Energy Efficiency and Performance Frontiers for Sparse Computations on GPU Supercomputers

Hartwig Anzt Stanimire Tomov Jack Dongarra
Innovative Computing Lab, University of Tennessee, Knoxville, USA
hanzt@icl.utk.edu tomov@icl.utk.edu dongarra@eecs.utk.edu

Abstract

In this paper we unveil some energy efficiency and performance frontiers for sparse computations on GPU-based supercomputers. To do this, we consider state-of-the-art implementations of the sparse matrix-vector (SpMV) product in libraries like cuSPARSE, MKL, and MAGMA, and their use in the LOBPCG eigen-solver. LOBPCG is chosen as a benchmark for this study as it combines an interesting mix of sparse and dense linear algebra operations with potential for hardware-aware optimizations. Most notably, LOBPCG includes a blocking technique that is a common performance optimization for many applications. In particular, multiple memory-bound SpMV operations are blocked into a SpM-matrix product (SpMM), that achieves significantly higher performance than a sequence of SpMVs. We provide details about the GPU kernels we use for the SpMV, SpMM, and the LOBPCG implementation design, and study performance and energy consumption compared to CPU solutions. While a typical sparse computation like the SpMV reaches only a fraction of the peak of current GPUs, we show that the SpMM achieves up to a $6\times$ performance improvement over the GPU's SpMV, and the GPU-accelerated LOBPCG based on this kernel is 3 to $5\times$ faster than multicore CPUs with the same power draw, e.g., a K40 GPU *vs.* two Sandy Bridge CPUs (16 cores). In practice though, we show that currently available CPU implementations are much slower due to missed optimization opportunities. These performance results translate to similar improvements in energy consumption, and are indicative of today's frontiers in energy efficiency and performance for sparse computations on supercomputers.

Categories and Subject Descriptors G4 [*MATHEMATICAL SOFTWARE*]: Algorithm design and analysis

Keywords sparse eigensolver, LOBPCG, GPU supercomputer, energy efficiency, blocked sparse matrix vector product

1. Introduction

Building an Exascale machine using the technology employed in today's fastest supercomputers would result in a power dissipation of about half a gigawatt [1]. Providing suitable infrastructure poses a significant challenge, and with a rough cost of one million USD per megawatt year, the running cost for the facility would quickly exceed the acquisition cost. This is the motivation for replacing homogeneous CPU clusters with architectures generating most of their performance with low-power processors, or accelerators.

Indeed, the success of using GPU accelerators to reduce energy consumption is evident by the fact that the 15 greenest systems are accelerated by GPUs, according to the June 2014 Green500 list [2], which ranks the supercomputers according to their performance/watt ratio on the HPL benchmark [3]. A drawback of this ranking is that HPL provides GFLOP/s numbers that usually are by orders of magnitude above the performance achieved in real-world applications, and thus, it is insufficient to understand how energy-efficient the accelerated systems are for scientific computing [4].

Recent work has addressed this question by comparing the energy efficiency of a large set of current hardware platforms when running a Conjugate Gradient solver [5], and the results revealed that when optimizing with respect to energy efficiency, low-power processors and the newer CPUs are able to keep up with the GPUs. Although [5] contains a comprehensive study on different optimization scenarios, considering an isolated CG fails to provide insight into the energy efficiency of more complex applications.

In this paper, we focus on GPU kernel optimizations for the memory-bound SpMV, which serves as a building block of many sparse solvers. Standard Krylov methods, for example, are typically bounded by the SpMV performance, which is only a few percent of the peak of current GPUs, as shown in Figure 1, right. One approach to improve Krylov methods is to break up the data dependency between the `SpMV` and the dot products like in the recent work on the s-step and communication-avoiding Krylov methods [7, 8]. The improvement in these methods comes from grouping SpMVs together into a (so called) Matrix Powers Kernel, which allows reuse of the sparse matrix and vector reads during the computation of a Krylov subspace. Another approach that we consider in this paper is to generate a block-Krylov subspace. Higher performance for the numerical algorithm is achieved by several vectors being multiplied simultaneously in an SpMM kernel. Communication is reduced in this case by accessing the matrix only once, and reusing it for the multiplication with all vectors, which results in significant performance benefits. SpMMs are needed in numerical algorithms like the Discontinuous Galerkin, high-order FEM, cases when vector quantities are simulated, or in specially designed block solvers, like the Locally Optimal Block PCG (LOBPCG) used for finding a set of smallest/largest eigenstates of an SPD matrix [9].

The LOBPCG method provides an interesting mix between sparse and dense linear algebra operations. It also serves as a backbone for many simulations, e.g., in quantum mechanics, where eigenstates and molecular orbitals are defined by eigenvectors, or principle component analysis. Non-linear CG methods have been successfully used in predicting the electronic properties of large

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA.

http://dx.doi.org/10.1145/10.1145/2712386.2712387

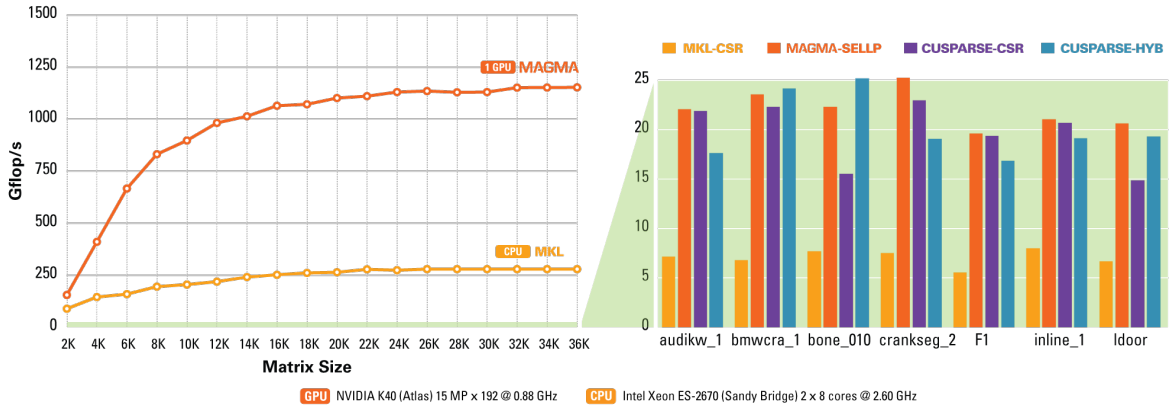


Figure 1: Nvidia K40 GPU computing efficiency on compute intensive (Left: dense LU in double precision) vs. memory-bound computation (Right: SpMV in double precision for various data formats) on state-of-the-art hardware and software libraries. CPU runs use MKL (denoted MKL-CSR) with the `numactl interleave=all` policy (see also MKL's benchmarks [6]).

nanostructures [10, 11], where the LOBPCG method in particular has shown higher performance and improved convergence (reduced number of SpMV products compared to non-blocked CG methods). Applying this algorithm efficiently to multi-billion size problems served as the backbone of two Gordon-Bell Prize finalists that ran many-body simulations on the Japanese Earth Simulator [12, 13].

In this paper, we compare the state-of-the-art multi-threaded CPU implementations of LOBPCG in BLOPEX [14], for which the PETSc and Hypre libraries provide an interface [15], with our GPU implementation, that is now available through the MAGMA library [16]. The target platform is the Piz Daint Supercomputer located at the Swiss National Computing Centre[1] (CSCS).

2. Related Work

Energy Efficiency: An overview of energy-efficient scientific computing on extreme-scale systems is provided in [17], where the authors address both hardware and algorithmic efforts to reduce the energy cost of scientific applications. Jiménez et al. [18] analyze the trend towards energy-efficient microprocessors. Kestor et al. [19] break down the energy cost of the data movement in a scientific application. Targeting different mainstream architectures, Aliaga et al. present in [5] an energy comparison study for the CG solver. Concerning more complex applications, Charles et al. [20] present an efficiency comparison when running the COSMO-ART simulation model [21] on different CPU architectures. For an agro-forestry application, Padoin et al. [22] investigate performance and power consumption on a hybrid CPU+GPU architecture, revealing that a changing workload can drastically improve energy efficiency. In [23], Krueger et al. compare the energy efficiency of a CPU and a GPU implementation of a seismic modeling application against a general-purpose manycore chip design called "Green Wave," that is optimized for high-order wave equations. Based on the analysis on Intel Sandy Bridge processors, Wittmann et al. [24] extrapolate the energy efficiency of a Lattice-Boltzmann CFD simulation to a petascale-class machine.

Blocked SpMV: As there exists significant need for SpMM products, NVIDIA's cuSPARSE library provides this routine for the CSR format [25]. Aside from a straight-forward implementation assuming the set of vectors being stored in column-major order, the library also contains an optimized version taking the block of vectors as a row-major matrix that can be used in combination with a preprocessing step, transposing the matrix to achieve significantly higher performance [26]. Still, we show that the SpMM product that we propose outperforms the cuSPARSE implementations.

Orthogonalizations for GPUs: Orthogonalization of vectors is a fundamental operation for both linear systems and eigenproblem solvers, and many applications. Therefore there has been extensive research on both its acceleration and stability. Besides the classical and modified Gram-Schmidt orthogonalizations [27] and orthogonalizations based on LAPACK (xGEQRF + xUNGQR) [28] and correspondingly MAGMA for GPUs [16], recent work includes communication-avoiding QR [29], also developed for GPUs [30, 31]. For tall and skinny matrices, these orthogonalizations are, in general, memory bound. Higher performance, using Level 3 BLAS, is also possible in orthogonalizations like the Cholesky QR or SVD QR, but they are less stable (error bounded by the square of the condition number of the input matrix). For the LOPBPCG method, after the SpMM kernel, the orthogonalizations are the most time consuming building block. In particular, LOBPCG contains two sets of orthogonalizations per iteration.

LOBPCG implementations: The BLOPEX package maintained by A. Knyazev is state-of-the-art for CPU implementations of LOBPCG, and popular software libraries like PETSc and Hypre provide an interface to it [15]. Also Scipy [32], Octopus [33], and Anasazi [34] as part of the Trilinos library [35] feature LOBPCG implementations. The first implementation of LOBPCG for GPUs has been available since 2011 in the ABINIT material science package [36]. It benefits from utilizing the generic linear algebra routines available in the CUDA [37] and MAGMA [16] libraries. More recently, NVIDIA announced that LOBPCG will be included in the GPU-accelerated Algebraic Multigrid Accelerator AmgX [2].

[1] The authors would like to thank the CSCS for access to Piz Daint and the support running the experiments.

[2] https://developer.nvidia.com/amgx

3. LOBPCG

The LOBPCG method [9, 38] finds m of the smallest (or largest) eigenvalues λ and corresponding eigenvectors x of a symmetric and positive definite eigenvalue problem:

$$Ax = \lambda x.$$

Similarly to other CG-based methods, this is accomplished by the iterative minimization of the Rayleigh quotient:

$$\rho(x) = \frac{x^T Ax}{x^T x}.$$

The minimization at each step is done locally, in the subspace of the current approximation x_i, the previous approximation x_{i-1}, and the preconditioned residual $P(Ax_i - \lambda_i x_i)$, where P is a preconditioner for A. The subspace minimization is done by the Rayleigh-Ritz method.

Note that the operations in the algorithm are blocked and therefore can be very efficient on modern architectures. Indeed, the AX_i is the SpMM kernel, and the bulk of the computations in the Rayleigh-Ritz minimization are general matrix-matrix products (GEMMs). The direct implementation of this algorithm becomes unstable as the difference between X_{i-1} and X_i becomes smaller, but stabilization methods can provide an efficient workaround [9, 39]. While the LOBPCG convergence characteristics usually benefit from using an application-specific preconditioner [11, 40–43], we refrain from including one as we are particularly interested in the performance of the top-level method. The LOBPCG we develop is hybrid, using both the GPUs and CPUs available. All data resides on the GPU memory and the bulk of the computation – the preconditioned residual, the accumulation of the matrices for the Rayleigh-Ritz method, and the update transformations – are handled by the GPU. The small and not easy to parallelize Rayleigh-Ritz eigenproblem is solved on the CPU using LAPACK. More specifically, to find

$$X_{i+1} = argmin_{y \in \{X_i, X_{i-1}, R\}} \rho(y),$$

the Rayleigh-Ritz method computes on the GPU

$$\begin{aligned} \tilde{A} &= [X_i, X_{i-1}, R]^T A\, [X_i, X_{i-1}, R] \\ B &= [X_i, X_{i-1}, R]^T\, [X_i, X_{i-1}, R], \end{aligned}$$

and solves the small generalized eigenproblem $\tilde{A}\ \phi = B\ \phi$ on the CPU, to finally find (computed on the GPU)

$$X_{i+1} = [X_i, X_{i-1}, R]\ \phi(1:m).$$

For stability, various orthogonalizations are performed, following the LOBPCG Matlab code from A. Knyazev [3]. We used our highly optimized GPU implementations based on the Cholesky QR to get the same convergence rates as the reference CPU implementation from BLOPEX (in HYPRE) on all our test matrices from the University of Florida sparse matrix collection (see Section 6).

4. Sparse Matrix-Vector-Block Product

To develop an efficient SpMM kernel, we use the recently proposed SELL-P format (padded sliced ELLPACK [44]). The performance numbers reported throughout this section are for double precision (DP) arithmetic.

Implementation of `SpMM` for SELL-P

In addition to the well known sparse matrix formats like CSR [45], work on efficient SpMV products for GPUs has motivated a number of new formats, and in particular, the one standing out is ELLPACK [46], where padding of the different rows with zeros is applied for a uniform row-length suitable for coalesced memory accesses of the matrix and instruction parallelism. However, the ELLPACK format incurs a storage overhead, which is determined by the maximum number of nonzero elements aggregated in one row. Depending on the particular problem, the overheads in using ELLPACK may result in poor performance, despite the coalesced memory access that is highly favorable for streaming processors.

A workaround to reduce memory and computational overhead is to split the original matrix into row blocks before converting them into the ELLPACK format. In the resulting sliced ELLPACK format (SELL or SELL-C, where C denotes the size of the row blocks [47, 48]), the overhead is no longer determined by the matrix row containing the largest number of non-zeros, but by the row with the largest number of nonzero elements in the respective block. While sliced SELL-C reduces the overhead very efficiently, e.g., choosing C=1 results in the storage-optimal CSR, assigning multiple threads to each row requires padding the rows with zeros, so that each block has a row-length divisible by this thread number. This is the underlying idea of the SELL-P format: partition the sparse matrix into row-blocks, and convert the distinct blocks into ELLPACK format [46] with the row-length of each block being padded to a multiple of the number of threads assigned to each row when computing an SpMV or SpMM product.

Although the padding introduces some zero fill-in, the comparison between the formats in Figure 3 reveals that the blocking strategy may still render significant memory savings (also see Table 1), which translates into reduced computational cost for the `SpMV` kernel. For the performance of the `SpMM` routine, it is not sufficient to reduce the computational overhead, but essential to optimize the memory access pattern. Doing so requires the accessed data to be aligned in memory whenever possible. For consecutive memory access, and with the motivation of processing multiple vectors simultaneously, we implement the `SpMM` assuming that the tall-and-skinny dense matrix composed of the vectors is stored in row-major order. Although this requires a preprocessing step of transposing the dense matrix from column to row-major format prior to the `SpMM` call, the aligned memory access to the vectors' values compensates for the extra work.

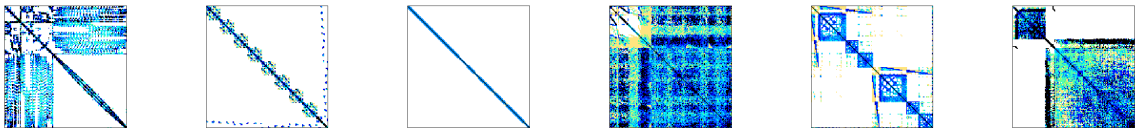
The algorithmic kernel for the `SpMM` operation arises as a natural extension of the `SpMV` kernel for the SELL-P format proposed in [44]. Like in the `SpMV` kernel, the x-dimension of the thread block processes the distinct rows of one SELL-P block, while the y-dimension corresponds to the number of threads assigned to each row (see Figure 4). Partial products are written into shared memory and added in a local reduction phase. For the `SpMM` it is beneficial to process multiple vectors simultaneously, which motivates for extending the thread block by a z-dimension, handling the distinct vectors. While assigning every z-layer of the block to one vector would provide a straight-forward implementation, keeping the set of vectors, in texture memory, makes an enhanced approach more appealing. The motivation is that in CUDA (version 6.0) every texture read fetches 16 bytes, corresponding to two IEEE double or four IEEE single precision floating point values. As using only part of them would result in performance waste, every z-layer may process two (double precision case) or four (single precision case) vectors, respectively. This implies that, depending on the precision format, the z-dimension of the thread block equals half or a quarter the column count of the tall-and-skinny dense matrix.

As assigning multiple threads to each row requires a local reduction of the partial products in shared memory (see Figure 4), the x- y- and z- dimensions are bounded by the characteristics of the GPU architecture [37]. An efficient workaround when processing a large number of vectors is given by assigning only one thread per z-dimension to each row (choose y-dimension equal 1), which removes the reduction step and the need for shared memory.

[3] http://www.mathworks.com/matlabcentral/fileexchange/48-lobpcg-m

Acronym	Matrix	#nonzeros (n_z)	Size (n)	n_z/n	n_z^{row}	ELLPACK $n_z^{ELLPACK}$	ELLPACK overhead	SELL-P n_z^{SELL-P}	SELL-P overhead
AUDI	AUDIKW_1	77,651,847	943,645	82.28	345	325,574,775	76.15%	95,556,416	18.74%
BMW	BMWCRA1	10,641,602	148,770	71.53	351	52,218,270	79.62%	12,232,960	13.01%
BONE010	BONE010	47,851,783	986,703	48.50	64	62,162,289	23.02%	55,263,680	13.41%
F1	F1	26,837,113	343,791	78.06	435	149,549,085	82.05%	33,286,592	19.38%
INLINE	INLINE_1	38,816,170	503,712	77.06	843	424,629,216	91.33%	45,603,264	19.27%
LDOOR	LDOOR	42,493,817	952,203	44.62	77	73,319,631	42.04%	52,696,384	19.36%

Table 1: Matrix characteristics and storage overhead for ELLPACK and SELL-P formats. SELL-P employs a blocksize of 8 with 4 threads assigned to each row. n_z^{FORMAT} refers to the explicitly stored elements (n_z nonzero elements plus the explicitly stored zeros for padding).



(a) AUDIKW_1 (b) BMWCRA_1 (c) BONE_010 (d) F1 (e) INLINE_1 (f) LDOOR

Figure 2: Sparsity plots of selected test matrices.

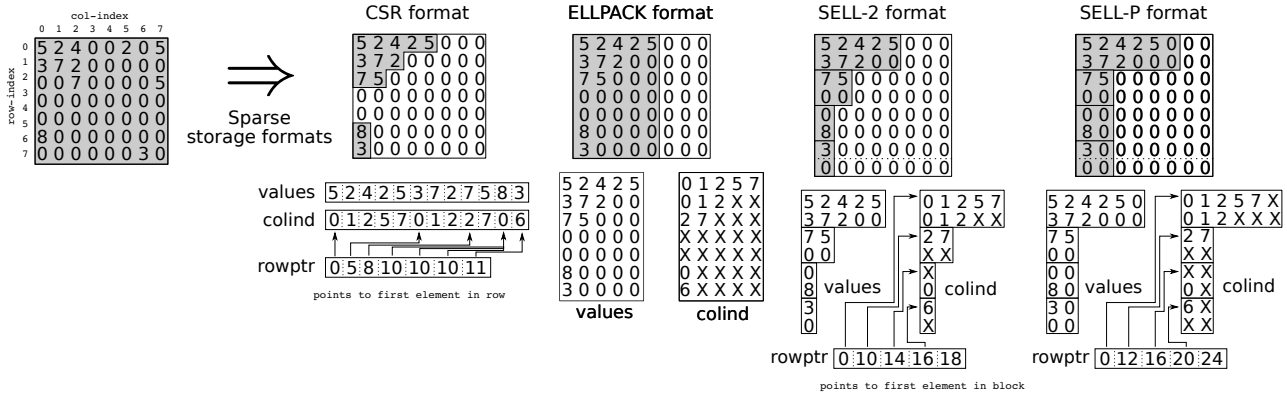


Figure 3: Visualizing the CSR, ELLPACK, SELL-C, and SELL-P formats. The memory demand corresponds to the grey areas. Note that choosing the block size 2 for SELL-C (SELL-2) and SELL-P requires adding a zero row to the original matrix. Furthermore, padding the SELL-P format to a row length divisible by 2 requires explicit storage of a few additional zeros.

Performance of `SpMM` for SELL-P

For the runtime analysis we use a Kepler K40 GPU which is newer than the K20 GPUs in the Piz Daint supercomputer. We benchmarked the SELL-P SpMV for a larger set of test matrices taken from the University of Florida Matrix Collection (UFMC)[4] than the ones we target with the LOBPCG algorithm in Section 6.

With some key characteristics collected in Table 1, and sparsity plots shown in Figure 2, we tried to cover a large variety of systems common in scientific computing. The hardware we used is a two socket Intel Xeon E5-2670 (Sandy Bridge) platform accelerated by an NVIDIA Tesla K40c GPU with a theoretical peak performance of 1,682 GFLOP/s. The host system has a theoretical peak of 333 GFLOP/s, main memory size is 64 GB, and theoretical bandwidth is up to 51 GB/s. On the K40 GPU, 12 GB of main memory are accessed at a theoretical bandwidth of 288 GB/s. The implementation of all GPU kernels is realized in CUDA [37], version 6.0 [49], while we also include in the performance comparisons routines taken from NVIDIA's cuSPARSE [25] library. On the CPU, Intel's MKL [50] is used in version 11.0, update 5.

In Figure 5, for different test matrices we visualize the performance scaling of the previously described `SpMM` kernel with respect to the number of columns in the dense matrix (equivalent to the number of vectors in a blocked `SpMV`). The results reveal that the `SpMM` performance exceeds 100 GFLOP/s as soon as the number of columns in the dense matrix exceeds 30. The characteristic oscillation of the performance can be explained by the more or less efficient memory access, but in particular the cases where the column-count equals a multiple of 16 provide very good performance. Using the `SpMM` kernel vs. a set of consecutive `SpMV` sparse products (that typically achieve less than 25 GFLOP/s on this architecture [44]; see Figure 1, Right) results in speedup factors of around five, see Table 2. Similar performance improvement (up to $6.1\times$) is observed on CPUs when replacing consecutive MKL `SpMV` kernels by the MKL `SpMM` routine, see Table 3.

While these results are obtained by assuming the performance-beneficial row-major storage of the tall-and-skinny dense matrix,

[4] UFMC; see http://www.cise.ufl.edu/research/sparse/matrices/

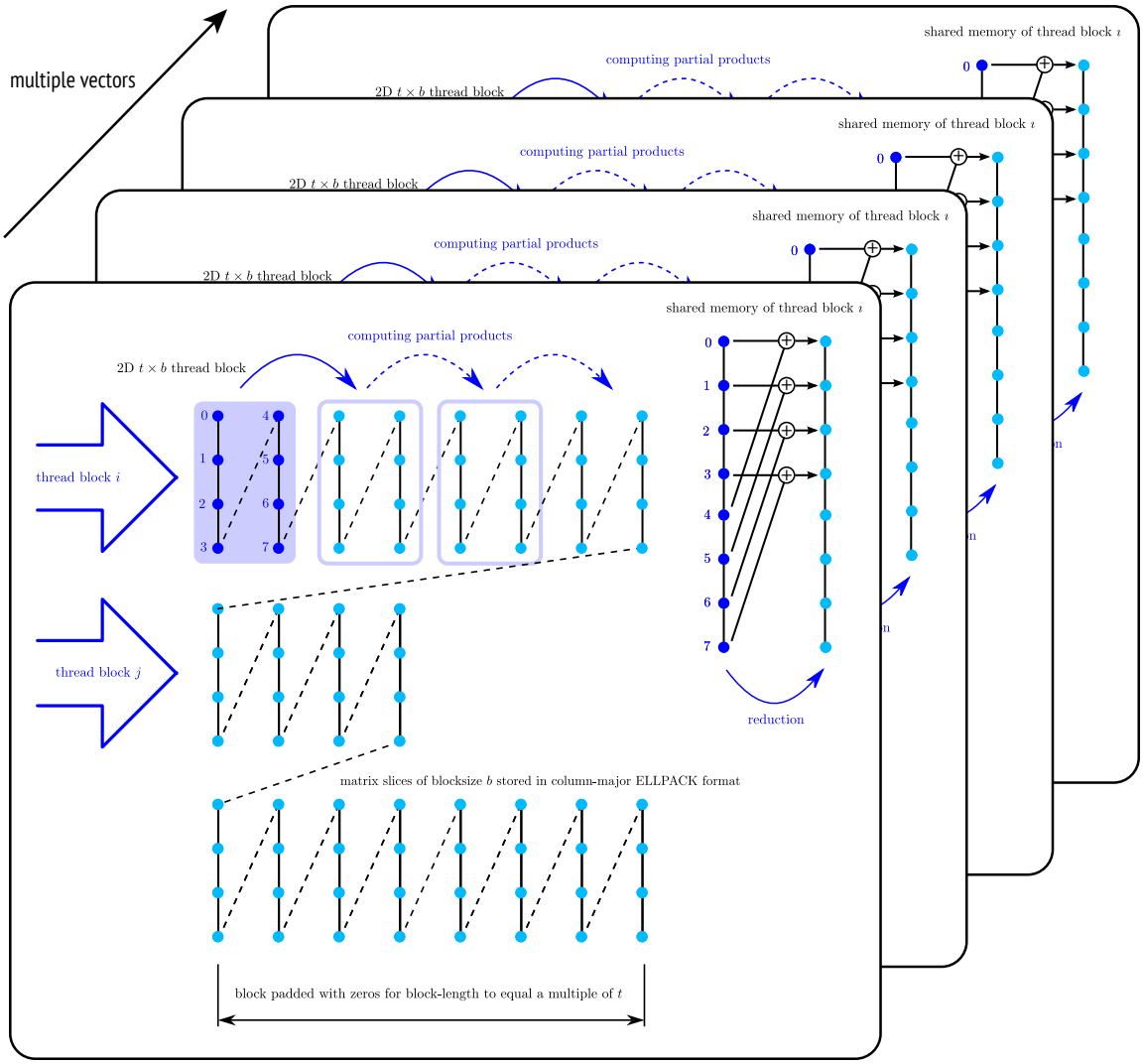


Figure 4: Visualization of the SELL-P memory layout and the SELL-P SpMM kernel including the reduction step using the blocksize $b = 4$ (corresponding to SELL-4), and always assigning two threads to every row ($t = 2$). Adding a z-dimension to the thread-block allows for processing multiple vectors simultaneously.

many applications and algorithms use dense matrices stored in column-major format to benefit from highly optimized BLAS implementations (available for matrices in column-major format). For this reason, when comparing the performance of the SELL-P implementation against the cuSPARSE CSRSpMM [25], we include the preprocessing time needed to transpose the tall-and-skinny dense matrix (see Figure 6). Aside from the standard CSRSpMM assuming column-major storage, the cuSPARSE also includes a highly tuned version assuming, like the MAGMA implementation, row-major storage [26]. Combining this with a preprocessing step transposing the matrix provides the same functionality at significantly higher GFLOP rates. While the standard SpMM achieves between 20 and 60 GFLOP/s for most matrices, the highly tuned row-major based implementation often gets close to 100 GFLOP/s, sometimes even above (see results labelled "cuSPARSE CSR-SpMM v2" in Figure 6). Our SpMM based on the SELL-P format outperforms both cuSPARSE SpMM implementations. With significant speedup factors over the standard SpMM, the performance improvement compared to the cuSPARSE SpMM ranges between 13% and 41% (see results for CANT and CRANK, respectively).

Matrix	cuSPARSE CSR	cuSPARSE HYB	MAGMA SELL-P	MAGMA SpMM	MAGMA speedup
AUDI	21.9	17.7	22.1	111.3	5.0
BMW	22.3	24.2	23.6	122.0	3.8
BONE010	15.5	25.2	22.3	121.6	4.1
F1	19.3	16.9	19.6	106.3	5.4
INLINE	20.7	19.1	21.1	108.8	3.8
LDOOR	14.9	19.3	20.7	111.2	5.4

Table 2: Asymptotic DP performance [GFLOP/s] for a large number of vectors using consecutive SpMVs (cuSPARSE CSR, cuSPARSE HYB, MAGMA SELL-P SpMV) *vs.* the SpMM kernel on GPUs. The last column is the speedup of the SpMM against the respective best SpMV. See Table 1 for the matrix characteristics.

Figure 7 compares our SpMM kernel on a K40 with the DCSRMM kernel from MKL (routine mkl_dcsrmm) on two eight-

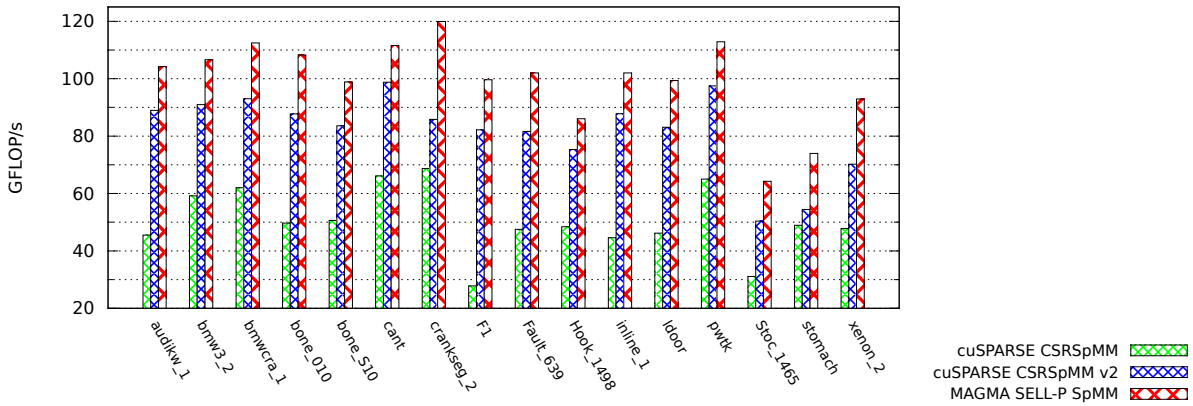


Figure 6: Performance comparison between the developed SpMM kernel and the CSRSpMM kernels provided by NVIDIA for selected matrices and a set of 64 vectors in DP arithmetic.

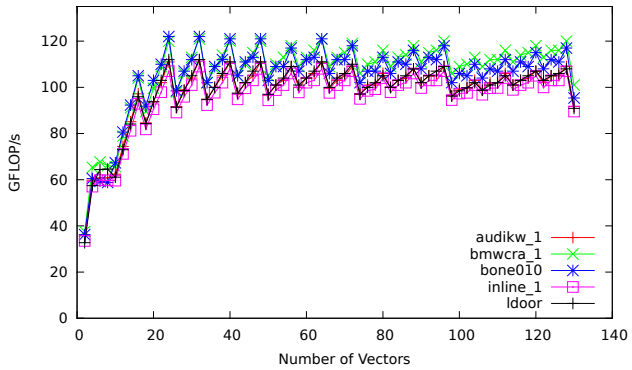


Figure 5: DP performance scaling with respect to the vector count of the SpMM kernel for the matrices listed in Table 1.

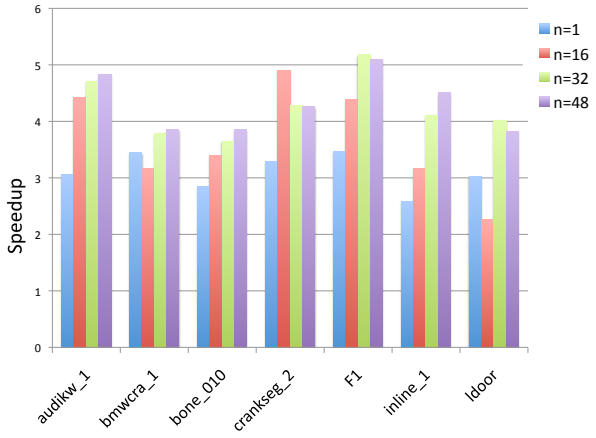


Figure 7: Speedup of the developed SpMM kernel on a K40 *vs.* the DCSRMM kernel provided by Intel's MKL on two eight-core Intel Xeon E5-2690s for selected matrices and number of vectors n.

Matrix	mkl_dcsrmv	mkl_dcsrmm	speedup
AUDI	7.24	22.5	3.1
BMW	6.86	32.2	4.7
BONE010	7.77	30.5	3.9
F1	5.64	20.1	3.6
INLINE	8.10	28.9	3.6
LDOOR	6.78	41.5	6.1

Table 3: Asymptotic DP performance [GFLOP/s] of sparse test matrices and a large number of vectors with a set of consecutive SpMVs *vs.* blocked SpMVs (SpMM) on CPUs using MKL. The last column is the speedup of the SpMM kernel against the SpMV.

core Intel Xeon E5-2690s for selected matrices and number of vectors n. Both implementations assume the vectors to be multiplied by the sparse matrix to be stored in row-major data format. On both architectures, the row-major storage allows for significantly higher GFLOP rates. The CPU runs are using the `numactl --interleave=all` policy, which is well known to improve performance. The performance obtained is consistent with benchmarks provided by Intel [6]. The results show a 3 to 5× acceleration from CPU to GPU implementation, which is expected from the compute and bandwidth capabilities of the two architectures.

5. Experiment Framework

The Piz Daint Supercomputer at the Swiss National Computing Centre in Lugano was listed in June 2014 as the sixth fastest supercomputer according to the TOP500 [3], while its energy efficiency ranked number five in the Green500 [2]. A single node is equipped with an 8-core 64-bit Intel Sandy Bridge CPU (Intel Xeon E5-2670), an NVIDIA Tesla K20X with 6 GB GDDR5 memory, and 32 GB of host memory. The nodes are connected by the "Aries" proprietary interconnect from Cray, with a dragonfly network topology [51]. Piz Daint has 5,272 compute nodes, corresponding to 42,176 CPU cores in total - with the ability to use up to 16 virtual cores per node when hyperthreading (HT) is enabled, i.e., 84,352 virtual cores in total, and 5,272 GPUs. The peak performance is 7.8 Petaflops [51]. PMDB enables the user to monitor power and energy usage for the host node and separately for the accelerator at a frequency of 10 Hz [52]. For the BLOPEX code running exclusively on the CPU of the host nodes, we obtain the pure CPU power by subtracting the power draft of the (idle) GPUs.

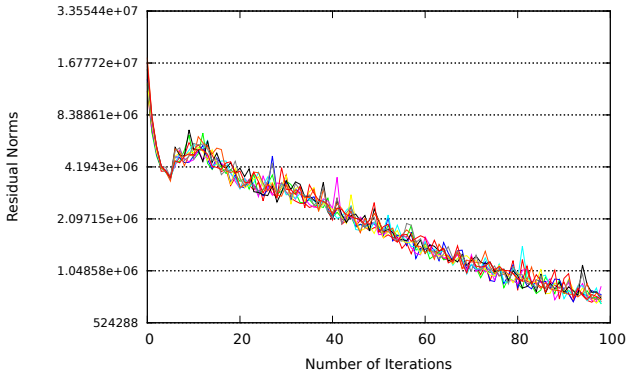


Figure 8: Visualizing the convergence of 10 eigenvectors when applying the developed GPU implementation of LOBPCG based on the SpMM kernel to the AUDI test matrix.

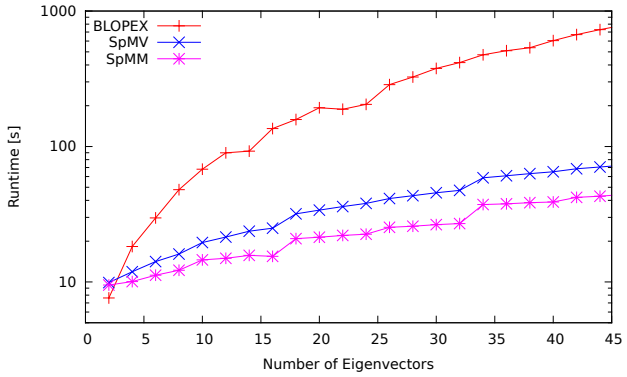


Figure 9: Runtime to complete 100 DP iterations on the AUDI problem using either the BLOPEX via the Hypre interface on 4 nodes (32 threads), or the GPU implementation using either a sequence of SpMVs or the SpMM kernel.

6. Runtime and Energy Analysis of LOBPCG

In this Section we quantify the runtime and energy efficiency of two LOBPCG implementations: the GPU-accelerated LOBPCG, and the multithreaded CPU implementation from BLOPEX [14]. All computations use DP arithmetic. For the runtime and energy analysis, we used the BLOPEX code via the Hypre interface, assigning 8 threads to each node (one thread per core) for up to 128 nodes (1024 cores) of the hardware platform listed in Section 5. We note that we use the BLOPEX LOBPCG out-of-the-box, without attempting any optimizations that are not included in the software library. The convergence rate of the GPU implementation is matching the one from BLOPEX. In Figure 8 we visualize the convergence of 10 eigenvectors for the AUDI test matrix. As convergence properties are not the focus of the research, all further results are based on 100 iterations of either implementation.

The LOBPCG implementation in BLOPEX is matrix free, i.e., the user is allowed to provide their choice of SpMV/SpMM implementation. In these experiments we use the Hypre interface to BLOPEX, linked with the MKL library.

The number of operations executed in every iteration of LOBPCG can be approximated by

$$2 \cdot nnz \cdot n_v + 36 \cdot n \cdot n_v^2 \qquad (1)$$

where nnz denotes the number of non-zeros of the sparse matrix, n the dimension, and n_v the number of eigenvectors (equivalent to the number of columns in the tall-and-skinny dense matrix). The first term of the sum reflects the SpMM operation generating the Krylov vectors, while the second contains the remaining operations including the orthogonalizations. Due to the n_v^2 term, the runtime is expected to increase superlinearly with the number of vectors. This can be observed in Figure 9 where we visualize the time needed to complete 100 iterations on the AUDI problem using either the BLOPEX code via the Hypre interface on 4 nodes (32 threads), or the GPU implementation using either a sequence of SpMVs or the SpMM kernel. While the BLOPEX implementation also shows some variances for different numbers of vectors, the runtime pattern of the GPU LOBPCG reflects the efficiency of the orthogonalization routines favoring cases where 16, 32, or 48 vectors are processed. This characteristic pattern remains when replacing the consecutive SpMVs with the SpMM, as this kernel also promotes certain column-counts of the tall and skinny dense matrix. Figure 10 shows the runtime of the SpMM-based GPU implementation of LOBPCG to complete 100 iterations for different test matrices. Comparing the results for the AUDI problem, we are 1.3

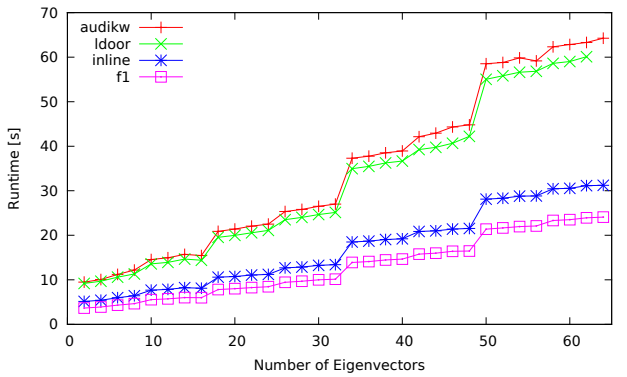


Figure 10: Runtime needed to complete 100 iterations in DP using the LOBPCG GPU implementation based on SpMM.

and 1.2× faster when computing 32 and 48 eigenvectors, respectively, using the SpMM instead of the SpMV in the GPU implementation of LOBPCG. Note that although in this case the SpMM performance is about 5× the SpMV performance, the overall improvement of correspondingly 30% and 20% reflects that only 12.5% and 8.7% of the overall LOBPCG FLOP/s are in SpMVs for the 32 and 48 eigenvector problems, respectively (see equation (1) and the matrix specifications in Table 1). While the BLOPEX implementation also shows some variances for different numbers of vectors, the runtime pattern of the GPU LOBPCG reflects the efficiency of the orthogonalization routines favoring cases where 16, 32, or 48 vectors are processed. This characteristic pattern is even amplified when replacing the consecutive SpMVs with the SpMM, as this kernel also promotes certain column-counts of the tall and skinny dense matrix, as shown in Figure 10.

Figure 11 shows the scaling of the Hypre implementation when computing a set of eigenvectors for the AUDI problem. Taking one node (8 threads) as the baseline configuration, the algorithm scales almost linearly up to 16 nodes. The associated energy need remains constant within the linear scaling range, but rises as soon as the speedup is smaller than the node increase. We further observe that the Hypre LOBPCG scales almost independently of the number of eigenvectors to compute. We define the computation of a set of 16 and 32 eigenvectors as the two benchmarks for the further analysis. In a first step, we investigate how energy efficiency and runtime performance are related for the BLOPEX implementation.

For this, we evaluate these metrics when running on 1, 2, 4, 8, 16, 32, 64, and 128 nodes, and identify the configuration providing the best runtime performance and energy efficiency (see Table 4). Due to the non-linear scaling of the BLOPEX implementation, and the power draft increasing almost linearly with the hardware resources, optimizing for runtime results in significant overhead for energy efficiency and vice versa. While this is different for the hybrid implementation of LOBPCG using a single CPU+GPU node, the energy balance has to also account for the power draft of the GPU (see Table 5). Theoretically, each node of Piz Daint is equipped with a multicore Sandy Bridge CPU that provides a theoretical peak of 166.4 GFLOP/s at a power consumption of 115W (1.44 GFLOP/W), whereas the K20X GPU consumes 225W when providing 1311 GFLOP/s (5.83 GFLOP/W) [52]. Hence, assuming full load, adding the accelerator theoretically increases the power need per node by a factor of around 1.67. However, the GPU-acclerated LOBPCG uses both the CPU host and the GPU, but not heavily. The power usage therefore scales according to the load. Hence, the energy improvement depends on the specific test case, but is in general smaller than the runtime improvement when comparing the BLOPEX implementation running on one node with the hybrid code. This is reflected in the first line of Table 6, where we list the runtime and energy requirement of the BLOPEX LOBPCG scaled to the MAGMA LOBPCG results (this is equivalent to the speedup and greenup of the MAGMA implementation over the BLOPEX implementation). When computing 16 eigenvectors (see speedup shown on the left top of Table 6), increasing the computing resources enables the BLOPEX implementation to outperform the MAGMA LOBPCG for some test cases; however, this comes at the cost of a significantly higher energy usage (see greenup of MAGMA over BLOPEX on the right top of Table 6). With the greenup ranging between 4 and 180, the BLOPEX code is, for no configuration, even close to the energy efficiency of the GPU-accelerated solver. Targeting the computation of 32 eigenvectors, speedup and greenup grow even more (see bottom of Table 6).

To complete the performance analysis, we concentrate on the single node performances in Table 6. Based on the `SpMM` kernel analysis in Figure 7, the expectation for 16 vectors is that an optimized CPU code (blocking the `SpMV`s) must be only about $5\times$ slower than the GPU code. The $10\times$ acceleration indicates that the Hypre interface to BLOPEX is probably not blocking the `SpMV`s. Computing more vectors (e.g., 32, shown in Table 6, bottom) reduces the fraction of `SpMV` FLOP/s to the total FLOP/s (see equation (1)), and thus making the `SpMV` implementation less critical for the overall performance. The fact that the speedup of the GPU *vs.* the CPU LOBPCG continues to grow, reaching about $30\times$, shows that there are other missed optimization opportunities in the CPU implementation. In particular, these are where the majority of the FLOP/s are – the GEMMs in assembling the matrix representations for the local Rayleigh-Ritz minimizations and the orthogonalizations. These routines are highly optimized in our GPU implementation, especially the GEMMs, which due to the specific sizes of the matrices involved – tall and skinny matrices A and B with a small square resulting matrices $A^T B$ – required modifications to the standard GEMM algorithm for large matrices [53]. What worked very well is splitting the $A^T B$ GEMM into smaller GEMMs based on tuning the MAGMA GEMM [53] for the particular small sizes, all grouped for execution into a single batched GEMM, followed by the addition of the local results [54].

As a bottom-line, we observe that using more hardware resources may enable a scalable CPU-based algorithm to keep up with the performance of the hybrid implementation, but this increasingly fails to provide the energy efficiency desired.

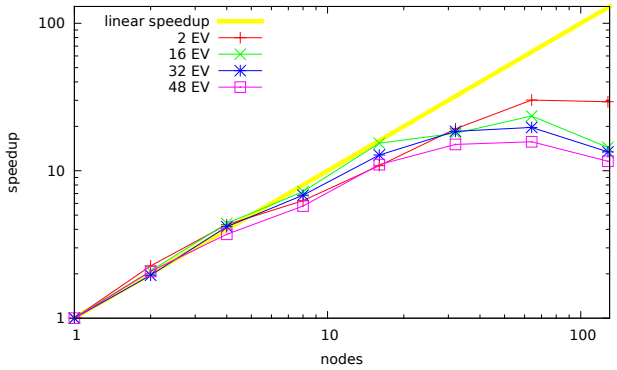


Figure 11: Scaling of the Hypre LOBPCG using 8 threads/node when computing a set of eigenvectors for the AUDI problem.

matrix	best performance nodes	time [s]	energy [J]	best energy efficiency nodes	time [s]	energy [J]
AUDI	64	8.88	73956	8	24.21	27611
BMW	8	3.59	3713	2	7.57	2291
BONE010	64	6.06	38656	8	15.33	18078
F1	32	8.12	32179	4	17.20	9587
INLINE	64	6.22	46744	4	18.08	10558
LDOOR	64	7.14	56041	8	14.41	16976

matrix	best performance nodes	time [s]	energy [J]	best energy efficiency nodes	time [s]	energy [J]
AUDI	32	34.86	151800	2	307.20	96060
BMW	8	14.41	14564	2	30.20	9215
BONE010	64	23.85	192492	2	258.93	81873
F1	32	23.86	101979	1	206.64	31659
INLINE	32	24.31	105512	8	36.73	42382
LDOOR	32	24.93	107172	2	233.35	74638

Table 4: Runtime and energy consumption of 100 iterations in DP of the BLOPEX LOBPCG computing 16 (top) and 32 (bottom) eigenvectors when optimizing the hardware configuration (number of nodes #) w.r.t. performance (left) or energy efficiency (right).

matrix	EVs	time [s]	energy CPU [J]	energy GPU [J]
AUDI	16	14.69	1075.00	1664.00
	32	19.47	1377.00	2539.00
BMW	16	2.54	182.00	232.00
	32	3.37	241.00	375.00
BONE010	16	14.58	996.00	1609.00
	32	18.97	1307.00	2456.00
F1	16	5.57	390.00	581.00
	32	7.40	536.00	958.00
INLINE	16	7.71	602.00	906.00
	32	10.11	709.00	1248.00
LDOOR	16	13.87	951.00	1537.00
	32	18.39	1283.00	2368.00

Table 5: Runtime and energy balance of 100 iterations of the GPU-accelerated LOBPCG in DP.

	speedup						greenup					
nodes	audi	bmw	bone	F1	inline	ldoor	audi	bmw	bone	F1	inline	ldoor
1	14.88	7.00	11.37	12.35	11.27	10.61	12.37	6.56	9.84	10.69	8.88	9.22
2	6.56	2.98	5.25	5.89	5.93	5.10	10.98	5.53	9.18	10.23	9.42	9.01
4	3.36	1.88	2.68	3.09	2.35	2.64	10.92	6.32	9.07	9.87	7.00	9.04
8	1.65	1.41	1.05	2.02	1.46	1.04	10.24	8.97	6.94	12.66	8.20	6.82
16	1.97	2.38	1.36	2.02	2.16	1.71	23.27	24.73	16.72	24.96	23.98	20.71
32	0.73	1.62	0.45	1.46	0.88	0.53	16.99	41.73	10.13	33.14	17.94	11.61
64	0.60	1.99	0.42	1.86	0.81	0.51	27.42	97.39	14.84	83.50	31.00	22.52
128	0.69	2.60	0.50	2.12	1.23	0.55	47.80	231.79	37.36	190.68	86.05	49.36

	speedup						greenup					
nodes	audi	bmw	bone	F1	inline	ldoor	audi	bmw	bone	F1	inline	ldoor
1	33.60	23.09	28.01	27.92	28.88	27.43	26.12	19.48	22.02	21.19	23.19	21.62
2	15.78	8.96	13.65	15.00	14.97	12.69	24.87	14.96	21.76	22.78	24.09	20.44
4	9.78	4.95	7.24	7.48	9.02	7.20	29.59	15.23	22.46	21.69	27.73	22.32
8	5.14	4.28	4.03	4.98	3.63	3.82	30.34	23.64	24.24	27.14	21.66	23.09
16	4.37	4.63	3.94	4.79	4.46	3.89	48.11	54.70	43.88	51.92	50.09	43.05
32	1.79	5.17	1.32	3.22	2.40	1.36	39.31	117.84	27.75	68.26	53.92	29.35
64	1.80	7.91	1.26	4.03	2.51	1.54	73.85	351.47	51.15	163.09	106.38	63.08
128	1.93	7.61	1.50	5.01	3.08	1.67	153.09	682.25	121.28	410.40	262.81	136.72

Table 6: Speedup and greenup of the MAGMA LOBPCG *vs.* the CPU LOBPCG when computing 16 (top) or 32 (bottom) eigenvectors.

7. Summary and Outlook

In this paper we have compared the performance and energy efficiency of a CPU and a hybrid CPU+GPU LOBPCG eigensolver on the Piz Daint supercomputer. For the GPU-accelerated LOBPCG, we provide a comprehensive description of the GPU-kernel used to generate the Krylov subspace by computing the `SpMM` product. As this building block also serves as the backbone for other block-Krylov methods, we include a runtime analysis that reveals significant speedups over a set of consecutive `SpMV`s, and an equivalent routine provided in NVIDIA's cuSPARSE library. Integrating it into the GPU implementation of LOBPCG, we outperform the BLOPEX CPU implementation running on one node by more than an order of magnitude for both, runtime performance and energy efficiency, when computing a set of eigenvectors. Increasing the resources for the BLOPEX code improves runtime performance at the cost of a rising energy balance. This indicates, that resource efficiency of scientific applications can be improved by utilizing GPU accelerators lauded for their high GFLOP/W ratios.

Various insights regarding energy efficiency can be extracted from the results presented. For example, as a general rule Krylov space methods achieve (on systems comparable to the one tested) about 0.033 GFLOP/W on CPUs vs. 0.2 GFLOP/W on GPUs. This makes GPUs about six times more energy efficient. Performance-wise, two Sandy Bridge CPUs achieve around 5 GFLOP/s vs. around 20 GFLOP/s for a K20 GPU. Further, employing techniques to reduce communications has a significant impact. In particular, blocking of size 16 in kernels like `SpMM`, brings the GPU's energy efficiency to about 1 GFLOP/W and the performance to about 100 GFLOP/s. Compared to the baseline `SpMV`-based method this brings 30x greenup. The greenup for an entire application would depend on the other operations as well, e.g., being around $7\times$ for the LOBPCG for 16 vectors (see Table VI), and converging to the asymptotic $30\times$ as the number of vectors grows (and more if the performance of the CPU implementation does not scale with increasing the number of nodes used).

The results presented were shown to be state-of-the-art, and therefore to be indicative of today's frontiers in energy efficiency and performance for sparse computations on supercomputers. Indeed, the performance and energy efficiency on other optimizations techniques can be evaluated based on their expected FLOP/Byte ratio, and extrapolated from the results for the basic building blocks presented (like the `SpMV` and `SpMM`, and their interactions with other BLAS kernels). Future work includes simplifying this process of evaluation by setting up MAGMA's basic kernels as benchmarks, and their use in the development of performance and energy efficiency evaluation tools for sparse computing applications.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, Department of Energy grant No. DE-SC0010042, the Russian Scientific Fund (Agreement N14-11-00190), and NVIDIA. The authors would like to thank the Swiss National Computing Centre for granting access to their system, and support in deploying the energy measurements.

References

[1] J. Dongarra *et al*, "The international ExaScale software project roadmap," *IJHPCA*, vol. 25, no. 1, 2011.

[2] The green 500 list, `http://www.green500.org/`.

[3] The top 500 list, `http://www.top.org/`.

[4] J. Dongarra and M. A. Heroux, "Toward a New Metric for Ranking High Performance Computing Systems," SANDIA REPORT SAND2013-4744, June 2013.

[5] J. Aliaga, H. Anzt, M. Castillo, J. Fernández, G. Léon, J. Pérez, and E. Quintana-Ortí, "Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors," *Concurrency and Computation: Practice and Experience*, 2014.

[6] "Intel® Math Kernel Library. Sparse BLAS and Sparse Solver Performance Charts: DCSRGEMV and DCSRMM," October 2014. [Online]. Available: `https://software.intel.com/en-us/intel-mkl`

[7] M. F. Hoemmen, "Communication-avoiding krylov subspace methods," Ph.D. dissertation, EECS Department, UC, Berkeley, Apr 2010.

[8] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, "Improving the performance of CA-GMRES on multicores with multiple GPUs," in *IPDPS'14*. Washington, DC, USA: 2014, pp. 382–391.

[9] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM J. Sci. Comput*, vol. 23, pp. 517–541, 2001.

[10] S. Tomov, J. Langou, J. Dongarra, A. Canning, and L.-W. Wang, "Conjugate-gradient eigenvalue solvers in computing electronic proper-

ties of nanostructure architectures." *IJCSE*, vol. 2, no. 3/4, pp. 205–212, 2006.

[11] C. Vömel, S. Tomov, O. A. Marques, A. Canning, L.-W. Wang, and J. J. Dongarra, "State-of-the-art eigensolvers for electronic structure calculations of large scale nano-systems." *J. Comput. Physics*, vol. 227, no. 15, pp. 7113–7124, 2008.

[12] S. Yamada, T. Imamura, and M. Machida, "16.447 tflops and 159-billion-dimensional exact-diagonalization for trapped fermion-hubbard model on the earth simulator," in *Proc. of SC'05*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, p. 44.

[13] S. Yamada, T. Imamura, T. Kano, and M. Machida, "High-performance computing for exact numerical approaches to quantum many-body problems on the earth simulator," in *Proc. of the ACM/IEEE SC'06*. New York, NY, USA: ACM, 2006.

[14] A. Knyazev. https://code.google.com/p/blopex/.

[15] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov, "Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc." *SIAM J. Scientific Computing*, vol. 29, no. 5, pp. 2224–2239, 2007.

[16] I. C. Lab, "Software distribution of MAGMA version 1.5," `http://icl.cs.utk.edu/magma/`, 2014.

[17] D. Donofrio, L. Oliker, J. Shalf, M. F. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin, "Energy-efficient computing for extreme-scale science," *Computer*, vol. 42, no. 11, pp. 62–71, 2009.

[18] V. Jiménez, R. Gioiosa, E. Kursun, F. Cazorla, C.-Y. Cher, A. Buyuktosunoglu, P. Bose, and M. Valero, "Trends and techniques for energy efficient architectures," in *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, Sept 2010, pp. 276–279.

[19] G. Kestor, R. Gioiosa, D. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 56–65.

[20] J. Charles, W. Sawyer, M. F. Dolz, and S. Catalán, "Evaluating the performance and energy efficiency of the cosmo-art model system," *Computer Science - Research and Development*, pp. 1–10, 2014.

[21] C. e. a. Knote, "Towards an online-coupled chemistry-climate model: evaluation of trace gases and aerosols in cosmo-art," *Geoscientific Model Development*, vol. 4, no. 4, pp. 1077–1102, 2011.

[22] E. Padoin, L. Pilla, F. Boito, R. Kassick, P. Velho, and P. Navaux, "Evaluating application performance and energy consumption on hybrid CPU+GPU architecture," *Cluster Computing*, vol. 16, no. 3, pp. 511–525, 2013.

[23] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund, "Hardware/software co-design for energy-efficient seismic modeling," in *Proc. of SC'11*. New York, NY, USA: ACM, 2011, pp. 73:1–73:12.

[24] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein, "An analysis of energy-optimized lattice-boltzmann cfd simulations from the chip to the highly parallel level," *CoRR*, vol. abs/1304.7664, 2013.

[25] NV, *CUSPARSE LIBRARY*, July 2013.

[26] M. Naumov, "Preconditioned block-iterative methods on gpus," *PAMM*, vol. 12, no. 1, pp. 11–14, 2012.

[27] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[28] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' Guide (Third Ed.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[29] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-avoiding parallel and sequential QR factorizations," *CoRR*, vol. abs/0806.2159, 2008.

[30] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding QR decomposition for GPUs," EECS Department, UC, Berkeley, Tech. Rep. UCB/EECS-2010-131, Oct 2010.

[31] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple gpu accelerators." in *IPDPS*. IEEE, 2011, pp. 932–943.

[32] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online: `http://www.scipy.org/`].

[33] A. Castro *et al*, "Octopus: a tool for the application of time-dependent density functional theory," *phys. stat. sol. (b)*, vol. 243, no. 11, pp. 2465–2488, 2006.

[34] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist, "Anasazi software for the numerical solution of large-scale eigenvalue problems," *ACM TOMS*, vol. 36, no. 3, pp. 13:1–13:23, Jul. 2009.

[35] M. Heroux *et al*, "An Overview of Trilinos," Sandia National Laboratories, Tech. Rep. SAND2003-2927, 2003.

[36] X. G. et al., "First-principles computation of material properties: the ABINIT software project," *Computational Materials Science*, vol. 25, no. 3, pp. 478 – 492, 2002.

[37] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 6th ed., NVIDIA Corporation, April 2014.

[38] A. V. Knyazev, "Preconditioned eigensolvers - an oxymoron?" *ETNA*, vol. 7, pp. 104–123, 1998.

[39] U. Hetmaniuk and R. Lehoucq, "Basis selection in LOBPCG," *Journal of Computational Physics*, vol. 218, no. 1, pp. 324 – 332, 2006.

[40] P. Arbenz and R. Geus, "Multilevel preconditioned iterative eigensolvers for Maxwell eigenvalue problems," *Applied Numerical Mathematics*, vol. 54, no. 2, pp. 107 – 121, 2005, 6th IMACS International Symposium on Iterative Methods in Scientific Computing.

[41] P. Benner and T. Mach, "Locally optimal block preconditioned conjugate gradient method for hierarchical matrices," *PAMM*, vol. 11, no. 1, pp. 741–742, 2011.

[42] T. V. Kolev and P. S. Vassilevski, "Parallel eigensolver for H(curl) problems using H1-auxiliary space AMG preconditioning," LLNL, Livermore, CA, Tech. Rep. UCRL-TR-226197, 2006.

[43] A. Knyazev and K. Neymeyr, *Efficient Solution of Symmetric Eigenvalue Problems Using Multigrid Preconditioners in the Locally Optimal Block Conjugate Gradient Method*, ser. UCD/CCM report. University of Colorado at Denver, 2001.

[44] H. Anzt, S. Tomov, and J. Dongarra, "Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs," University of Tennessee, Tech. Rep. ut-eecs-14-727, March 2014.

[45] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[46] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Dec. 2008.

[47] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *Proc. of HiPEAC'10*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–125.

[48] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for modern processors with wide simd units," *CoRR*, vol. abs/1307.6209, 2013.

[49] N. Corp., *NVIDIA CUDA TOOLKIT V6.0*, July 2013.

[50] "Intel® Math Kernel Library for Linux* OS," Document Number: 314774-005US, October 2007, Intel Corporation.

[51] (2014) Piz Daint Computing Resources. Swiss National Computing Centre.

[52] G. Fourestey, B. Cumming, L. Gilly, and T. C. Schulthess. (2014, August) First Experiences With Validating and Using the Cray Power Management Database Tool.

[53] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 4, pp. 511–515, Nov. 2010.

[54] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra, "Mixed-precision orthogonalization scheme and adaptive step size for ca-gmres on gpus," *VECPAR 2014 (Accepted)*, jan 2014.