Optimization of the Sparse Matrix-Vector Products of an IDR Krylov Iterative Solver in EMGeo for the Intel KNL Manycore Processor

Tareq Malas(✉), Thorsten Kurth, and Jack Deslippe

National Energy Research Scientific Computing Center,
Lawrence Berkeley National Laboratory, Berkeley, USA
tmalas@lbl.gov

Abstract. In geophysical-imaging, medium properties can be studied by performing scattering experiments using electromagnetic or seismic waves. Quantities such as densities, elasticities, stress etc. can be obtained from fitting the observed measurements to the results predicted by a simulation. The EMGeo software performs these simulations and solves the inverse scattering problem in the Laplace-Fourier domain. In this paper, we focus on the Seismic part and forward step of the inverse scattering problem, which involves inverting a large sparse matrix. For this purpose, EMGeo uses an Induced Dimensional Reduction (IDR) Krylov subspace solver. The Sparse Matrix Vector (SpMV) product is responsible for more than half of the total runtime. We demonstrate how we use spatial and multiple Right Hand Side (RHS) blocking cache optimizations to increase arithmetic intensity and thus the performance, as SpMV product is memory bandwidth-bound. Our optimizations achieve 5.0× and 4.8× speedup in the SpMV product in Haswell and KNL processors, respectively. We also achieve 1.8× and 3.3× speedup in the overall IDR solver in Haswell and KNL processors, respectively. We also give an outlook over possible future optimizations.

Keywords: Intel knights landing optimization · Matrix vector product optimization · IDR Krylov solver optimization · Multiple right-hand side blocking · Spatial blocking

1 Introduction

Problem Description: EMGeo is a seismic tomography software which infers the composition of the ground using electromagnetic (EM) and seismic scattering information. It is a 3D full waveform inversion scheme for elastic wave propagation in the Fourier domain. The EM and seismic parts are very similar as the system has the same number of unknowns. However, the latter is more involved as the propagating waves have longitudinal as well as transversal polarized components which are tightly coupled through properties of the medium. In this presentation, we are going to present improvements for the seismic problem in the

© Springer International Publishing AG 2016
M. Taufer et al. (Eds.): ISC High Performance Workshops 2016, LNCS 9945, pp. 378–389, 2016.
DOI: 10.1007/978-3-319-46079-6_27

forward step of the computation. This step requires solving large-scale implicit linear systems in the frequency domain. EMGeo mitigates that by avoiding performing a brute-force forward inversion of the transfer matrix which describes the medium, but instead inverting on a set of representative vectors using Krylov iterative solvers. The workflow is as follows: the general objective of EMGeo is to minimize the cost functional

$$\phi(\mathbf{m}) = \frac{1}{2}\sum_{s_k}\sum_{q}\left\|\mathbf{E}\left[\mathbf{d}_q^{\mathrm{obs}}(s_k) - \mathbf{d}_q^{\mathrm{sim}}(\mathbf{m}, s_k)\right]\right\|^2 + \frac{\lambda}{2}\left\|\mathbf{W}\mathbf{m}\right\|^2. \quad (1)$$

Here, $\mathbf{d}_q^{\mathrm{obs}}(s_k), \mathbf{d}_q^{\mathrm{sim}}(\mathbf{m}, s_k)$ are observed and predicted signals at position q, $\mathbf{W}$ a regularization matrix, λ a Lagrange parameter to avoid overfitting, $\mathbf{E}$ is a (diagonal) error matrix which includes uncertainties in the measured data and $\mathbf{m}$ are the model parameters. Furthermore, s_k denotes a 'frequency' along with a dampening term, i.e. $s_k = \sigma_k + i\,\omega_k$, used in the Laplace transformation of the time-dependent fields $\mathrm{d}_q^{\mathrm{obs}}(t)$. The expensive part in expression (1) is the computation of the simulated response, i.e. $\mathbf{d}_q^{\mathrm{sim}}(\mathbf{m}, s_k)$. It is obtained by interpolating the velocity field $\mathbf{v}$ of the seismic waves propagating through the medium, i.e. $\mathbf{d}_q^{\mathrm{sim}}(\mathbf{m}, s_k) = \hat{\mathbf{G}}\mathbf{v}_q(\mathbf{m}, s_k)$. The velocities are in turn computed by solving a sparse linear system which can be discretized with finite differences (FD) on a staggered grid (c.f. [8,9] for more details):

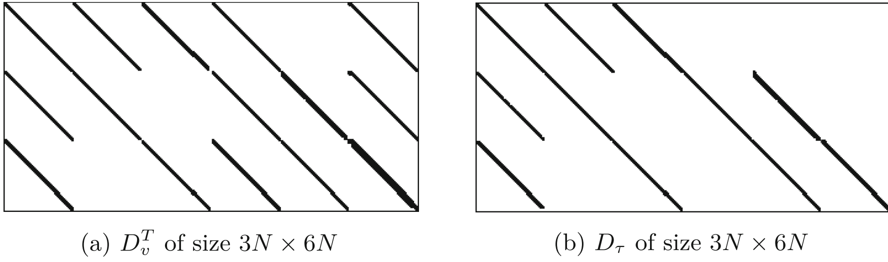
$$\left[1 - \langle\mathbf{b}\rangle\mathbf{D}_\tau \cdot (\langle\mathbf{k}\mu\rangle \circ \mathbf{D}_v)\right]\mathbf{v}_q = \mathbf{f}_q, \quad (2)$$

where $\circ$ denotes the Hadamard product, $\langle\mathbf{k}\mu\rangle$ and $\langle\mathbf{b}\rangle$ are block matrices of averaged elastic parameters which describe the medium and $\mathbf{D}_\tau, \mathbf{D}_v$ are block-matrices of FD operators. Furthermore, $\mathbf{v}_q$ is the velocity vector of interest and $\mathbf{f}_q$ the source vector in the Laplace-Fourier domain. Both (super-)vectors are in structure-of-array form, i.e. $\mathbf{g} = (\mathbf{g}_x, \mathbf{g}_y, \mathbf{g}_z)^T$ for $\mathbf{g} = \mathbf{v}$ or $\mathbf{g} = \mathbf{f}$ respectively. The components $\mathbf{g}_i$ contain all i-components of the respective field for all grid points in x, y, z-major order.
The matrices in (2) can then be written as follows [9]:

$$\mathbf{D}_\tau = \begin{pmatrix} \tilde{D}_x & D_y & D_z & \tilde{D}_x & 0 & \tilde{D}_x \\ \tilde{D}_y & D_x & 0 & \tilde{D}_y & D_z & \tilde{D}_y \\ \tilde{D}_z & 0 & D_x & \tilde{D}_z & D_y & \tilde{D}_z \end{pmatrix}^T \;;\; \mathbf{D}_v = \begin{pmatrix} D_x & \tilde{D}_y & \tilde{D}_z & 0 & 0 & 0 \\ 0 & \tilde{D}_x & 0 & D_y & \tilde{D}_z & 0 \\ 0 & 0 & \tilde{D}_x & 0 & \tilde{D}_y & D_z \end{pmatrix}. \quad (3)$$

Here, D_i and $\tilde{D}_i$ denote FD operators for direction i. In the following, we will denote $\langle\mathbf{b}\rangle\mathbf{D}_\tau$ by D_τ and $\langle\mathbf{k}\mu\rangle \circ \mathbf{D}_v)$ by D_v. This is just for brevity as multiplying the medium dependent factors do not change the sparsity pattern of these matrices as long as they are not zero (which is usually true). Figure 1 depicts the sparsity pattern of these two matrices, where N is the number of total grid points.

Challenges: There are two challenging aspect in this calculations. First, the SpMV product in (2) needs to be optimized, as it amounts to two thirds of the time spent in the linear solver. However, SpMV operation is notoriously memory bandwidth-bound as its arithmetic intensity is low. In Sect. 2, we explain how we address this



(a) D_v^T of size $3N \times 6N$ (b) D_τ of size $3N \times 6N$

Fig. 1. The sparsity pattern of D_v and D_τ matrices

issue with cache blocking. Second, global reductions and halo exchanges need to be optimized in order to improve strong scaling of our code. This can in part be done by solving (2) for multiple right hand sides and overlapping communication and computation in a clever way, which is left for future work.

Hardware: We use two systems for our performance measurements. The first one is the HPC system Cori Phase I at NERSC, which features 1630 nodes with 128 GiB DDR memory, Cray Aries interconnect and two Xeon® E5 CPUs per node. The Intel Xeon® E5, also termed Haswell, micro-architecture is a 22 nm fabric with support for AVX2. The NERSC Cori Phase I system uses revision E5-2698, which is comprised of 16 physical cores which can host up to 2 threads per core. It achieves 2.3 GHz in sustained and 3.6 GHz in turbo mode. It is further equipped with 3 cache levels, where L1 and L2 caches are of size 64 kiB and 256 kiB respectively. The L3 cache is shared between all cores on a physical CPU and of 40 MiB in size. The theoretical peak DDR memory bandwidth of this architecture is 68 GB/s per socket. The sockets are connected via Intel Quickpath Interconnect® with 9.6 GT/s, which translates to an effective memory bandwidth of ~19.6 GB/s.

The second system we are considering is the new Intel Knights Landing (KNL) CPU, which is a second generation processor from the Intel Xeon Phi® family. We are using B0 stepping in revision 7210, which features 64 improved Silvermont® cores with 1.3 GHz clock rate, improved out-of-order processing as well as up to four hyperthreads per core. The cores are connected in a two-dimensional mesh of tiles, where each tile is comprised of two cores with 1 MiB L2 cache and two AVX512-enabled vector units each. Furthermore, KNL features 16 GiB MCDRAM, which is a high-bandwidth on-package memory with transfer-rates of up to 430 GB/s. The additional DDR memory, of which there are 96 GB in our setup, can be accessed at about 90 GB/s. The KNL processor mesh as well as the memory can be configured in different ways: the processor can be partitioned into 1, 2 or 4 partitions referred to as *Quadrant*, *SNC2* and *SNC4* mode respectively. The *SNC2* and *SNC4* abbreviations stand for *sub-NUMA clustering* and allow for fine-grained control over thread binding and thus possibly to a mitigation of memory access latency. The different memory configuration options are referred to as *Cache*, *flat* and *hybrid*. In *Cache* mode,

MCDRAM acts as a huge L3 cache and thus cannot be addressed manually by the user. In the *flat* model case, MCDRAM and DDR work side by side and the user has to manually allocate memory in either one. The *hybrid* mode allows for mixing the first two options, i.e. by assigning 50 % or 75 % of the MCDRAM to work as DDR cache, and the remaining fractions can be addressed manually by the user.

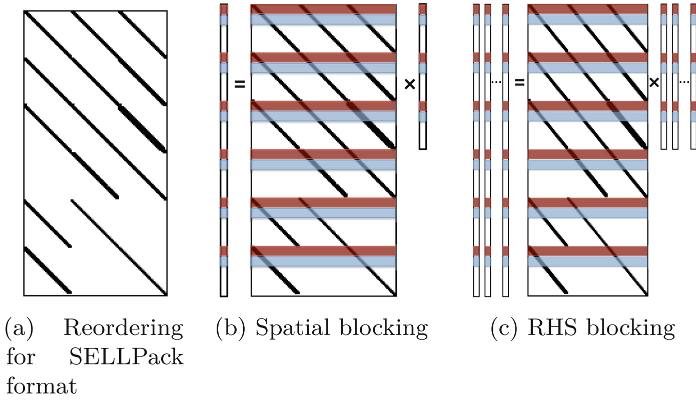
2 Approach

We apply techniques to reduce the memory traffic and increase the in-cache data reuse in the SpMV product of EMGeo code. We replace the ELLPack sparse matrix format, which is used in EMGeo code, with Sliced ELLPack (SELLPack) format to reduce the FLOPS and memory transfers. We also apply cache blocking techniques to increase the SpMV product operation *Arithmetic Intensity* (AI). Namely, we use Spatial Blocking (SB) and multiple Right Hand Side (mRHS) cache blocking.

EMGeo uses ELLPack data format because the maximum number of Non-Zero (NNZ) elements in each row is 12. ELLPack allocates a rectangular matrix, setting the width to the largest row width and pads smaller rows with zeroes. Most of the rows in D_τ matrix contain 12 NNZ/row, so the padding overhead of the rows is minimal. However, half of the rows in D_v matrix contain 8 NNZ/row, so we use the SELLPack format proposed in [6]. SELLPack format allows defining different number of NNZ/row in different sections of the same matrix. We reorder D_v matrix, as shown in Fig. 2a, to have 12 NNZ/row in the first half of the matrix and 8 NNZ/row in the second half of the matrix. The reordering does not impact the performance, as the code performs it once. This effectively saves 17 % of D_v SpMV product operations.

We apply SB techniques [1,4] to reduce the main memory data traffic of the multiplied vector. In the regular update order of the SpMV product, the elements of the multiplied vector are accessed several times. As the vectors are larger than the cache memory, the vector elements are brought from main memory several times. SB changes the operation order in the matrix, such that blocks of matrix rows touching the same vector elements are updated together, while these vector elements are in the cache memory. This idea is illustrated in Fig. 2b. First the SpMV product of the dark red rows of the matrix is performed, while keeping the dark red part of the vector in the cache memory. Then the bright blue part is updated similarly, etc. As long as the block size fits in the cache memory, each element of the vector is brought once from the main memory. We show in Sect. 3 that combining SB and mRHS blocking can be inefficient in KNL due to the small cache memory size. Therefore, we reorder the loop over the Matrix components (i.e., row blocks of size N) with the loop over the rows of one component, which effectively reduces the SB block size to one row. As a result, the first row of each matrix component is evaluated first, then the next row, etc.

EMGeo solves Eq. (2) for multiple independent sources (RHS). In the RHS cache blocking approach we perform the SpMV product over several RHS's while



(a) Reordering for SELLPack format (b) Spatial blocking (c) RHS blocking

Fig. 2. Showing the reordered D_v matrix for SELLPack format in (a). Also, showing spatial (b) and RHS blocking (c) update order (Color figure online)

a block of the matrix is in the cache memory, which is relevant to [2,3]. RHS blocking amortizes the cost of loading the matrix from main memory, which is the dominant contributor of the main memory data traffic. We use row-major ordering in the RHS matrix to provide contiguous and vectorization-friendly data access pattern. The RHS blocking update order, combined with SB, is illustrated in Fig. 2c. First, each dark red block of the matrix performs the SpMV product over all the RHS, while the block is in the cache memory, then the bright blue blocks are updated, etc.

3 Analysis and Modeling SpMV Product Optimization

We analyze the FLOPS and memory data transfer requirement of the base case (i.e., the unoptimized code) and our improvements. Modeling the performance provides bounds of the expected performance improvement. We use the cache block size model to reduce the cache block size tuning parameter search space. In the following, we analyze the matrix and vector requirements separately then combine the total requirements together to estimate the AI of different setups.

Matrix data transfer and FLOPS requirement: The matrix is loaded once from main memory per SpMV product. Each element of the matrix requires loading 16 Bytes for the double complex number and 4 Bytes for the index. Moreover, the SpMV product requires 6 FLOPS to perform each complex number multiplication as well as twice the matrix bandwidth minus one FLOPS per matrix row for the reduction. Table 1.a. summarizes the total FLOPS and bytes requirement of D_v and D_τ. We notice that SELLPack format saves 20 % and 21 % in the data and the FLOPS compared to ELLPack data format, respectively.

Vectors data transfer requirement: In the “naïve” SpMV product update order, the vectors are loaded multiple times from main memory because they cannot fit

Table 1. Improvements of D_v and D_t (a,b) and their combined effects (c). N is the number of total grid points

Matrix	Format	Data (Bytes)	FLOPS
D_v	ELLPack	**1440N** =12*(16+4)*6N	**564N** =(12*6 + 11*2)*6N
	SELL	**1200N** =10*(16+4)*6N	**468N** =(10*6 + 9*2)*6N
D_τ	ELLPack	**720N** =12*(16+4)*3N	**282N** =(12*6 + 11*2)*3N

(a) D_v and D_τ Matrices data transfer and FLOPS requirement

Approach	D_v	D_τ	Total transferred elements
Naïve	**27N** =3N*5+6N*2	**15N** =6N*1.5+3N*2	42N
SB	**15N** =3N+6N*2	**12N** =6N+3N*2	27N
Improvement	1.8×	1.25×	1.6×

(b) Naïve and spatial blocking data transfer requirement

Approach	Flops	Bytes	AI	Transfer improvement
Naïve	846N	2832N	0.30	–
SELL	750N	2592N	0.29	1.09×
SB	846N	2592N	0.33	1.09×
SELL + SB	750N	2352N	0.32	1.20×

(c) Combined improvement

in the cache memory. Each nonzero $N \times N$ block of the matrix, requires loading N numbers of the multiplied vector. Hence, D_v SpMV product requires loading 15 N numbers (loading the multiplied vector 5 times) and D_τ SpMV product requires loading 9 N numbers (loading the multiplied vector 1.5 times). SB can ideally load the multiplied vector once during the SpMV product by reusing each vector element completely while in the cache memory. The results vector requires two data transfers per number between the cache and the main memory, assuming no streaming stores operations. In Table 1.b, we show the vectors data transfer model of D_v and D_τ SpMV products using the naïve and SB approaches. The total data transfer requirement of the vectors is insignificant compared to the matrices in the SpMV products. However, the vector data transfer becomes significant in the RHS cache blocking optimization.

Total data transfer and arithmetic intensity: We show the AI model, the total FLOPS, and data transfer requirement of the SpMV product in Table 1.c, when using SELLPack format and SB techniques. Although the SELLPack optimization does not improve the AI, it reduces the total FLOPS and data transfer, thus it improves the performance. We use the data transfer reduction factor as an indication to the performance improvements, as the SpMV product is memory bound. We model the RHS blocking improvements factor by considering the memory data transfer reduction as the ratio in the following equation:

$$\text{Improvement factor} = \frac{M_{RHS} \times (M_b + V_b)}{M_b + V_b \times M_{RHS}} \tag{4}$$

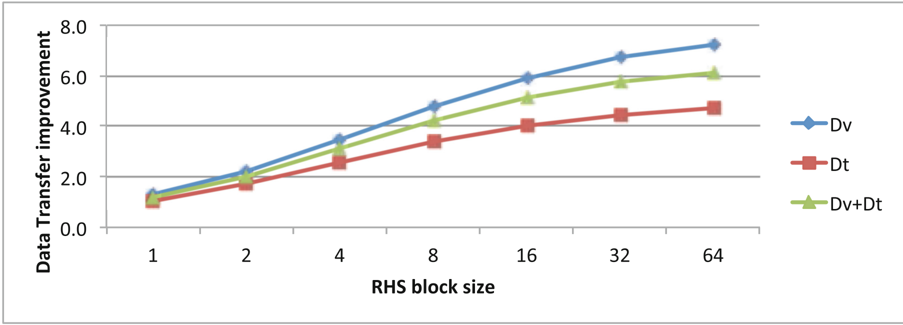


Fig. 3. Spatial and RHS blocking estimated data transfer improvement factor, as calculated in Eq. (4)

where M_{RHS} is the number of blocked RHS, M_b and V_b are the required matrix and vector main memory loads, respectively. Here we divide the required data transfer of separate SpMV products by the required data transfer of loading the matrix once with M_{RHS} vectors. In Fig. 3, we show the model data transfer improvement factor of the RHS blocking with SELLPack and SB, using the naïve implementation as the baseline. We observe significant improvement by the RHS blocking compared to other techniques, as it amortizes the matrix data transfer, which is the significant part. As the RHS block size increases we notice less improvement, for example, from 32 to 64 RHS.

3.1 Cache Block Size Model

Increasing the RHS block size reduces the number of matrix rows that fit in the cache memory. We construct a cache block size model to estimate the number of rows that fit in the cache memory from the parameters setup. The number of rows that fit in a given Cache memory is

$$C \Big/ \left(\frac{V_e}{\text{row}} \times M_{RHS} \times 16 + 2 \times \frac{NNZ}{\text{row}} \times (16+4) \right) \tag{5}$$

where C is the cache memory block size in bytes. V_e/row is the number of loaded vector elements per matrix row, for example, $V_e/\text{row} = 4$ in D_v SpMV product, as we need to store one element of the result vector and read three elements of the multiplied vector. We show examples of the expected block size of various RHS block sizes and relevant cache sizes for the D_v and D_τ SpMV products in Fig. 4. Each core in Haswell has 2.5 MiB L3 cache memory per core. We use the rule-of-thumb that half of this value is usable for blocking [5,10], i.e., 1.25 MiB. Similarly we consider that KNL has 128 kiB L2 cache per thread when using two-threads per core. We observe that the block size decreases significantly in KNL when the RHS block is >16, which results in a significant control flow overhead. Therefore, we replace the SB with loop reordering in KNL to reduce the cache block size requirements.

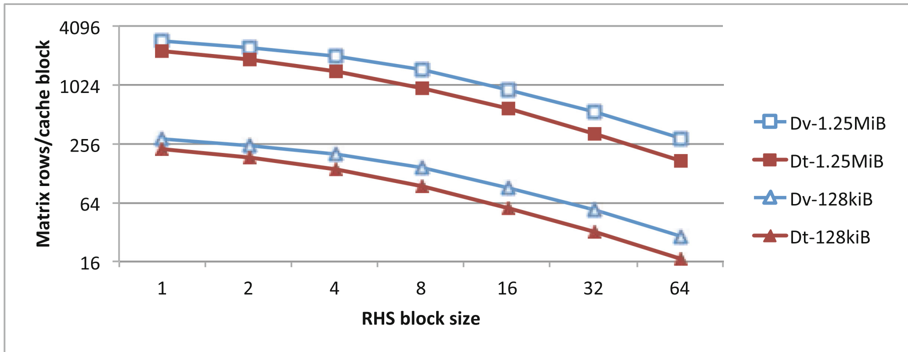


Fig. 4. RHS cache block size model at various setups. The legend refers to matrix–"cache memory size".

4 Performance Results

We study the impact of the D_v SpMV product optimization techniques in a separate benchmark to understand their characteristics. We also observe the impact of our OpenMP parallelization of the code and using our SpMV product optimization techniques in the EMGeo application.

4.1 SpMV Product Benchmark

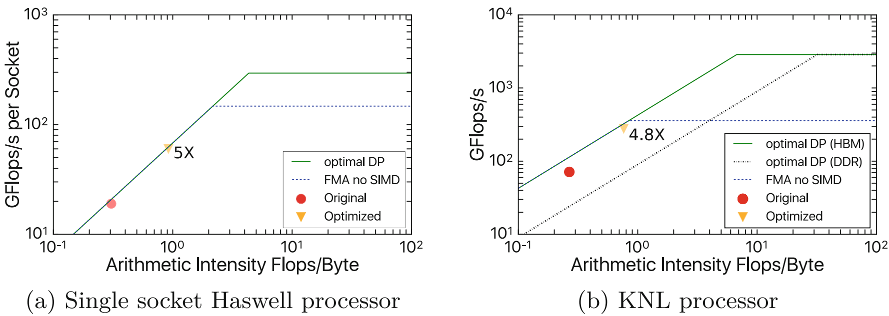
We use a benchmark code for the D_v SpMV product in the EMGeo application, as the SpMV products consume significant portion of the code runtime, as we show in Sect. 4.2. Table 2 shows the performance improvements and the transferred memory volumes improvements model prediction and measurement, using different optimization combinations. We show results for a single socket Haswell and KNL processors, using a grid of size $110 \times 110 \times 105$. The results in the last row in Table 2 use SB and loop reordering in Haswell and KNL, respectively, in addition to the SELL and mRHS optimizations. We do not use SB in KNL optimizations because it results in less performance than the naïve code. In the RHS blocking optimization, we use 32 and 64 RHS block size in Haswell and KNL, respectively. The SpMV operation is repeated 100 times, where every 10 repetitions are timed together. We report the median time of the experiments.

KNL results are reported in SNC2-flat mode using MCDRAM only, as the data fits in the MCDRAM in the production code. We observe similar performance in all KNL modes. Using the MCDRAM memory, compared to using the DDR4, increases the performance in KNL by a factor of $3.0\times$ and $4.2\times$ in the naïve and optimized codes, respectively. KNL is faster than a single socket Haswell processor by over a factor of $3\times$, which is mainly attributed to the higher memory bandwidth.

We make several observations regarding the transferred memory volume improvement model and measurements in Table 2. The measurements are closer

Table 2. D_v SpMV product benchmark measurements and model of different optimizations in shared memory. "Best" in the last row refers to SB and loop reordering in Haswell and KNL, respectively

Processor	Haswell (1 socket)			KNL			KNL/
	Perf.	Mem vol. improv.		Perf.	Mem vol. improv.		Haswell
Approach	Speedup	Measured	Model	Speedup	Measured	Model	Speedup
Naïve	-	-	-	-	-	-	3.71
SELL	1.05	1.06	1.15	1.19	1.09	1.15	4.20
SB	1.05	1.11	1.11	0.88	1.06	1.11	3.08
SELL+SB	1.26	1.28	1.30	1.13	1.23	1.30	3.33
Best+SELL+mRHS	4.97	5.36	6.75	4.79	3.50	7.20	3.58



(a) Single socket Haswell processor (b) KNL processor

Fig. 5. The roofline results of different optimization techniques over the D_v SpMV product benchmark

to the model in Haswell, especially the "SB" and "SELL+SB" results, which may be attributed to its larger cache memory per core. The optimizations in the last row result in large gap between the memory measurements and model. We observed that the gap increases as we use larger RHS block size. Consequently, the KNL result has larger gap as it uses double the RHS block size.

The roofline analysis [12–14] of the D_v benchmark results is shown in Fig. 5, where we used the techniques described in [7] and using Intel Software Development Emulator [11] to prepare these results. The roofline model shows that our RHS blocking technique significantly improves the AI. The code is still memory bandwidth-bound, so it cannot benefit from vectorization optimizations.

4.2 EMGeo Performace

We measure the time in the major components of EMGeo code. The time is mainly dominated by the IDR solve, which is in turn is dominated by the SpMV products and the MPI communication. Our experiments consist of single Cori node, single KNL processor, and 16 Cori nodes results.

Table 3. Single node (a) and multi-node performance (b) of EMGeo code. Note that SpMV product and communication time are parts of the IDR solve time

	IDR solve	SpMV		Communication		MPI topology			Threads/
	Time	Time	%	Time	%	X	Y	Z	process
	Haswell								
Original	208	147	71%	30	14%	2	4	4	1
Optimized	118.5	40	34%	14	12%	1	1	2	16
Speedup	1.76x	3.68x	-	2.14x	-	-	-	-	-
	KNL								
Original	151.7	79	52%	53	35%	4	8	4	1
Optimized	45.6	19.25	42%	14	31%	4	4	4	2
Speedup	3.33x	4.10x	-	3.79x	-	-	-	-	-

(a) Single node performance

	IDR solve	SpMV		Communication		MPI topology			Threads/
	Time	Time	%	Time	%	X	Y	Z	process
Original	342	122	36%	216	63%	4	8	16	1
Optimized	174	38	22%	64	37%	2	4	4	16
Speedup	1.97x	3.21x	-	3.38x	-	-	-	-	-

(b) performance on 16 Haswell nodes

We summarize the single node results in Table 3.a. The experiment evaluates 32 RHS with 500 Krylov solver iterations, using a $100 \times 50 \times 50$ grid size, which is comparable to the subdomain size in productions scenarios. The original code does not have shared memory parallelization, so it uses one MPI rank per core. The optimized code uses 32 RHS block size and uses single MPI rank per socket in Haswell experiment. We observe that the SpMV product takes over half of the runtime in the original IDR solver implementation. Our SpMV product optimizations result a speedup of 3.7× in Haswell and 4.1× in KNL, which is different than D_v SpMV product improvements in the benchmark. In addition to the difference in the grid size, D_τ SpMV product has less benefit from our optimization because it does not utilize the SELLPack format. Moreover, several SpMV product kernels in the application are fused with other kernels to improve the data locality. The reduction in the MPI ranks by a factor of 16× has significant impact in speeding up the code, as less ranks are involved in the reductions and halo exchange operations.

We show results for KNL in Quad flat mode, as we obtain the same performance in the other modes. The whole application data fits in the MCDRAM memory, so we run the code using the MCDRAM only. The best performance in KNL is observed at two threads per core. We tuned the MPI ranks vs. the OpenMP threads manually in the optimized code. We observe 3.3× speedup in the code, where the SpMV and communication operations run about 4.1× and 3.8× faster, respectively. Using the MCDRAM memory, compared to using the DDR4 only, increases the performance in KNL by a factor of 4.2× in the optimized code.

We summarize the results of a 16 Cori nodes experiment in Table 3.b. The experiment evaluates 32 RHS with 2500 Krylov solver iterations limit, using a grid of size 100^3. The SpMV product optimizations result in less improvement in this code, mostly, because smaller subdomains are evaluated in shared memory. The MPI communication consumes about half the runtime due to the increased surface-to-volume ratio of the subdomains. Again, by reducing the number of MPI processes by a factor of 16×, our optimized version achieves about 3.4× speedup in the communication time. We discuss further ideas to handle this issue in the future work section.

5 Conclusion and Outlook

In this paper, we present optimization techniques in Intel Haswell and KNL processors for EMGeo software. In particular, we optimize the SpMV product in the IDR Krylov solver part, where most of the application time is spent. We obtain performance improvements by reducing the data traffic to the main memory in the SpMV products and reducing the MPI communication time by using hybrid MPI+OpenMP parallelization. We use SB, SELLPack sparse matrix format, and most importantly a RHS cache blocking technique. We deploy performance modeling to identify relevant optimizations and to understand the optimization quality and issues.

Our optimizations improved the performance of the D_v SpMV product by a factor of 5.0× in Haswell and 4.8× in KNL. We improve the performance of the forward step of EMGeo application by incorporating our SpMV optimizations and using OpenMP parallelization. As a result, The application runs 1.8× and 3.3× faster in Haswell and KNL, respectively.

In general, KNL achieves better performance than Haswell due to the higher available memory bandwidth. SB did not improve the KNL performance, but we gained the desired improvement by reordering the loops to access the matrix rows in an array-of-structures pattern.

RHS blocking provides significant performance improvements and prepares the code to use block IDR algorithm and overlap computations with communication in the solver. We plan to implement and validate the Block IDR method, which is expected to significantly reduce the required iteration count to convergence. We also plan to overlap the computations and communication of independent RHS to obtain better strong scaling performance.

Acknowledgments. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Datta, K.: Auto-tuning stencil codes for cache-based multicore platforms. Ph.D. thesis, EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-177.html
2. Gropp, W., Kaushik, D., Keyes, D., Smith, B.: Toward realistic performance bounds for implicit CFD codes. In: Proceedings of parallel CFD, vol. 99, pp. 233–240. Citeseer (1999)
3. Kreutzer, M., Thies, J., Röhrig-Zöllner, M., Pieper, A., Shahzad, F., Galgon, M., Basermann, A., Fehske, H., Hager, G., Wellein, G.: GHOST: building blocks for high performance sparse linear algebra on heterogeneous systems abs/1507.08101 (2015). http://arxiv.org/abs/1507.08101
4. Malas, T., Hager, G., Ltaief, H., Stengel, H., Wellein, G., Keyes, D.: Multicore-optimized wavefront diamond blocking for optimizing stencil updates. SIAM J. Sci. Comput. **37**(4), C439–C464 (2015). doi:10.1137/140991133
5. Malas, T.M.: Tiling and asynchronous communication optimizations for stencil computations. Ph.D. thesis, King Abdullah University of Science and Technology, December 2015
6. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 111–125. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11515-8_10
7. NERSC: Measuring arithmetic intensity. https://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity
8. Petrov, P.V., Newman, G.A.: Three-dimensional inverse modelling of damped elastic wave propagation in the fourier domain. Geophys. J. Int. **198**(3), 1599–1617 (2014)
9. Petrov, P.V., Newman, G.A.: 3d finite-difference modeling of elastic wave propagation in the laplace-fourier domain. Geophysics **77**(4), T137–T155 (2012). doi:10.1190/geo2011-0238.1
10. Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 207–216. ACM (2015)
11. Tal, A.: Intel software development emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator
12. Williams, S.: Auto-tuning performance on multicore computers. Ph.D. thesis, EECS Department, University of California, Berkeley, December 2008
13. Williams, S., Watterman, A., Patterson, D.: Roofline: an insightful visual performance model for floating-point programs and multicore architectures. Commun. ACM. **52**(4), 65–76 (2009)
14. Williams, S., Stralen, B.V., Ligocki, T., Oliker, L., Cordery, M., Lo, L.: Roofline performance model. http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/