially vectorized by compilers. However, the low speedup of the AVX-512 version over the AVX2 version confirms that vectorization can not be exploited at its best level in our case. This is an expected result since only the internal forces computation exhibits a SIMD friendly pattern.

The programming effort for the vectorization process is relatively low. The difficulty resides in identifying the SIMD pattern and adapting the memory accesses. The vectorized implementation keeps the same code organization as the Fortran version, only some intrinsics are required to load and store data.

We consider several directions for the future. The first one is the integration of the proxy application back into the EFISPEC application. Since it represents 90% of the full application and since we achieve a speedup of 2, we can expect a global speedup of 1.8. The second one consists in studying new data layouts to improve the memory locality since memory is the main bottleneck. The intra-node scalability of the EFISPEC application may also be improved by replacing MPI with OpenMP. The proxy application may be extended to study multithreading strategies. We finally plan to develop a Domain Specific Language (DSL) for FEM solvers to completely hide the intrinsics and abstracting the data layout thus helping the developer concentrate on the internal forces computation.

References

[1] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack J Dongarra, Christopher Earl, Joël Falcou, Azzam Haidar, Ian Karlin, Tzanio V Kolev, Ian Masliah, and Stanimire Tomov. 2016. High-performance Tensor Contractions for GPUs. In *International Conference on Computational Science 2016 (ICCS 2016)*, Vol. 80. 108–118.

[2] Alexander Breuer, Alexander Heinecke, and Michael Bader. 2016. 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*. 854–863.

[3] Márcio Castro, Emilio Francesquini, Fabrice Dupros, Hideo Aochi, Philippe O. A. Navaux, and Jean-François Méhaut. 2016. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Comput.* 54 (2016), 108–120.

[4] Emmanuel Chaljub, Emeline Maufroy, Peter Moczo, Jozef Kristek, Fabrice Hollender, Pierre-Yves Bard, Enrico Priolo, Peter Klin, Florent De Martin, Zhenguo Zhang, et al. 2015. 3-D numerical simulations of earthquake ground motion in sedimentary basins: testing accuracy through stringent models. *Geophysical Journal International* 201, 1 (2015), 90–111.

[5] E. Darve Cris Cecka, Adrian J. Lew. 2010. *Assembly of finite element methods on graphics processors*. https://doi.org/85(5):640âĂŞ669

[6] Yifeng Cui, Kim B Olsen, Thomas H Jordan, Kwangyoon Lee, Jun Zhou, Patrick Small, Daniel Roten, Geoffrey Ely, Dhabaleswar K Panda, Amit Chourasia, et al. 2010. Scalable earthquake simulation on petascale supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 1–20.

[7] E. Cuthill and J. McKee. 1969. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference (ACM '69)*. ACM, New York, NY, USA, 157–172.

[8] Florent De Martin. 2011. Verification of a Spectral-Element Method Code for the Southern California Earthquake Center LOH.3 Viscoelastic Case. *Bull. Seism. Soc. Am.* 101, 6 (2011), 2855–2865. https://doi.org/10.1785/0120100305

[9] P. F. Fischer and E. M. Rønquist. 1994. Spectral-element methods for large scale parallel Navier-Stokes calculations. *Comput. Methods Appl. Mech. Engrg.* 116 (1994), 69–76.

[10] Damien Genet, Abdou Guermouche, and George Bosilca. 2014. *Assembly Operations for Multicore Architectures Using Task-Based Runtime Systems*. Springer International Publishing, Cham.

[11] Dominik Göddeke, Dimitri Komatitsch, Markus Geveler, Dirk Ribbrock, Nikola Rajovic, Nikola Puzovic, and Alex Ramírez. 2013. Energy efficiency vs. performance of the numerical solution of PDEs: An application study on a low-power ARM-based cluster. *J. Comput. Physics* 237 (2013), 132–150.

[12] Dimitri Komatitsch, Jesús Labarta, and David Michéa. 2008. *A Simulation of Seismic Wave Propagation at High Resolution in the Inner Core of the Earth on 2166 Processors of MareNostrum*. Springer Berlin Heidelberg, Berlin, Heidelberg, 364–377.

[13] D. Komatitsch and J. Tromp. 2002. Spectral-Element Simulations of Global Seismic Wave Propagation-I. Validation. *Geophys. J. Int.* 149, 2 (2002), 390–412. https://doi.org/10.1046/j.1365-246X.2002.01653.x

[14] Martin Kronbichler and Katharina Kormann. 2012. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids* 63 (2012), 135 – 147.

[15] Filip KruÅijel and Krzysztof BanaÅŻ. 2013. Vectorized OpenCL implementation of numerical integration for higher order finite elements. *Computers & Mathematics with Applications* 66, 10 (2013), 2030 – 2044.

[16] Krzysztof BanaÅŻ and Filip KruÅijel and Jan BielaÅĎski. 2016. Finite element numerical integration for first order approximations on multi- and many-core architectures. *Computer Methods in Applied Mechanics and Engineering* 305 (2016), 827 – 848.

[17] Y. Maday and A. T. Patera. 1989. Spectral element methods for the incompressible Navier-Stokes equations. *State of the art survey in computational mechanics* (1989), 71–143.

[18] E Maufroy, E Chaljub, F Hollender, J Kristek, P Moczo, P Klin, E Priolo, A Iwaki, T Iwata, V Etienne, F De Martin, N. Theodulidis, M Manakou, C Guyonnet-Benaize, K Pitilakis, and P.-Y. Bard. 2015. Earthquake Ground Motion in the Mygdonian Basin, Greece: The E2VP Verification and Validation of 3D Numerical Simulation up to 4 Hz. *Bulletin of the Seismological Society of America* 105, 3 (2015), 1342–1364.

[19] Peter Moczo, Jozef Kristek, and Martin Gális. 2014. *The Finite-Difference Modelling of Earthquake Motions: Waves and Ruptures*. Cambridge University Press.

[20] A. T. Patera. 1984. A spectral element method for fluid dynamics: laminar flow in a channel expansion. *J. Comput. Phys.* 54 (1984), 468–488.

[21] Max Rietmann, Peter Messmer, Tarje Nissen-Meyer, Daniel Peter, Piero Basini, Dimitri Komatitsch, Olaf Schenk, Jeroen Tromp, Lapo Boschi, and Domenico Giardini. 2012. Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In *Proceedings of the ACM / IEEE Supercomputing SC'2012 conference*, Jeffrey K. Hollingsworth (Ed.). IEEE Computer Society Press, Salt Lake City, United States, article n 38. ISBN: 978-1-4673-0804-5.

[22] Daniel Roten, Yifeng Cui, Kim B. Olsen, Steven M. Day, Kyle Withers, William H. Savran, Peng Wang, and Dawei Mu. 2016. 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*. 957–968.

[23] Gauthier Sornet, Fabrice Dupros, and Sylvain Jubertie. 2017. A Multilevel Optimization Strategy to Improve the Performance of Stencil Computation. *Procedia Computer Science* 108 (2017), 1083 – 1092. International Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.

[24] Ricardo Taborda and Jacobo Bielak. 2011. Large-scale earthquake simulation: computational seismology and complex engineering systems. *Computing in Science & Engineering* 13, 4 (2011), 14–27.

[25] Loïc Thébault, Eric Petit, and Quang Dinh. 2015. Scalable and Efficient Implementation of 3D Unstructured Meshes Computation: A Case Study on Matrix Assembly. *SIGPLAN Not.* 50, 8 (Jan. 2015), 120–129. https://doi.org/10.1145/2858788.2688517

[26] Seiji Tsuboi, Kazuto Ando, Takayuki Miyoshi, Daniel Peter, Dimitri Komatitsch, and Jeroen Tromp. 2016. A 1.8 trillion degrees-of-freedom, 1.24 petaflops global seismic wave simulation on the K computer. *IJHPCA* 30, 4 (2016), 411–422.