Accelerating SeisSol by Generating Vectorized Code for Sparse Matrix Operators

Alexander BREUER [a,1], Alexander HEINECKE [a], Michael BADER [a] and Christian PELTIES [b]

[a] *Department of Informatics, Technische Universität München, Germany*

[b] *Department of Earth and Environmental Sciences, Ludwig-Maximilians-Universität München, Germany*

Abstract

SeisSol is a software package for the simulation of seismic wave phenomena on unstructured grids, based on the discontinuous Galerkin method combined with ADER time discretization. A recent study shows that the intra-element performance of SeisSol is critical for its overall performance. Most of the element matrices are sparse operators implemented employing a standard sparse matrix storage scheme. Since their sparsity patterns are known a-priori we follow a different approach in this work. Here, we execute an enriched offline and initialization phase which is used to generate hardware-aware code, featuring optimal vectorization and removing indirect memory accesses by eliminating the index arrays and hard-wiring them into the code. Due to these optimizations we are able to run intra-element sparse matrix operations at up to 50% of achievable peak performance of an Intel Sandy Bridge core which results in speeding up SeisSol by a factor of more than 2.1 compared to the original implementation.

Keywords. SeisSol, ADER-DG, Code generation, Vectorization, Sparse-Matrix Multiplication

Introduction

The software package SeisSol is one of the leading codes for earthquake scenarios, in particular for simulating dynamic rupture processes and for problems that require discretizing very complex geometries. Spatial adaptivity in 3D is realized by flexible unstructured tetrahedral meshes. SeisSol uses the discontinuous Galerkin (DG) method for spatial and Arbitrary high order DERivatives (ADER) for time discretization. Due to its high degree of locality the ADER-DG method shows near-optimal speedups on supercomputing architectures [1,2,3].

We present a code generator for the inner kernel routines of SeisSol, where SeisSol spends most of the total computing time. These few kernel routines implement the

[1] Corresponding Author: Alexander Breuer, Department of Informatics, Technische Universität München, Boltzmannstr. 3, 85748 Garching bei München, Germany; E-mail: breuera@in.tum.de.

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

application of cell-local operators to multiple right-hand-side vectors, which are implemented as a sequence of matrix-matrix-multiplications: Element stiffness matrices, flux computation matrices, etc. (all sparse) are multiplied with a dense matrix that consists of only few columns (the right-hand-side vectors). The stiffness and flux matrices are typically of small size (35×35 for a fifth-order discontinuous Galerkin method), but with a strongly varying sparsity pattern. Because the sparsity patterns are known in the pre-compile phase, our code generator is able to exploit this knowledge and replaces the current kernels – using a sparse coordinate format at runtime – by a near-optimal implementation without index accesses, which shows significant speedups on state-of-the-art hardware compared to the original implementation.

1. Sparsity Patterns of the Kernel Operations

In this section, we describe the sparsity patterns of the ADER-DG method for the elastic wave equations. Please refer to [4,5] for a more detailed introduction in terms of both physical modeling and numerical discretization.

The elastic wave equations are the basic set of equations in SeisSol. In velocity-stress formulation the homogenous elastic wave equations are given by a quasi-linear hyperbolic system of partial differential equations:

$$q_t + A(x,y,z)q_x + B(x,y,z)q_y + C(x,y,z)q_z = 0, \tag{1}$$

where $q(t,x,y,z)$ is the nine-dimensional space-time-dependent vector of unknowns and A, B and C are the space-dependent Jacobians. Details about the extension to complex rheologies and dynamic rupture coupling of this set can be found in [3,6,7].

An ADER-DG method of convergence order $\mathscr{O}$ results in $B = \frac{1}{6}\mathscr{O}(\mathscr{O}+1)(\mathscr{O}+2)$ distinct basis functions in the discontinuous Galerkin approximation. Typical orders of production runs in SeisSol are five or six, corresponding to $B = 35$ or $B = 56$ basis functions. By applying the ADER-DG scheme for space and time discretization and using an orthogonal set of basis functions, we obtain from Eq. (1) an explicit updating scheme from time step t^n to t^{n+1} [4]:

$$\begin{aligned} Q_k^{n+1} = Q_k^n - \frac{|S_k|}{|J_k|} M^{-1} \Bigg(& \sum_{i=1}^{4} F^{-,i} I(t^n, t^{n+1}, Q_k^n) N_{k,i} A_k^+ N_{k,i}^{-1} \\ & + \sum_{i=1}^{4} F^{+,i,j,h} I(t^n, t^{n+1}, Q_{k(i)}^n) N_{k,i} A_{k(i)}^- N_{k,i}^{-1} \Bigg) \\ + M^{-1} K^{\xi} I(t^n, t^{n+1}, Q_k^n) A_k^* & \\ + M^{-1} K^{\eta} I(t^n, t^{n+1}, Q_k^n) B_k^* & \\ + M^{-1} K^{\zeta} I(t^n, t^{n+1}, Q_k^n) C_k^*. & \end{aligned} \tag{2}$$

In detail the individual matrices in Eq. (2) are given by:

- Q_k^{n+1}, Q_k^n – Unknowns for each basis function at time t^{n+1} and t^n in tetrahedron k: $B \times 9$

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

- $N_{k,i}A_k^+N_{k,i}^{-1}$, $N_{k,i}A_{k(i)}^-N_{k,i}^{-1}$ – Flux solver for element k and face i: 9×9
- A_k^*, B_k^*, C_k^* – Linear combinations of the Jacobians in tetrahedron k with respect to the corresponding transformation to the reference tetrahedron: 9×9
- K^ξ, K^η, K^ζ – Stiffness matrices over the reference tetrahedron: $B \times B$
- $F^{-,i}$, $F^{+,i,j,h}$ – Flux matrices over the reference tetrahedron: $B \times B$
- M^{-1} – Inverse of the diagonal mass matrix over the reference tetrahedron; stored implicitly as part of the flux and stiffness matrices: $B \times B$

$|J_k|$ and $|S_k|$ are scalars stored implicitly as part of the flux solver, and account for the transformation to the reference element and the corresponding transformation of the elements boundary.

The operator $I(t^n, t^{n+1}, Q_k^n)$ in Eq. (2) represents the ADER time integration:

$$I(t^n, t^{n+1}, Q_k^n) = \sum_{j=0}^{\mathcal{O}-1} \frac{(t^{n+1} - t^n)^{j+1}}{(j+1)!} \frac{\partial^j}{\partial t^j} Q_k(t^n), \tag{3}$$

where the time derivatives are defined via a recursive scheme with initial values $\frac{\partial^0}{\partial t^0} Q_k = Q_k^n$ [6]:

$$\frac{\partial^{j+1}}{\partial t^{j+1}} Q_k = -M^{-1} \left((K^\xi)^T \left(\frac{\partial^j}{\partial t^j} Q_k \right) A_k^* + (K^\eta)^T \left(\frac{\partial^j}{\partial t^j} Q_k \right) B_k^* + (K^\zeta)^T \left(\frac{\partial^j}{\partial t^j} Q_k \right) C_k^* \right). \tag{4}$$

Again the inverse mass matrix is stored implicitly as part of the transposed stiffness matrices $(K^\xi)^T$, $(K^\eta)^T$ and $(K^\zeta)^T$.

All of the involved matrices are of small rank and – except for the matrix of unknowns Q_k^n, the time integrated unknowns matrix $I(t^n, t^{n+1}, Q_k^n)$ and some matrices appearing in the flux computation – all matrices are sparse. The sparsity patterns of the matrices are known in the pre-compile phase and are independent of the actual tetrahedral mesh – except for special cases, where the faces are aligned to the physical *xyz*-coordinate system. In the following, we describe the sparsity patterns of the involved matrices in Eqs. (2) and (4):

ADER time integration: The crucial part of the ADER time integration – Eqs. (3) and (4) – is the recursive scheme (4), where we multiply the sparse transposed stiffness matrices $(K^\xi)^T$, $(K^\eta)^T$ and $(K^\zeta)^T$ (size $B \times B$) to the dense derivatives $\partial^j / \partial t^j Q_k$ (size $B \times 9$). Then we multiply the dense intermediate results of the sparse-dense multiplication with the sparse "Jacobians" A_k^*, B_k^* and C_k^* (size 9×9). Figure 1 shows all involved sparsity patterns of the ADER time integration for a fifth order ($B = 35$) scheme.

Volume integration: In the volume integration, given by the last three terms of Eq. (2), we multiply the sparse stiffness matrices K^ξ, K^η and K^ζ to the dense time integrated unknowns $I(t^n, t^{n+1}, Q_k^n)$ – previously obtained by the ADER time integration. The intermediate results are again multiplied by the sparse "Jacobians" A_k^*, B_k^* and C_k^*. Except for the non-transposed stiffness matrices, the sparsity pattern of the volume integration is identical to the ADER time integration, illustrated in Fig. 1.

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

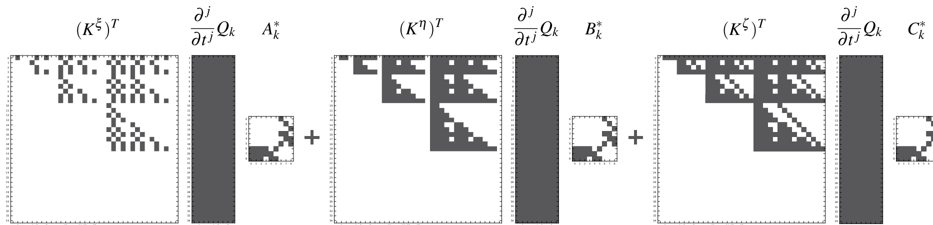

Figure 1. Sparsity patterns appearing in the ADER time integration for a fifth order method.

Boundary integration: The boundary integration is given by the two sums over the four tetrahedral faces in Eq. (2). The first sum is the contribution of the element itself to the numerical flux over the boundary: We multiply the sparse flux matrices $F^{-,i}$ (size $B \times B$, i is the i-th face) to the dense time integrated unknowns – obtained by the time integration again. The dense intermediate results are multiplied by the dense flux solver $N_{k,i} A_k^+ N_{k,i}^{-1}$ (size 9×9) afterwards. Fig. 2 shows all involved matrix patterns for a fifth-order method ($B = 35$).

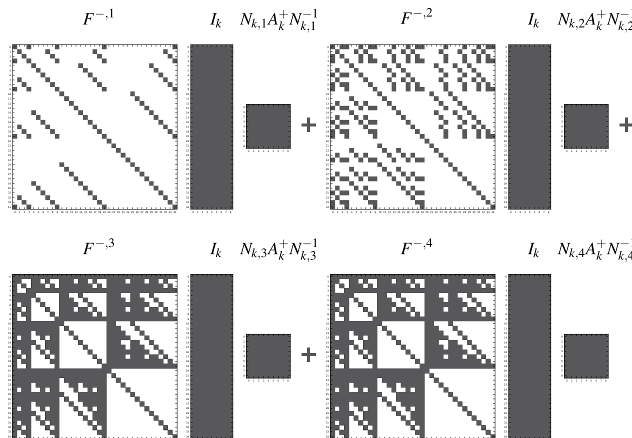

Figure 2. Sparsity patterns appearing in the current element's contribution to the boundary integration for a fifth order method.

The contribution of the neighboring elements to the numerical flux, given by the second sum of Eq. (2), is mathematically similar to the contribution of the element itself. The major difference are the 48 different flux matrices $F^{+,i,j,h}$ appearing throughout a simulation. Here the tuple (i, j, h) represents the different relations an element can have to its neighboring element: The index $i \in \{1,2,3,4\}$ accounts for the four different faces of the element itself, $j \in \{1,2,3,4\}$ for the corresponding face of the neighboring element and $h \in \{1,2,3\}$ for the possible vertex combinations of two neighboring faces. The sparsity patterns of these flux matrices are highly varying and can even be near-dense as the examples in Fig. 3 show.

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

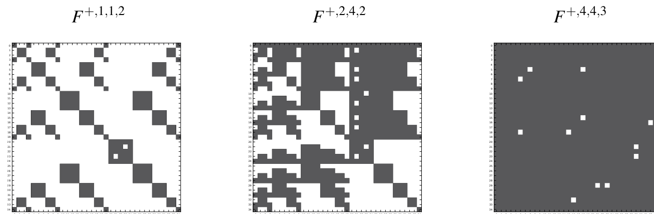

Figure 3. Examples of sparsity patterns appearing in the contributions by neighbor elements to the flux computation for a fifth order method.

2. Optimized Matrix Routines

In order to achieve high computational performance of the implementation of the time, volume and boundary integration, we have to make sure that the overhead stemming from handling sparse-matrix structures is as low as possible. We will compare our optimizations to the current implementation in SeisSol, which uses a plain coordinate format for accessing elements of the sparse matrix [6]. Since the matrices are rather small ($B = 35$ in case of fifth order) calling optimized library functions is not suitable as their high performance is shaded by calling and formatting overheads [8].

However, since the sparsity patterns described in Sect. 1 are statically known, we can generate code exactly matching our needs before calling the routines: By performing a dry run of all required matrix operations appearing in the different integrations, we are able to derive source code that is entirely free of any indexing arrays for navigating through the sparse matrices. This generated code is included by the pre-processor into SeisSol before compilation.

2.1. Code Generation of Matrix Kernels

Given Figs. 1–3 it becomes clear that we need to optimize two different kinds of matrix operations: *sparse* $\times$ *dense* $=$ *dense* and *dense* $\times$ *sparse* $=$ *dense*. We will now discuss details how we can implement these operators with respect to the data layout used in SeisSol in a way that the new kernels can easily be used with minimal impact to the remaining source code. Since SeisSol is written in FORTRAN, a column-major layout for dense matrices is used. Note that we exploit the matrix sparsity patterns known at compile time and therefore eliminate the usually required indexing arrays of sparse matrix computing by hard-wiring them into our code. As a consequence, this allows us to vectorize element accesses that could not be vectorized by the compiler's auto-vectorizer or by source-to-source pre-compilers like Scout [9], as these tools regard the index arrays as variables specifying offsets in the data and therefore fail on automatic vectorization.

Executing such routines on a modern architecture results in an L1-cache (L1$) bandwidth-bound performance. During the design phase of the code-generation process, we focused on the x86 vector extension AVX, which offers 128- and 256-bit-wide instructions, and was introduced with Intel Sandy Bridge (SNB). Note that SNB supports L1$ reads of 32 bytes and L1$ writes of 16 bytes per cycle, which means that we can only exploit 128-bit vector instructions efficiently. The required increase to 64 bytes read and 32 bytes write bandwidth is implemented in Intel's next-generation architecture code-named Haswell [10]. Based on experiments with different applications and kernels using

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

the 256-bit vector instructions on Sandy Bridge, the full performance benefit of a 2× speed-up (when switching from 128- to 256-bit vector instructions) can only be reached for kernels which can be perfectly register-blocked, e.g. DGEMM, see [11]. If in contrast only a standard 1d-blocking is possible, roughly a 1.5-1.6× speed-up can be achieved in comparison to AVX-128, see [12]. Thus, kernels performing *sparse* × *dense* = *dense* and *dense* × *sparse* = *dense* should be executed using AVX with 128-bit width for best performance, since they are L1$ bandwidth bound. Algorithms 1 and 2 illustrate the generated code for our *sparse* × *dense* = *dense* and *dense* × *sparse* = *dense* operators:

dense × *sparse* = *dense:* This is the easier case when dealing with dense matrices in a column-major storage scheme. As shown in Algorithm 1, it can be perfectly vectorized along the columns of A[2] and C: Based on the sparsity pattern of B we can completely unroll the multiplication of row m of A with the whole matrix B, which results in obtaining row m of C. Since all elements in a column of A and C are stored contiguously, we can fuse the calculation of several rows and therefore achieve a perfect vectorization.

sparse × *dense* = *dense:* In contrast to the *dense* × *sparse*-operator, this matrix multiplication can only be vectorized optimally, if all dense matrices are in row-major storage. However, on-demand transposition is too costly due to the limited size of these matrices. Therefore, we extended our code generation by a vectorization step which groups contiguous elements in the sparse columns of A and C into vector instructions as shown in Algorithm 2. Since we only employ 128-bit wide vector instructions this is a rather easy task, as such a group consists only of two elements. However, we made sure that our code generator is upwards compatible, e.g. for targeting Intel Haswell: Generating full 256-bit code with grouping of four or three (through appropriate masking) elements is already possible.

Algorithm 1 Vectorization kernel of the *dense* × *sparse* = *dense* operation. This routine is called by the vectorizer for each row m of A and C. lda and ldc denote the leading dimensions of A and C. B is stored in a virtual compressed column format (CSC) with row_B and col_B being index arrays describing the positions of matrix entries of B which are stored contiguously.

$m \leftarrow$ current row of A and C
for all $n \in$ #col of B **do**
  $b_{\#\text{Elements}} \leftarrow \text{col}_B[n+1] - \text{col}_B[n]$
  for $k = 0$ to $b_{\#\text{Elements}} - 1$ **do**
    Generate Instruction:
    $C[n \cdot \text{ldc} + m] += A[\text{row}_B[\text{col}_B[n] + k] \cdot \text{lda} + m] \cdot B[\text{col}_B[n] + k]$
  end for
end for

2.2. Performance Evaluation of Generated Code

We evaluated the performance of SeisSol using the generated kernel routines on the "CoolMAC" system installed at Leibniz-Supercomputing Centre[3]. This system features

[2]We are using BLAS-standard identifiers in this work.
[3]http://www.mac.tum.de/wiki/index.php/MAC_Cluster

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

Algorithm 2 Vectorization kernel of the *sparse* $\times$ *dense* $=$ *dense* operation. ldb and ldc denote the leading dimensions of B and C. A is stored in a virtual compressed column format (CSC) with row_A and col_A being index arrays describing the positions of matrix entries of A which are stored contiguously.

for all $n \in$ #col of B **do**
  for all $k \in$ #col of A **do**
    $a_{\#\text{Elements}} \leftarrow \text{col}_A[k+1] - \text{col}_A[k]$
    for $m = 0$ to $a_{\#\text{Elements}} - 1$ **do**
      if $m < a_{\#\text{Elements}} - 1$ **and** $\text{row}_A[\text{col}_A[k]+m]+1 = \text{row}_A[\text{col}_A[k]+m+1]$ **then**
        Generate 2 Element Vector Instruction starting at:
        $C[n \cdot \text{ldc} + \text{row}_A[\text{col}_A[k]+m]] += A[\text{col}_A[k]+m] \cdot B[n \cdot \text{ldb} + k]$
        $m \leftarrow m+1$
      else
        Generate Scalar Instruction:
        $C[n \cdot \text{ldc} + \text{row}_A[\text{col}_A[k]+m]] += A[\text{col}_A[k]+m] \cdot B[n \cdot \text{ldb} + k]$
      end if
    end for
  end for
end for

nodes equipped with two Intel SNB eight-core Xeon E5-2670 processors running at 2.6 GHz. Therefore one core of this system offers a theoretical peak performance of 20.8 GFLOPS (double precision). Since we have to keep the limited L1$ bandwidth in mind, the peak performance of L1$-bandwidth-limited codes halves to 10.4 GFLOPS.

As reference problem we solved the LOH.1 benchmark, see [4], with a total number of 386,518 elements. In this case the LOH.1 benchmark requires the handling of 12,658 free-surface and 9,022 absorbing faces in the boundary integration kernel accounting for numerical boundary conditions. Free-surface, absorbing and periodic boundary conditions are natively supported by the new kernels, while dynamic-rupture boundary conditions are supported via a fall-back solution to the classical code.

We utilized a complete node of the cluster by distributing the mesh across the 16 SNB cores of a single-node with 16 MPI-partitions generated by Metis [13]. All of the following measurements have been been obtained on a complete node, even if single-core results are shown for illustrative purposes. Each of the measurements was performed twice and all results averaged over the corresponding two runs. The number of floating point operations has been obtained theoretically via the equations given in Section 1 and verified against an optional counter directly in the code.

Figure 4 shows the single-core performance results of the individual kernels for $B =$ 4, 10, 20, 35 and 56 basis functions ($\mathcal{O}$ = 2, 3, 4, 5 and 6) in SeisSol. The timings have been obtained from 1,000 time steps (386,518,000 samples) for each kernel. In general we observe an increasing performance of the kernels for a growing number of basis functions, which is due to the increasing computational intensity. Depending on the different kernels and order of approximation, we are able to obtain up to 50% of the achievable peak performance. On average we achieve roughly 3 GFLOPS across all kernels. However, we see a noticeable performance drop of the boundary kernel for 35 and 56 basis functions which can be explained by an exceeding of the instruction

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

cache with the high number of unique instructions generated for the 52 different matrices, which are partly near dense as shown in Figure 3.

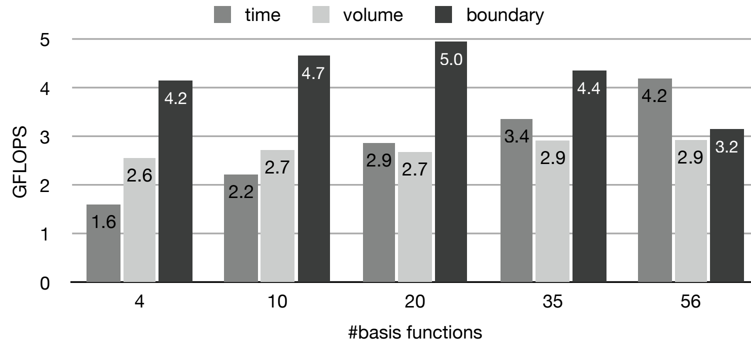

Figure 4. Performance of the time, volume and boundary integration kernel in SeisSol.

As illustrated in Figure 5, the kernels mainly responsible for the overall performance of SeisSol are time and boundary integration. The relative computational cost of the time integration increases with the order of the scheme, because of the increasing number of summands in Equation 3.

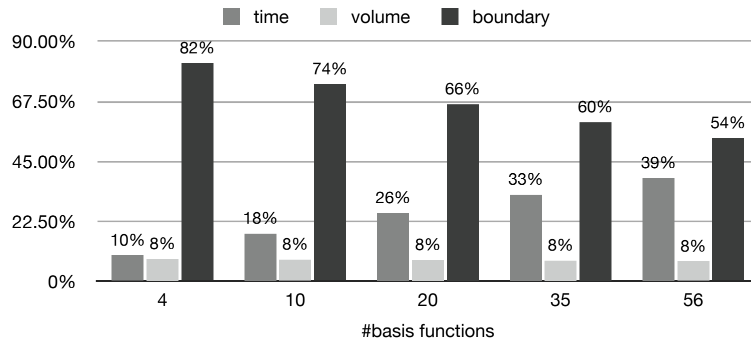

Figure 5. Distribution of floating point operations in the time, volume and boundary integration kernel.

Taking into account the results from Figs. 4 and 5, we can see that the time integration performs well for high approximation order, but unfortunately the decreasing speed of the boundary integration reduces the overall performance, because most of the flops are still spent here. As stated earlier, we are currently using our sparse matrix multiplication generator for handling even the nearly dense matrices in the boundary handling. This causes significant performance penalties, since optimization techniques such as register blocking [14,15] cannot be exploited.

We would like to close our performance study by comparing timings of an entire LOH.1-simulation in SeisSol. In contrast to the results given in Figure 4, we measured the complete time-marching loop of the simulation with an included output of 40 receivers in the computational domain at 900 points in time. We used the receiver output to verify the results of the new kernels against the classical code and obtained negligible derivations in the order of machine precision. Figure 6 shows the overall performance of

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

the kernel generation approach in comparison to the classical code. Again only the floating point operations in the kernels are considered in the presented performance results. On average we see a speed-up of roughly 2.2 across all orders of approximation, which is the optimal and expected result: Keeping in mind the L1$ load and store limitations of SNB-EP, we are employing the AVX-128 vector instruction set which allows processing of two element vectors. The reason for being slightly faster than 2× can be found in the saved handling of index arrays in comparison to the original coordinate-based implementation.

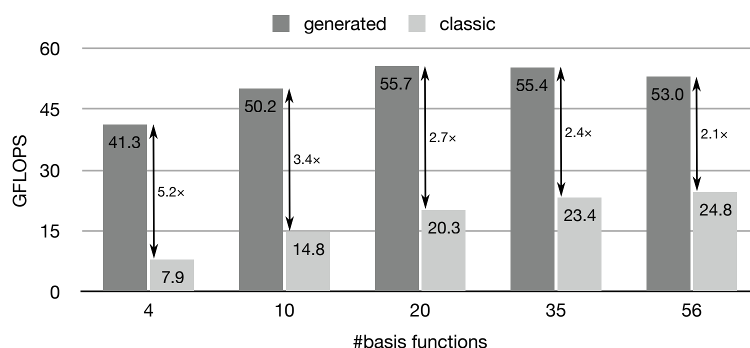

Figure 6. Performance of the generated and classical kernels in a complete LOH.1 simulation.

3. Conclusion and Future Work

In this paper, we presented an optimization strategy based on code generation for element-local sparse matrix operations for the SeisSol software package. Since the sparsity patterns of these operators are known a-priori, the code generation phase materializes as an offline phase before compiling SeisSol and eliminates all indirect matrix element accesses during the elements processing. We evaluated the performance of this new methodology on a dual-socket Intel Sandy Bridge node (16 cores) exploiting AVX-128 instructions and obtained a near-optimal speed-up of 2.

During the performance discussion we identified the handling of the boundary integration as a potential bottleneck, because nearly dense A matrices may occur. Such operators will perform much better when switching to a dense storage scheme. However, due to the small size of the matrices, calling highly efficient BLAS functions from vendor implementations is not a good option. Therefore, we plan to implement customized dense kernels following the principles proposed in [15,14] in conjunction with an autotuned switch that determines whether a generated dense kernel or sparse kernel should be called for a particular element operation. Note that this switch might also help to speed up other kernels. From a hardware perspective, we plan to support IBM's BlueGene/Q and Intel's Xeon Phi in near future in the code generation phase and to evaluate our generated code on desktop Haswell silicon in order to examine whether the improved L1$ bandwidth fixes the issues discussed in this paper.

In upcoming versions of our code generator we will also support viscoelastic attenuation in SeisSol. Numerically, we have to extend the current nine-dimensional vector

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25
©IOS Press 2014

of unknowns q in Eq. (1) by $6m$ unknowns, where m is the number of used viscoelastic mechanisms [6]. Another important extension is the native support of dynamic rupture boundary conditions, which can be time consuming due to highly refined meshes in the area of the fault system.

All preprocessing scripts and kernels used in the presented work are actively developed and freely available[4] under an open-source license.

References

[1] M. Käser, C. Pelties, C.E. Castro, H. Djikpesse, and M. Prange. Wavefield modeling in exploration seismology using the discontinuous Galerkin finite-element method on HPC infrastructure. *The leading Edge*, 29:76–85, 2010.

[2] M. Käser, C. Castro, V. Hermann, and C. Pelties. SeisSol–A Software for Seismic Wave Propagation Simulations. *High Performance Computing in Science and Engineering*, pages 281–292, 2010.

[3] Josep de la Puente, Martin Käser, Michael Dumbser, and Heiner Igel. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes–IV. Anisotropy. *Geophysical Journal International*, 169(3):1210–1228, 2007.

[4] M. Dumbser and M. Käser. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – II. The three-dimensional isotropic case. *Geophysical Journal International*, 167(1):319–336, 2006.

[5] Martin Käser and Michael Dumbser. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – I. The two-dimensional isotropic case with external source terms. *Geophysical Journal International*, 166(2):855–877, 2006.

[6] M. Käser, M. Dumbser, J. de la Puente, and H. Igel. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes – III. Viscoelastic attenuation. *Geophysical Journal International*, 168(1):224–242, 2007.

[7] Christian Pelties, Josep de la Puente, Jean-Paul Ampuero, Gilbert B. Brietzke, and Martin Käser. Three-dimensional dynamic rupture simulation with a high-order discontinuous Galerkin method on unstructured tetrahedral meshes. *Journal of Geophysical Research: Solid Earth*, 117(B2):2156–2202, 2012.

[8] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. High performance smart expression template math libraries. In *High Performance Computing and Simulation (HPCS)*, pages 367–373, 2012.

[9] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and WolfgangE. Nagel. Scout: A Source-to-Source Transformator for SIMD-Optimizations. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 137–145. Springer Berlin Heidelberg, 2012.

[10] T. Piazza and H. Jiang and P. Hammarlund and R. Singhal. Technology Insight: Intel(R) Next Generation Microarchitecture Code Name Haswell, September 2012.

[11] Alexander Heinecke and Carsten Trinitis. Cache-oblivious Matrix Algorithms in the Age of Multi- and Many-Cores. *Concurrency and Computation: Practice and Experience*, January 2013. accepted for publication.

[12] Alexander Heinecke and Dirk Pflüger. Emerging Architectures Enable to Boost Massively Parallel Data Mining using Adaptive Sparse Grids. *International Journal of Parallel Programming*, 41(3):357–399, 2013.

[13] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[14] Field G. Van Zee and Robert A. van de Geijn. BLIS: A Modern Alternative to the BLAS. *ACM Transactions on Mathematical Software*, 2013. submitted.

[15] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, 2008.

[4] `https://github.com/TUM-I5/seissol_kernels`

The final version of this article has been published in:
M. Bader et al. (Eds.): Parallel Computing – Accelerating Computational Science and Engineering (CSE)
Advances in Parallel Computing 25

©IOS Press 2014