Data-layout reorganization for an efficient intra-node assembly of a Spectral Finite-Element Method

Gauthier Sornet(1,2), Sylvain Jubertie(1), Fabrice Dupros(2),
Florent De Martin(2), Philippe Thierry(3), Sebastien Limet(1)
Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, France.(1)
BRGM, BP 6009, 45060 Orléans Cedex 2, France.(2)
Intel Corporation, Paris, France. (3)
Email: gauthier.sornet@gmail.com

***Abstract*—The Finite-Element Method (FEM) is routinely used to solve Partial Differential Equations (PDE) in various scientific domains. For seismic waves modeling, the Spectral Element Method (SEM), which is a specific formulation of the classical FEM approach, have gained significant attention for the last two decades. This is explained both from the very good numerical accuracy of this method and from the parallel performance of classical MPI-based implementations that scale up to several tens of thousands computing cores.**

Nevertheless, the trend for current processors with an increasing level of low-level parallelism requires significant efforts at the shared-memory level. One major bottleneck is coming from the standard FEM assembly phase that leads to significant amount of irregular memory accesses.

This prevents any efficient automatic optimizations from the compiler for instance. In this paper, we extract a kernel from a spectral-element application dedicated to earthquake simulations in complex geological medium (EFISPEC code developed at BRGM, the French Geological Survey). We study the intra-node behavior and we propose different levels of optimization (data-layout, manual vectorization, multi-threading) to fully benefit from SIMD units and NUMA architectures. Experiments performed on Intel Broadwell architecture show that the proposed optimizations dramatically improve the intra-node performance of the mini-application. Moreover, our results show a good match with rooflines theoretical performance models. We believe that these optimizations are not specific to this mini-application and may be implemented in different SEM and FEM based solvers as well.

I. Introduction

Physics-based three-dimensional numerical simulations are becoming more predictive and have already become essential in geosciences. In seismology, simulations with a very fine scale resolution are crucial for land-use planning. As a consequence, the large processing requirements of such methodologies is a challenge for the High Performance Computing community.

On one hand, the finite-element method is the cornerstone of many scientific softwares. For instance, the spectral finte-element method is widely used to solve the wave propagation problem due to its numerical efficiency, but a major bottleneck to scale out is the summation of the element contributions (assembly phase). This is due both from the shared values between neighboring elements and the inherent indirection for data accesses induced by the FEM.

One the other hand, the trend is to increase the complexity of computing node at the hardware level. Current applications should deal with multiple levels of hierarchical memories and increasing number of cores with larger SIMD registers. The increasing gap between the computing power and data transfers is a major issue. A careful attention is required to map threads and memory on such platforms. These evolutions must be taken into account to adapt and re-design current applications mainly based on a flat programming model.

In this paper, we study an extracted kernel from the EFISPEC code [1] developed at BRGM, the French Geological Survey. The paper proceeds as follows. Section II describes the related work. Section III details the elastodynamic equations and the spectral-element method implementation. Section IV introduces our methodology to optimize the computation at the shared-memory level. In sections V and VI, we introduce the experimental tests and discuss the results obtained with a focus on the peak performance.

II. Related Work

Current supercomputer nodes rely on shared-memory architectures and integrate many different levels of parallelism. At the core level, we have both the Instruction-Level Parallelism and the Data-Level Parallelism, the latter is provided through SIMD (Single Instruction on Multiple Data) units. Cores are assembled into multi-core processors which provide the Thread Level Parallelism. A single node may also integrate several multi-core processors to form a NUMA (Non-Uniform Memory Access) architecture.

Regarding parallel FEM assembly, the local computation requires to gather values of each element through an indirection table. This indirection reduces the efficiency of the cache, since non-contiguous data are likely to be accessed. Our experiments with several compilers also show that it prevents automatic compiler vectorization

even when adding SIMD pragmas. However, this phase may be performed in parallel since no concurrent writes occurs. This is not the case for the assembly phase where the element contributions are summed since elements shared values at their boundaries. Thus, optimizing this kernel to take advantage of current architectures, from the cache hierarchy to the different levels of parallelism, is not straightforward and the scientific literature dealing with this topic is abundant.

For instance, optimized implementations on GPU have been described in [2], [3], [4]. Most of these approaches implement mesh coloring strategy and fully benefit from the memory bandwidth available on the underlying architecture. At the shared-memory level, FEM implementations described in [5], [4], [6] underlines the impact of SIMD instructions and data-reuse at the cache memory level. Additionally, advanced algorithms described in [7] introduced a divide and conquer methodology to build a tree of dependent tasks. This cache-aware recursive procedure is implemented using Cilk multithreading library. Recent work [8] deals with a combination of optimizations to improve the performances of a spectral element kernel.

III. EFISPEC: Spectral finite element solver

A. Numerical background

The spectral-element method (SEM) appeared more than 20 years ago in computational fluid mechanics [9], [10], [11]. 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 $\square = \Lambda \times \Lambda \times \Lambda$.

The mapping from the reference cube to a hexahedral element Ω_e is done by a regular diffeomorphism $\mathcal{F}_e : \square \to \Omega_e$. In a finite-element method, the domain of study is discretized by subdividing its volume Ω into welded non-overlapping hexahedral elements Ω_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 (see Fig. 1). 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

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

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; $\ddot{\mathbf{u}}$ 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.

IV. Multi-level optimization strategies

A. Arithmetical intensity

To compute an element of order 4, there are $(4+1)^3 = 125$ GLL values to load with other parameters (GLL weights or Lagrange derivatives). It costs 8120 bytes to load but 48150 floating point operations to compute. Then, the Arithmetical Intensity of an order 4 computing element is $48150/8120 = 5.93$. This is a flops per loaded bytes factor such as a byte is involved into 5.93 flops.

B. Data-layout

Our kernel requires to transfer data between a global and a local representation by indirect memory access as shown in figure 1. The x86 architecture loads DRAM data by continuous aligned lines of 64 bytes. In the worst case, floats load 64 bytes and waste 60 unused bytes. Moreover, any byte stores corrupt a cache line for other threads such as they have to reload it. The situation is worsened on NUMA architecture where one needs to carefully control threads and data affinity. Finally, packing data to SIMD vector is costly.

We proceed as follows. First of all, we reorder the memory locations of the GLL values to make them contiguous. Then, as shown in Figure 1, we interleave pack of elements GLLs to build an indirection SIMD vector. We exploit dedicated intrinsics that can gather and scatter values from data arrays that follows SIMD indirection vectors. Finally, we interleave the other corresponding parameters of each GLL element. Thanks to this reorganization of the data-layout, we can tackle the vectorization.

C. Manual vectorization

Automatic compiler vectorization capabilities are limited as exposed in the paper [12]. As the Boost SIMD AVX512 library is not free, some low-level substitution functions have been implemented with intrinsics to abstract the architecture level (add, sub, mul ...). Thus, we decide to compute packs of elements with the same SIMD operations. It is made efficient thanks to the adapted

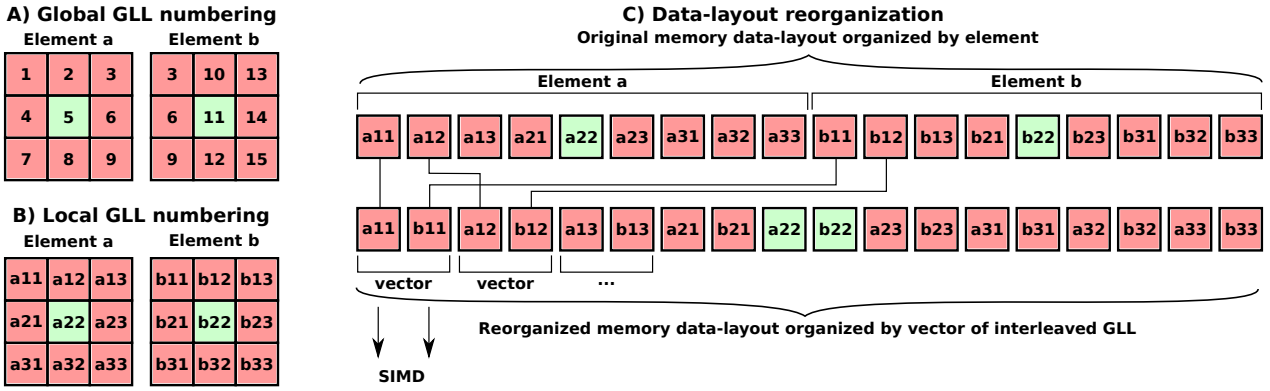

Figure 1: Memory-layout of the GLL points for two-dimensional order 2 elements.

data-layout described in the previous section. In fact, the required packed data can be directly loaded to the SIMD vectors. This way, instead of computing one by one element, it computes several elements at a time.

D. Multithreading and mesh coloring

Efficient exploitation of shared-memory platforms is not straightforward. As explained from the data-layout section, The threads should not share any line of the cache. Even worse, it must avoid any race conditions. To be efficient, we have to prevent starvation and synchronization's overhead. Moreover, the memory data localities have to match as much as possible with core affinities.

These warm points require a multi-level strategy. On the one hand, at low levels, atomic do not work with SIMD instructions. On the other hand, at high levels, the divide and conquer strategy described in [7] implies several costly modifications of the original application. Therefore, we implement a coloring strategy with a minimal number of synchronizations (one by color). We schedule as many blocks as threads by color to prevent threads starvation. Thus, a block of several elements SIMD vectors are computed by a same thread. It relies on OpenMP library and *parallel for* directives.

V. Experimental setup

We evaluate our strategy on a dual-socket platform with Intel Xeon E5-2697 v4 Broadwell processors, for a total of thirty-six cores at 2.3 Ghz (bi-socket with 2 numa-nodes by socket). In order to double-check our results, we use two compilers : Clang 5 and ICC 17 with `-O3 -march=native` optimization flags. Two versions of our kernel have been experimented for automatic compiler vectorization and for manual developper vectorization. A mesh of more than two million GLLs is used for our experiments based on order 4 approximation. We build a roofline model based on the SGEMM and the Stream benchmarks (figure 2). If we consider the peak performance provided by the SGEMM

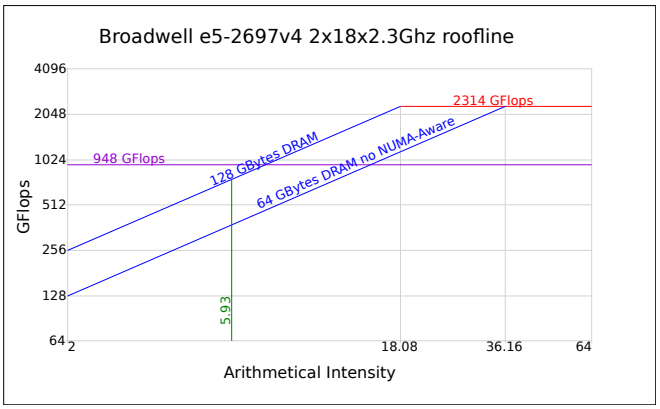

Figure 2: Roofline model on Broadwell platform.

routine, our kernel is memory-bound. Additionally, we have introduced a peak application performance. These values are computed with a small example able to fit in the cache-memory. In this case, we measure a peak value of 948 Glops on Broadwell.

The NUMA effect makes significantly lower the expected performance with an average of 400 Gflops on Broadwell.

VI. Experimental results

A. Compiler versus developper vectorization

First of all, a sequential implementation get a 5 times speedup from a manual explicit use of intrinsics. In fact, the compiler does not provide any vectorization. The multithreading results are summarized in the figure 3. Firstly, we can observe in poor blue peak performance and the limited scaling of implementations automatically vectorized by the compiler. We hardly achieve an average of 140 Gflops, this represents less than 5% of the theoretical peak performance. The manually vectorized version shows much better results with a maximum of 316 Gflops. This demonstrates the efficiency of our implementation and the performance measured are coherent with the theoretical values coming from the roofline model.

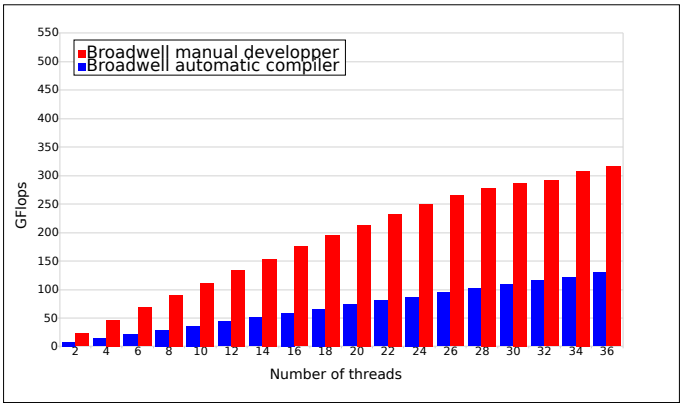

Figure 3: Comparison between automatic clang and manual developper vectorization on Broadwell platform for the irregular version using the Clang compiler.

The trend is very similar between Intel and Clang compilers. In fact, the performances of the automatic compiler (ICC17 and Clang5) SIMD implementations are respectively 123GFlops and 131GFlops. The manual developer SIMD implementations reach 218GFlops(Intel) and 316GFlops(Clang).

We can also observe the plot shape on Broadwell platform that underlines a saturation of the performance as we increase the number of cores. This behavior could be explained thanks to the roofline model and underline that the order 4 version is more likely to be memory bound on Broadwell.

VII. Conclusion and future work

We have identified the bottlenecks in the original EFISPEC code that prevent good intra-node scalability. Firstly, the standard MPI implementation may not allow a good scaling on multi-core and NUMA nodes, while showing good results on distributed architectures. We have discussed the performance of the parallel assembly phase. Secondly, mapping from global to local numbering requires gathering data from different memory locations, thus preventing a good usage of the memory bandwidth. Finally, auto-vectorization is hardly performed by the compilers, even when adding SIMD pragmas in the code ([12]).
For each of these limitations, we have proposed a solution and an implementation into a kernel extracted from EFISPEC. Significant gains have been reported on an AVX-2 platform. In fact, we have improved the overall peak performance of the assembly phase by a factor 2.3 on a Broadwell architecture. Moreover, the results obtained match with the theoretical performance given by the roofline models.
Several directions of improvements of these results could be considered since we do not reach the upper bound of the roofline model. For instance, we plan to implement a better strategy for threads and memory mapping on NUMA architecture. Moreover, back porting these optimizations into the full EFISPEC application is another major step of our work. This represents a challenge as the performance observe at the mini-app level could be dramatically reduced when the full application is considered [13]. Finally, a Domain Specific Language (DSL) could help to the physicists to focus only on the description of their algorithms.

References

[1] F. De Martin, "Verification of a spectral-element method code for the southern california earthquake center loh.3 viscoelastic case," *Bull. Seism. Soc. Am.*, vol. 101, no. 6, pp. 2855–2865, 2011.

[2] E. D. Cris Cecka, Adrian J. Lew, *Assembly of finite element methods on graphics processors*. 2010.

[3] M. Rietmann, P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini, "Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures," in *Proceedings of the ACM / IEEE Supercomputing SC'2012 conference* (J. K. Hollingsworth, ed.), (Salt Lake City, United States), p. article n 38, IEEE Computer Society Press, Nov. 2012. ISBN: 978-1-4673-0804-5.

[4] K. Banaś, F. Krużel, and J. Bielański, "Finite element numerical integration for first order approximations on multi- and many-core architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 305, pp. 827 – 848, 2016.

[5] D. Komatitsch, J. Labarta, and D. Michéa, *A Simulation of Seismic Wave Propagation at High Resolution in the Inner Core of the Earth on 2166 Processors of MareNostrum*, pp. 364–377. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[6] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-performance Tensor Contractions for GPUs," in *International Conference on Computational Science 2016 (ICCS 2016)*, vol. 80, pp. 108–118, 2016.

[7] L. Thébault, E. Petit, M. Tchiboukdjian, Q. Dinh, and W. Jalby, "Divide and conquer parallelization of finite element method assembly," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pp. 753–762, 2013.

[8] T. Ichimura, K. Fujita, P. E. B. Quinay, L. Maddegedara, M. Hori, S. Tanaka, Y. Shizawa, H. Kobayashi, and K. Minami, "Implicit nonlinear wave simulation with 1.08t dof and 0.270t unstructured finite elements to enhance comprehensive earthquake simulation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 4:1–4:12, ACM, 2015.

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

[10] Y. Maday and A. T. Patera, "Spectral element methods for the incompressible navier-stokes equations," *State of the art survey in computational mechanics*, pp. 71–143, 1989.

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

[12] 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.

[13] M. Christen, O. Schenk, and Y. Cui, "Patus for convenient high-performance stencils: evaluation in earthquake simulations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 11:1–11:10, IEEE Computer Society Press, 2012.