August 2017

Porting of the DBCSR library for Sparse Matrix-Matrix Multiplications to Intel Xeon Phi systems

Iain BETHUNE [a,1], Andreas GLÖSS [b,2], Jürg HUTTER [b,3], Alfio LAZZARO [b,4], Hans PABST [c,5] and Fiona REID [d,6]

[a] *Hartree Centre, Science and Technology Facilities Council, United Kingdom*
[b] *University of Zurich, Department of Chemistry, Switzerland*
[c] *Intel Semiconductor AG, Switzerland*
[d] *EPCC, The University of Edinburgh, United Kingdom*

Abstract. Multiplication of two sparse matrices is a key operation in the simulation of the electronic structure of systems containing thousands of atoms and electrons. The highly optimized sparse linear algebra library DBCSR (Distributed Block Compressed Sparse Row) has been specifically designed to efficiently perform such sparse matrix-matrix multiplications. This library is the basic building block for linear scaling electronic structure theory and low scaling correlated methods in CP2K. It is parallelized using MPI and OpenMP, and can exploit GPU accelerators by means of CUDA. We describe a performance comparison of DBCSR on systems with Intel Xeon Phi Knights Landing (KNL) processors, with respect to systems with Intel Xeon CPUs (including systems with GPUs).

We find that the DBCSR on Cray XC40 KNL-based systems is 11%-14% slower than on a hybrid Cray XC50 with Nvidia P100 cards, at the same number of nodes. When compared to a Cray XC40 system equipped with dual-socket Intel Xeon CPUs, the KNL is up to 24% faster.

Keywords. sparse matrix-matrix multiplications, vectorization, multi-threading, MPI parallelization, accelerators, Intel Xeon Phi, Knights Landing

1. Introduction

Multiplication of two sparse matrices (SpGEMM) is a key operation in the simulation of the electronic structure of systems containing thousands of atoms and electrons [1]. Examples of such systems include electronic devices, complex interfaces, macromolecules and large disordered systems, with applications in the fields of renewable energy and electronics. The theory that enables such studies is linear scaling Density Functional

[1] E-mail: iain.bethune@stfc.ac.uk
[2] E-mail: andreas.gloess@chem.uzh.ch
[3] E-mail: hutter@chem.uzh.ch
[4] E-mail: alfio.lazzaro@chem.uzh.ch
[5] E-mail: hans.pabst@intel.com
[6] E-mail: f.reid@epcc.ed.ac.uk

Theory (DFT) [2]. In the atomistic simulation package CP2K [3], the linear scaling DFT implementation exploits the fact that for large enough systems, operators in a localized atomic basis become sparse [1]. The matrices have several thousands of non-zero elements per row and *a priori* unknown sparsity patterns. In these simulations, SpGEMM typically accounts for more than 80% of the total runtime. The computational cost depends strongly on the evolution of the sparsity during the iterations, which in turn depends on the chemical properties of the system studied, the precise algorithm employed, the system size, and the required accuracy [1]. The highly optimized sparse linear algebra library DBCSR (Distributed Block Compressed Sparse Row) has been specifically designed to efficiently perform such block-sparse matrix-matrix multiplications [4,5,6]. It is parallelized using MPI and OpenMP, and can exploit GPU accelerators by means of CUDA.

Here we describe our evaluation of DBCSR on systems equipped with Intel Xeon Phi 'Knights Landing' (KNL) processors. These systems are emerging as a viable but segment-specific alternative to traditional x86-64 CPU systems and systems with GPUs to reach higher computational density [7]. Although running on the Intel Xeon Phi is straightforward, it poses several challenges for the application in order to obtain good performance, such as vectorization, memory management and multi-threading [8,9]. We compare performance between runs on KNL systems with respect to systems with Intel Xeon CPUs (including systems with GPUs), on up to 144 nodes. In the interest of portability, the same DBCSR code was used for CPU and KNL executions, i. e. we do not use any particular code optimization specific for KNL systems besides the optimization provided by the compiler.

1.1. Related Work

The classical serial SpGEMM algorithm was first described by Gustavson [10]. The parallel implementation in a distributed memory system presents several challenges, such as load-balance and communication costs relative to arithmetic operations, and several algorithms have been proposed [11,12,13]. DBCSR considers the general case where *a priori* knowledge of the input and output matrix sparsity is not employed, and is aimed at delivering good performance in the 'nearly dense' regime i. e. many non-zeros per row. It uses a random permutation of the rows and columns of the matrix to achieve a good average load-balance. Consequently, the data and the corresponding operations are statically distributed across processes in the same way as for dense matrices, and existing algorithms for dense matrix-matrix multiplications (e.g. [14]) can be adopted and refined for the sparse case. Recently, we have implemented a 2.5D algorithm that is able to improve the performance for large number of processors [6].

Several papers report on SpGEMM implementations for single-node GPU-enabled systems [15,16,17]. The work of Liu and Vinter [18] addresses heterogeneous CPU-GPU processors. A recent paper by Deveci *et al.* [19] introduces an implementation that specifically targets GPU and KNL. None of these implementations target block-sparse matrices. Concerning parallel implementation in a hybrid CPU-GPU multinode system, Rubensson and Rudberg [20] reported a parallel implementation where the mapping of data and work to physical resources is performed dynamically during the calculation. Like DBCSR, this implementation is able to work effectively with block-sparse matrices and runs on hybrid multi-cores CPU and GPU systems, however it does not employ optimized libraries for the small block multiplications.

2. DBCSR Library

DBCSR is written in Fortran and is freely available under the GPL license from `https://dbcsr.cp2k.org`. DBCSR matrices are stored in a blocked compressed sparse row (CSR) format distributed over a two-dimensional grid of P MPI processes. Inter-process communication is based on the communication-reducing 2.5D algorithm [6]. In the tests reported in this paper, the data of the matrix multiplication $C = C + A \cdot B$ is decomposed such that it requires only the communication of the A and B matrix data. These communications are implemented with asynchronous point-to-point, MPI calls, using the MPI Funneled mode. The local multiplication will start as soon as all the data has arrived at the destination process (by using a `mpi_waitall` call). The amount of communicated data by each process scales as $\mathcal{O}(1/\sqrt{P})$.

The local computation consists of pairwise multiplications of small dense matrix blocks, with dimensions $(m \times k)$ for A blocks and $(k \times n)$ for B blocks. It employs a cache oblivious matrix traversal to fix the order in which matrix blocks need to be computed, in order to improve memory locality. First, the algorithm loops over A matrix row-blocks and then, for each row-block, over B matrix column-blocks. A filtering procedure is applied on the multiplication (on-the-fly filtering) of the blocks so that only blocks for which the product of their norms exceeds a given threshold will be actually multiplied. This filtering increases sparsity but also avoids performing calculations that fall below the filtering threshold, which results in a significant speed-up of the entire operation [1]. Then, the corresponding multiplications are organized in batches. Multiple batches can be computed in parallel on the CPU by means of OpenMP threads or alternatively executed on a GPU. A static assignment of batches with a given A matrix row-block to threads is employed in order to avoid race conditions. Processing the batches has to be highly efficient. For this reason specific libraries were developed that outperform vendor BLAS libraries, namely `LIBCUSMM` for GPU and `LIBXSMM` for CPU/KNL systems [5,21].

For GPU execution, data is organized in such a way that the transfers between the host and the GPU are minimized. A double-buffering technique, based on CUDA streams and events, is used to maximize the occupancy of the GPU and to hide the data transfer latency. When the GPU is fully loaded, computation may be simultaneously done on the CPU. `LIBCUSMM` employs an auto-tuning framework to find optimal parameters and implementations for each given set of block dimensions. In this way the library is able to achieve a speedup in the range of 2–4x with respect to batched DGEMM in cuBLAS [5]. For Nvidias P100 we re-optimized the kernel parameter set. The performance results for the execution of blocked multiplication batches are shown in Figure 1.

`LIBXSMM` is a library targeting Intel Architecture for small, dense or sparse matrix multiplications, and small convolutions. The library generates executable code Just-In-Time (JIT) by assembling the instructions in-memory. All flavors of AVX extensions are supported via JIT-code, and particular emphasis is given to AVX-512. Besides redispatching generated code for every multiplication, the library can generate or dispatch the code ahead of time. This is used by DBCSR as the block sizes of the multiplication batches are known upfront. To quantify the advantage of `LIBXSMM` over vendor BLAS, we measured the performance (DP-GFLOP/s) of multiplications $C = C + A_i \cdot B_i$ with $i = 1 \ldots N$ such that N amounts to a working set of 2 GB (`LIBXSMM`'s SMM sample), and calculated the geometric mean of the performance for a series of kernels (the same as in Figure 1). Streaming A and B matrices from memory (DDR4 or MCDRAM), and accumulating

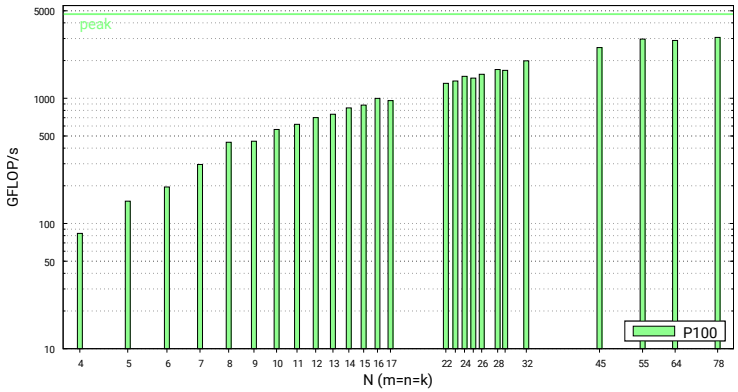

Figure 1. `LIBCUSMM` performance on Nvidia P100 card for selected optimized CUDA-kernels comprising $(m = n = k) = \{4, ..., 78\}$ block sizes. The FLOP-rates, as obtained from individual kernel launches in a mini-app that mimics `DBCSR` multiplication of batches, are shown as green bars. The horizontal line refers to the double-precision peak performance of the Nvidia P100 used in our tests.

into C (cached) conforms with `DBCSR`'s batched block multiplication. We measured a speedup of 2.9x for `LIBXSMM` over `MKL`. We have not implemented `MKL`'s batch-GEMM in `DBCSR` and did not try MKL_DIRECT, but expect `LIBXSMM` to maintain the advantage [21]. In absolute numbers when compared to Figure 1, KNL yields higher absolute performance for smaller kernel sizes. The latter is true even when the mini-app (used to tune `LIBCUSMM`) is assumed to stream A and B matrices from memory (rather than running hot in cache). In turn, relying on in-cache block multiplications with `LIBXSMM` peaks at 1.9 TF/s (32x32 kernel).

3. Performance Results

We present the results of running `DBCSR` within `CP2K` benchmark applications, representative of large-scale and long-running science runs of `CP2K` for linear scaling calculations. Importantly, these result in matrices with different block sizes and occupation, which affects performance and scalability.

Timings are obtained from a `CP2K` internal timing framework. We did not perform any lower-level measurements of performance, such as based on hardware event counters. We considered only the execution time of the `DBCSR` multiplication part, and not any other `CP2K` specific parts. Results are taken as the average of 4 independent application runs, each consisting of tens of multiplications – fluctuations are found to be less than 5%. Elements of the generated matrices are double precision floating point numbers.

3.1. Single-node Performance Results

We present the results of running the `CP2K H2O-64` benchmark (a small system of 64 water molecules) on the three systems:

- *ARCHER*: 4920 Cray XC30 compute nodes with Intel Xeon E5-2697 v2 (12 cores, dual-socket @ 2.7 GHz), 64 GB of RAM on 4544 nodes and 128 GB of RAM on the remaining 376 nodes.

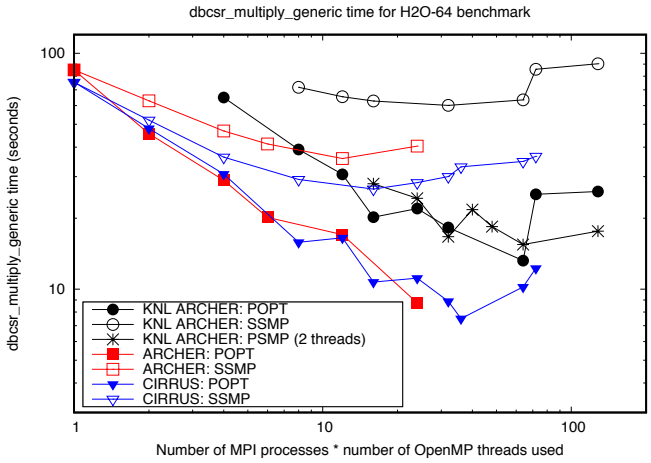

Figure 2. `H2O-64` performance for ARCHER, Cirrus and ARCHER-KNL on a single node execution. POPT denotes a pure MPI run, SSMP denotes pure OpenMP run and PSMP denotes a run using both MPI and OpenMP.

- *Cirrus*: 280 node SGI ICE XA with Intel Xeon E5-2695 v4 (18 cores, dual-socket @ 2.1 GHz), 256 GB of RAM;
- *ARCHER-KNL*: 12 Cray KNL compute nodes (64 cores Intel Xeon Phi CPU 7210 @ 1.3 GHz, 16 GB MCDRAM), 96 GB of RAM.

The goal of these tests is to get a first insight into the KNL performance. The results are shown in Figure 2. The block sizes for this test are combinations of $(m,n,k) = \{9,22,32\}$ with fully occupied matrices. For each system we ran the benchmark in pure MPI (POPT), pure OpenMP (SSMP) and for KNL only using both MPI + OpenMP (PSMP). On the KNL we used a node with full CACHE and QUADRANT clustering and tested all possible combinations of MPI processes and OpenMP threads. We include the best PSMP result which used 2 OpenMP threads per process.

For each system the POPT version running on a fully populated node gives the best performance with the SSMP version generally being around two times slower. On Cirrus and the ARCHER-KNL where the number of cores is greater than 64, we used hyperthreading (a maximum of 2 threads per core was tested) and it is obvious that this does not enhance the performance. The fastest result of 7.5s is obtained using 36 MPI processes on Cirrus, with the best result on ARCHER being 8.7s on 24 MPI processes and 13.2s on the ARCHER-KNL on 64 MPI processes.

3.2. Multiple-node Performance Results

Tests are based on three `CP2K` benchmarks:

- `S-E`: semi-empirical benchmark with 186,624 water molecules - highly sparse matrices (average occupancy 0.05%).
- `H2O-DFT-LS`: single-point energy calculation with linear scaling DFT consisting of 20,736 atoms - medium sparsity matrices (average occupancy 10%).
- `AMORPH`: single-point energy calculation with linear scaling DFT consisting of 13,846 atoms - low sparsity matrices (average occupancy 70%).

Table 1. Block sizes, dimension of matrices (rows and columns), typical occupancy of the matrices, number of multiplications performed, and DBCSR FLOPs for the three benchmarks.

	S-E	H2O-DFT-LS	AMORPH
Block sizes (m,n,k)	6	23	5, 13
# Rows/columns	1,119,744	158,976	141,212
Occupancy range (%)	$(4-6)\times 10^{-2}$	$7-15$	$34-77$
# Multiplications	618	193	187
DBCSR FLOPs ($\times 10^{12}$)	74	4,038	3,656

The block sizes, total number of rows/columns (all matrices are square), typical occupancy during the simulations, number of multiplications, and FLOPs executed by DBCSR part only are reported in Table 1.

We compare the performance obtained on several systems based on Intel Xeon CPUs, Nvidia P100 GPUs, and KNL, hosted at the Swiss National Supercomputing Centre (CSCS):

- *Grand Tavé*: 164 Cray XC40 compute nodes (64 cores Intel Xeon Phi CPU 7230 @ 1.3 GHz, 16 GB MCDRAM), 96 GB of RAM;
- *Piz Daint*: this system has two partitions: 5,320 Cray XC50 hybrid compute nodes (GPU partition) with Intel Xeon E5-2690 v3 (12 cores single socket @ 2.6 GHz) and Nvidia Tesla P100 (16 GB High Bandwidth Memory), 64 GB of RAM; 1,431 Cray XC40 CPU compute nodes (MC partition) with Intel Xeon E5-2695 v4 (18 cores, dual-socket @ 2.1 GHz), 64 GB of RAM.

All CPU cores have Intel Turbo and Intel Hyper Threading Technology enabled. The latter is not used in our benchmark runs, i. e. each thread runs on a single physical core. Indeed, we found that running more threads per core does not give any speed-up. Both systems feature Cray's Aries network. We also found that the module `craype-hugepages2M`, which enables page sizes of 2 MB, gives an average speed-up of 18% for the KNL runs.

We obtained the best performance by using a single MPI rank and 12 threads per node on the GPU partition, 4 ranks and 9 threads on the MC partition, and 4 ranks and 16 threads on KNL. These configurations give the best performance of all ranks/threads in a node with a speed-up of up to 40%. This result is an implicit consequence of the multiplication algorithm, which gives better performance for the communications of data (computation is not affected) when a minimal square number of ranks is employed [6]. The total number of MPI ranks for the KNL and MC benchmarks is 4 times the ranks of the GPU ones, which implies twice as much data to communicate per rank as a consequence of the multiplication algorithm (see Section 2).

All tests on KNL are executed in full CACHE mode for the MCDRAM management and QUADRANT clustering mode. It is worth underlining that the entire CP2K application requires a maximum of 10 GB per node, therefore it fits entirely in MCDRAM. Specific tests requiring the application to run in MCDRAM (by using FLAT mode and forcing all allocations in MCDRAM) did not show any significant speed-up in performance.

The DBCSR multiplication execution times for the three systems are reported in Table 2. We also report the average fractions of time spent in the `mpi_waitall` call, used for the communication of the A and B matrices data, and for the computation of the block

multiplication batches. The time spent in the `mpi_waitall` call is not the full communication time for the exchange of the data, but only the part that did not overlap with computation of the block multiplication batches. The remaining part of the execution time is the organization, scheduling and finalization of the matrix block multiplications. It is partially parallelized with OpenMP and is memory-bandwidth-bound. It also includes the transfer of the *C* matrix data between the central memory and the GPU memory [5]. The performance ratios are shown in Figure 3. On average, we find that KNL executions are:

- 11%-14% slower than GPU executions for the thee benchmarks;
- 18% slower than MC executions for `S-E`;
- 24% and 4% faster than MC executions for `H2O-DFT-LS` and `AMORPH`, respectively.

Although the interpretation of the data is difficult, as the algorithm is largely asynchronous, both with computation on the GPU/CPU and with communication across the network, we can explain these results with the following observations:

1. From the time spent in the multiplication of batches, we see that GPU is particularly efficient for the `H2O-DFT-LS` and `AMORPH` benchmarks, which involve somewhat large block sizes (see Figure 1): on average, KNL executions are 35% and 27% slower than GPU executions for `H2O-DFT-LS` and `AMORPH` benchmarks, respectively, while they are 7% faster for the `S-E` benchmark. On the other side, KNL executions are faster than MC executions for `H2O-DFT-LS` (67%), same performance for `AMORPH`, and slower for `S-E` (13%).
2. The time spent in the `mpi_waitall` call is directly related to the previous point, since it is the remaining time for communications that does not overlap with the computation. The `H2O-DFT-LS` is the most communication-bound, while the `AMORPH` is computation-bound. As expected, the communication fractions increase with the number of MPI ranks (see Section 2). Note that the GPU runs use 4 times fewer ranks than the others, therefore half as much data is communicated.
3. As a combination of the previous two points: on average, the KNL benchmarks are between 14%-22% slower than GPU benchmarks and 10%-17% faster than MC ones. These values are partially compensated by the remaining part of execution, which scales better on the KNL and MC systems.

In summary, KNL gives poorer performance than GPU for the computational part when kernels are large and it requires more communication time, but it has a faster execution of the remaining part, while the opposite is true when compared to MC. Which aspect dominates depends on the block sizes and scale of execution.

3.3. Multi-thread Scalability

We illustrate the threading scalability by considering the execution time on 144 nodes with different numbers of threads. The speedup values are shown in Figure 4. We can interpret these results by applying the aforementioned considerations for the execution time. In particular, the scalability is limited by the communication time (since only the master thread handles communications) and by the initialization and finalization of the matrix multiplications, which are partially parallelized and mostly memory-bandwidth

Table 2. DBCSR results (time-to-solution) for multiple nodes tests for the three benchmarks. We also report the average fractions of time spent in the `mpi_waitall` call and for the computation of the block multiplication batches. Average timings are obtained from the values of all involed MPI ranks.

	# nodes	S-E			H2O-DFT-LS			AMORPH		
		GPU	MC	KNL	GPU	MC	KNL	GPU	MC	KNL
Time-to-solution (seconds)	25	672	551	703	631	843	706	1050	1222	1208
	36	518	449	539	512	723	544	774	907	892
	64	361	362	414	374	533	414	493	577	551
	100	264	255	325	264	358	304	342	419	405
	144	218	215	274	231	308	267	268	321	303
Average time `mpi_waitall` (%)	25	18	38	25	28	22	31	2	3	2
	36	21	42	26	32	27	33	3	5	3
	64	29	50	32	45	38	45	6	11	5
	100	31	51	37	52	44	54	14	19	10
	144	35	56	40	55	52	57	21	27	15
Average time for multiplication batches (%)	25	25	26	21	33	57	42	67	80	77
	36	24	23	21	29	49	38	65	77	74
	64	22	17	18	21	37	30	60	71	73
	100	21	17	16	16	32	22	56	62	67
	144	19	15	14	13	26	18	51	55	62

limited. Because of that, the AMORPH benchmark shows the best thread-scalability since it is the most computation intensive. Closer analysis of the thread timing distribution during the block multiplication batches shows some load imbalance (of up to 30%). This is due the *a priori* static decomposition of the block multiplications among threads, where the load unbalance arises from the *a posteriori* on-the-fly filtering procedure (see Section 2). In the future we plan to change the algorithm to be dynamic by using OpenMP tasks. Finally, we observe that KNL scalability is always the best, due to the use of MCDRAM. Indeed, we observed a significant slowdown (between 10%-56%, depending on the benchmark) when performing tests where the application did not use the MCDRAM, i.e. FLAT mode forcing the allocations on DRAM.

4. Conclusions

We found that the DBCSR executions on Cray XC40 KNL-based systems are 11%-14% slower than on a hybrid Cray XC50 GPU based system with Nvidia P100 cards, at the same number of nodes. When compared to a Cray XC40 system equipped with dual-socket Intel Xeon CPUs, the KNL executions are of up to 24% faster. The best performance was obtained by configuring the KNL in full CACHE mode and QUADRANT clustering mode, without using hyperthreading.

Acknowledgments

This work was supported by grants from the Swiss National Supercomputing Centre (CSCS) under projects S238, K02 and UZHP and received funding from the Swiss University Conference through the Platform for Advanced Scientific Comput-

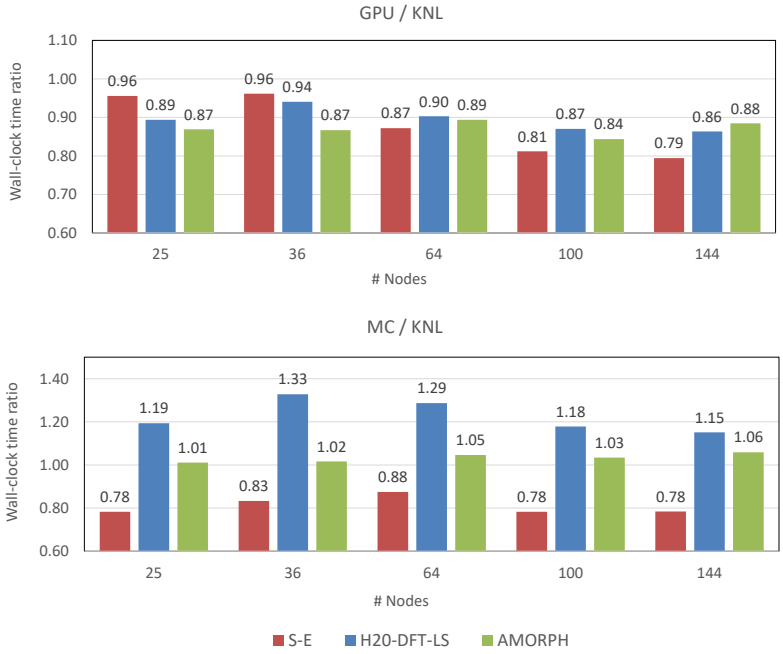

Figure 3. Ratio of `DBCSR` execution times between GPU (upper plot) and MC (lower plot) results with respect to KNL results: values greater (lower) than 1 mean that KNL executions are faster (slower). For each number of nodes, the bars refer to (from left to right): `S-E`, `H2O-DFT-LS`, `AMORPH`.

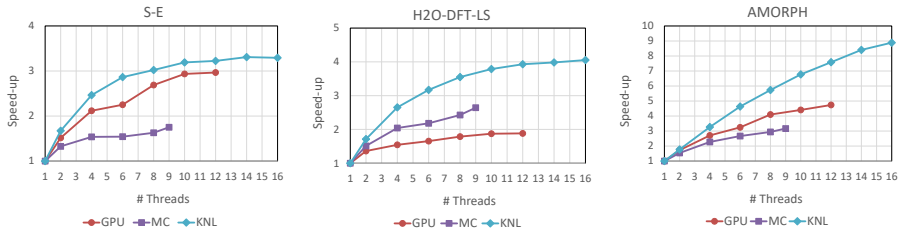

Figure 4. Speed-up when varying the number of threads with respect to the single thread execution of the `DBCSR` execution at 144 nodes for the `S-E`, `H2O-DFT-LS`, `AMORPH` benchmarks on the GPU, MC and KNL systems. The number of MPI ranks is fixed for the corresponding system.

ing (PASC). We acknowledge support from the Engineering and Physical Sciences Research Council (EPSRC) grant EP/K038583/1 and the Intel Parallel Computing Centre programme. This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk) and EPCC's Cirrus HPC Service (https://www.epcc.ed.ac.uk/cirrus).

References

[1] Joost VandeVondele, Urban Borstnik, and Jürg Hutter. Linear scaling self-consistent field calculations for millions of atoms in the condensed phase. *The Journal of Chemical Theory and Computation*, 8(10):3565–3573, 2012.

[2] David Bowler and Tsuyoshi Miyazaki. O(N) methods in electronic structure calculations. *Rep. Prog. Phys.*, 75(036503), 2012.

[3] Jürg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. CP2K: Atomistic Simulations of Condensed Matter Systems. *WIREs: Computational Molecular Science*, 4(1):15–25, 2014.

[4] Urban Borstnik, Joost VandeVondele, Valery Weber, and Jürg Hutter. Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library. *Parallel Computing*, 40(5-6):47–58, 2014.

[5] Ole Schütt, Peter Messmer, Jürg Hutter, and Joost VandeVondele. GPU Accelerated Sparse Matrix Matrix Multiplication for Linear Scaling Density Functional Theory. In *Electronic Structure Calculations on Graphics Processing Units*. John Wiley and Sons, 2015.

[6] Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '17, pages 3:1–3:9, New York, NY, USA, 2017. ACM.

[7] Top500. TOP 500 Supercomputer Sites. `http://www.top500.org`.

[8] Fiona Reid and Iain Bethune. Evaluating CP2K on Exascale Hardware: Intel Xeon Phi. Technical report, `http://www.prace-ri.eu/IMG/pdf/wp152.pdf`, 2013.

[9] Fiona Reid and Iain Bethune. Optimising CP2K for the Intel Xeon Phi. Technical report, `http://www.prace-ri.eu/IMG/pdf/wp140.pdf`, 2013.

[10] Fred G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978.

[11] Aydin Buluc and John R. Gilbert. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. *2008 37th International Conference on Parallel Processing (ICPP)*, pages 503–510, 2008.

[12] Valery Weber, Teodoro Laino, Alexander Pozdneev, Irina Fedulova, and Alessandro Curioni. Semiempirical Molecular Dynamics (SEMD) I: Midpoint-Based Parallel Sparse Matrix-Matrix Multiplication Algorithm for Matrices with Decay. *The Journal of Chemical Theory and Computation*, 11(7):3145–3152, 2015.

[13] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *ACM Trans. Parallel Comput.*, 3(3):18:1–18:34, December 2016.

[14] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication Optimal Parallel Multiplication of Sparse Random Matrices. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 222–231, New York, NY, USA, 2013. ACM.

[15] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing Sparse Matrix–Matrix Multiplication for the GPU. *ACM Trans. Math. Softw.*, 41(4):25:1–25:20, October 2015.

[16] Felix Gremse, Andreas Hfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM J. Scientific Computing*, 37(1), 2015.

[17] Lukas Polok, Viorela Ila, and Pavel Smrz. Fast Sparse Matrix Multiplication on GPU. In *Proceedings of the Symposium on High Performance Computing*, HPC '15, pages 33–40, San Diego, CA, USA, 2015. Society for Computer Simulation International.

[18] Weifeng Liu and Brian Vinter. A framework for general sparse matrixmatrix multiplication on GPUs and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47 – 61, 2015. IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[19] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 693–702. IEEE, 2017.

[20] Emanuel Rubensson and Elias Rudberg. Locality-aware parallel block-sparse matrix-matrix multiplication using the Chunks and Tasks programming model. *Parallel Computing*, 57:87–106, 2016.

[21] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.