Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–16
© The Author(s) 2016
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342016672081
hpc.sagepub.com
SAGE

On the performance and energy efficiency of sparse linear algebra on GPUs

Hartwig Anzt[1], Stanimire Tomov[1] and Jack Dongarra[1,2,3]

Abstract
In this paper we unveil some performance and energy efficiency frontiers for sparse computations on GPU-based supercomputers. We compare the resource efficiency of different sparse matrix–vector products (SpMV) taken from libraries such as cuSPARSE and MAGMA for GPU and Intel's MKL for multicore CPUs, and develop a GPU sparse matrix–matrix product (SpMM) implementation that handles the simultaneous multiplication of a sparse matrix with a set of vectors in block-wise fashion. While a typical sparse computation such as the SpMV reaches only a fraction of the peak of current GPUs, we show that the SpMM succeeds in exceeding the memory-bound limitations of the SpMV. We integrate this kernel into a GPU-accelerated Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) eigensolver. LOBPCG is chosen as a benchmark algorithm for this study as it combines an interesting mix of sparse and dense linear algebra operations that is typical for complex simulation applications, and allows for hardware-aware optimizations. In a detailed analysis we compare the performance and energy efficiency against a multi-threaded CPU counterpart. The reported performance and energy efficiency results are indicative of sparse computations on supercomputers.

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 (Dongarra et al., 2011). Providing suitable infrastructure poses a significant challenge, and with a rough cost of US$1 million 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 17 of the 18 greenest systems are accelerated by GPUs, according to the November 2014 Green500 list,[1] which ranks supercomputers according to their performance/Watt ratio on the HPL benchmark.[2] A drawback of this ranking is that HPL provides GFlop/s numbers that are usually 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 when running scientific applications (Dongarra and Heroux, 2013).

In this paper we unveil some performance and energy efficiency frontiers for sparse computations on GPU-based supercomputers. We focus in particular on GPU kernel optimizations for the memory-bound sparse matrix–vector product (`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 per cent of the peak of current GPUs, as quantified in Figure 1. 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 (Hoemmen, 2010; Yamazaki et al., 2014). The improvement in these methods comes from grouping `SpMV` s into a (so-called) Matrix Powers Kernel (MPK), which allows reuse of the sparse matrix and vector reads during the computation of a Krylov subspace. We consider in this paper a different approach that is based on building

[1]University of Tennessee, Knoxville, USA
[2]Oak Ridge National Laboratory, USA
[3]University of Manchester, UK

Corresponding author:
Hartwig Anzt, University of Tennessee, 1122 Volunteer Boulevard, Suite 203, Claxton, Knoxville, TN 37996, USA.
Email: hanzt@icl.utk.edu

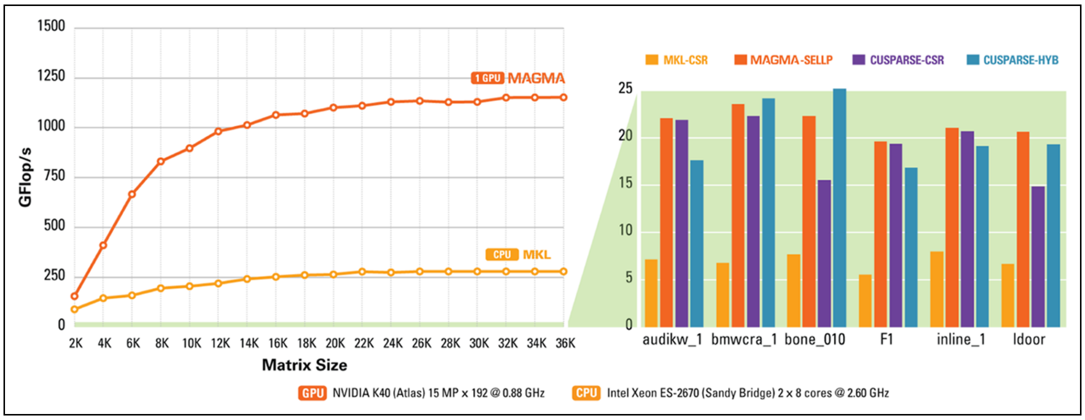

Figure 1. NVIDIA K40 GPU computing efficiency on compute intensive (left: dense LU in double precision) versus memory-bound computation (right: `SpMV` in double precision for various data formats) on state-of-the-art hardware and software libraries. CPU runs use MKL (denoted MKL-CSR) with the `numactl -interleave=all` policy (see also MKL's benchmarks (Intel Corporation, 2014)).

block-Krylov subspace. Higher computational intensity is achieved by multiplying several vectors simultaneously in a sparse matrix–matrix product (`SpMM`) kernel. Communication is reduced in this case by accessing the matrix entries only once, and reusing them for the multiplication with all vectors, which results in significant performance benefits. As the `SpMM`'s communication volume is less than the MPK's, the `SpMM` performance can be useful in modeling/predicting the upper performance bounds of s-step and communication-avoiding Krylov methods that rely on MPKs. Furthermore, `SpMMs` are needed in numerical algorithms such as the Discontinuous Galerkin, high-order finite element method (FEM), cases when vector quantities are simulated, or in specially designed block solvers, such as the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) used for finding a set of smallest/largest eigenstates of an SPD matrix (Knyazev, 2001). We chose the LOBPCG method as a benchmark algorithm as it provides an interesting mix between sparse and dense linear algebra operations, that is typical for complex simulation applications. It also serves as a backbone for many scientific simulations, e.g. in quantum mechanics, where eigenstates and molecular orbitals are defined by eigenvectors, or principle component analysis (PCA). Nonlinear Conjugate Gradient (CG) methods have been successfully used in predicting the electronic properties of large nanostructures (Tomov et al., 2006; Vömel et al., 2008), where the LOBPCG method in particular has shown higher performance and improved convergence (reduced number of `SpMV` products compared with non-blocked CG methods). Applying this algorithm efficiently to multi-billion size problems served as the backbone of two Gordon–Bell Prize finalists who ran many-body simulations on the Japanese Earth Simulator (Yamada et al., 2006, 2005).

To put the GPU results in the context of traditional processors, we include results for multicore CPUs. We compare the GPU `SpMV` and `SpMM` implementations with MKL's counterparts, and the GPU-accelerated LOBPCG solver against the multithreaded CPU implementation available in BLOPEX,[3] for which the PETSc and Hypre libraries provide an interface (Knyazev et al., 2007). Our GPU implementations are available through the open-source MAGMA library starting from MAGMA 1.5 (Innovative Computing Laboratory, UTK, 2014b). The target platforms are K20 and K40 GPUs, and the Piz Daint Supercomputer located at the Swiss National Computing Centre (CSCS).[4]

2 Related work

2.1 Energy efficiency

An overview of energy-efficient scientific computing on extreme-scale systems is provided in Donofrio et al. (2009), where the authors address both hardware and algorithmic efforts to reduce the energy cost of scientific applications. Jiménez et al. (2010) analyze the trend towards energy-efficient microprocessors. Kestor et al. (2013) break down the energy cost of the data movement in a scientific application. Targeting different mainstream architectures, Aliaga et al. (2015) present an energy comparison study for the CG solver. Concerning more complex applications, Charles et al. (2015) present an efficiency comparison when running the COSMO-ART simulation model (Knote, 2011) on different CPU architectures. For an agroforestry application, Padoin et al. (2013) investigate performance and power consumption on a hybrid CPU + GPU architecture, revealing that a changing workload can drastically improve energy efficiency. Krueger et al. (2011) 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. (2013) extrapolate the energy efficiency of a lattice-Boltzmann computational fluid dynamics (CFD) simulation to a petascale-class machine.

2.2 Blocked SpMV

As there exists significant need for SpMM products, NVIDIA's cuSPARSE library provides this routine for the CSR format.[5] Aside from a straightforward 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 (Naumov, 2012). Still, we show that the SpMM product that we propose outperforms the cuSPARSE implementations.

2.3 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 (Golub and Van Loan, 1996) and orthogonalizations based on LAPACK (xGEQRF + xUNGQR) (Anderson et al., 1999) and correspondingly MAGMA for GPUs (Innovative Computing Laboratory, UTK, 2014a), recent work includes communication-avoiding QR (Demmel et al., 2008), also developed for GPUs (Agullo et al., 2011; Anderson et al., 2010). For tall and skinny matrices, these orthogonalizations are, in general, memory bound. Higher performance, using Level 3 BLAS, is also possible in orthogonalizations such as the Cholesky QR or SVD QR (Stathopoulos and Wu, 2002), but they are less stable (error bounded by the square of the condition number of the input matrix). Recently, more stable versions were developed using mixed-precision arithmetic (Yamazaki et al., 2015). For the LOBPCG method, after the SpMM kernel, the orthogonalizations are the most time-consuming building block. In particular, LOBPCG contains two sets of orthogonalizations per iteration.

2.4 LOBPCG implementations

The BLOPEX package maintained by A. Knyazev is state-of-the-art for CPU implementations of LOBPCG, and popular software libraries such as PETSc and Hypre provide an interface to it (Knyazev et al., 2007). Also Scipy (Jones et al., 2001–2013), Octopus (Castro et al., 2006), and Anasazi (Baker et al., 2009) as part of the Trilinos library (Heroux et al., 2003) feature LOBPCG implementations. The first implementation of LOBPCG for GPUs has been available since 2011 in the ABINIT material science package (Gonze et al., 2002). It benefits from utilizing the generic linear algebra routines available in the CUDA (NVIDIA, 2009) and MAGMA (Innovative Computing Laboratory, UTK, 2014a) libraries. More recently, NVIDIA announced that LOBPCG will be included in the GPU-accelerated Algebraic Multigrid Accelerator library AmgX.[6] This work extends the results presented by Anzt et al. (2015) by providing a more comprehensive analysis on energy and performance, introducing models for estimating hardware-imposed optimization bounds, and enhancing the LOBPCG solver with preconditioning.

3 Performance bounds for SpMV kernels

The performance of sparse computations, including the performance of standard Krylov iterative methods, is typically bounded by the performance of the SpMV. The performance of the SpMV itself is typically bounded by the memory bandwidth of the system at hand. To demonstrate this, we consider the SpMV performance on a state-of-the-art GPU for a set of sparse matrices from the University of Florida Matrix Collection (UFMC)[7] with characteristics listed in Table 1. Unless the non-zeros of a sparse matrix are equally distributed over the distinct rows, which would imply that all rows contain a very similar number of non-zeros, the compressed sparse row (CSR (Barrett et al., 1994); see Section 5 for details) is the most compact format for storing the matrix. In this general case, the CSR is also the format minimizing the data transfers in an SpMV kernel. Against this background, a lower bound of data transfers in bytes (t_B) needed for a double precision (DP) SpMV product in the form $Ax = y$ can be evaluated as follows:

$$t_B = \underbrace{8n_z}_{\text{Read matrix coefficients}} + \underbrace{4n_z}_{\text{Read column indices}} + \underbrace{4n}_{\text{Read row indices}} + \underbrace{8n}_{\text{Read } x} + \underbrace{8n}_{\text{Write } y} = 12n_z + 20n. \tag{1}$$

The number of floating point operations (Flops) for an SpMV is at most $2n_z$. Thus, taking into account equation (1), we can derive the Flops per byte (Flops/B) ratio for the SpMV. Note that the number is very low and, moreover, we can use it to derive an upper performance bound for the SpMV. In particular, if we disregard the time for the SpMV Flops, the time to transfer the data will give us an upper bound for the performance (P_{max}):

$$P_{max} = \frac{\text{Flops}}{t_B} \cdot \text{bandwidth} \tag{2}$$

where the bandwidth is the achievable data transfer rate from/to the main memory. On a K40 GPU for

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

Acronym	Matrix	#nonzeros (n_z)	Size (n)	n_z/n	n_z^{row}	ELLPACK		SELL-P	
						$n_z^{ELLPACK}$	overhead	n_z^{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 2. Efficiency of the MAGMA `SpMV` in SELL-P format as a percentage of the P_{max} peak from equation (2). The numbers are obtained based on the performances in Figure 1, Table 1, and bandwidth of 180 GB/s. This bandwidth is the maximum achievable on the K40 GPU for a dense matrix–vector product by the cuSPARSE and MAGMA libraries, as well as the bandwidth reported by NVIDIA's `bandwidthTest` benchmark.

Matrix acronym	Percent of peak for MAGMA SELL-P	
	without overhead	with overhead
AUDI	76	93
BMW	80	93
BONE010	77	89
F1	67	82
INLINE	72	84
LDOOR	71	87

example, the achievable bandwidth using NVIDIA's `bandwidthTest` benchmark is about 180 GB/s, and thus, the upper performance bound for the `SpMV`, according to equation (2), can be approximated by disregarding the $20n$ term in t_B as

$$P_{max} \leq \frac{2n_z}{12n_z} \cdot 180 = \frac{180}{6} = 30 \text{ GFlop}/s.$$

For example, taking into account the $20n$ term on the bmw matrix, yields a $P_{max} = 29.3$ GFlop/s. Note that MAGMA achieves 23.6 GFlop/s (see Figure 1), which is 80% of the P_{max} peak. Furthermore, note that this performance is achieved using a SELL-P format that, in this particular case, has 13% overhead (see Table 1). Therefore, if we take into account the overhead, our implementation achieves 93% of the P_{max} peak. The efficiencies achieved as a percentage of the P_{max} peak for the other matrices are given in Table 2.

These results illustrate the bandwidth limitations and the current state-of-the-art for the `SpMV` kernel and sparse computations in general. Furthermore, the comparison in Figure 1 shows that even if the `SpMV` is 100% efficient, the overall performance will still only be about 2.4% of the performance capabilities of the K40 GPU. This is a motivation for the work in this paper to further improve and uncover what is possible through techniques for reducing the communications in sparse computations.

4 LOBPCG

The LOBPCG method (Knyazev, 1998, 2001) 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 (Hetmaniuk and Lehoucq, 2006; Knyazev, 2001). While the LOBPCG convergence characteristics usually benefit from using an application-specific preconditioner (Arbenz and Geus, 2005; Benner and Mach, 2011; Knyazev and Neymeyr, 2001; Kolev and Vassilevski, 2006; Vömel et al., 2008), we refrain from including specialized preconditioners as we are particularly interested in the performance of the top-level method and general cases where preconditioners such as ILU would be applied. 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

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

the Rayleigh–Ritz method computes on the GPU

$$\tilde{A} = [X_i, X_{i-1}, R]^{\mathrm{T}} A [X_i, X_{i-1}, R]$$
$$B = [X_i, X_{i-1}, R]^{\mathrm{T}} [X_i, X_{i-1}, R]$$

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.[8] 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 7).

5 Sparse matrix–vector–block product

To develop an efficient `SpMM` kernel, we use the recently proposed SELL-P format (padded sliced ELLPACK (Anzt et al., 2014)). The performance numbers reported throughout this section are for DP arithmetic.

5.1 Implementation of `SpMM` *for SELL-P*

In addition to the well-known sparse matrix formats such as CSR (Barrett et al., 1994), work on efficient `SpMV` products for GPUs has motivated a number of new formats, and in particular, the one standing out is ELLPACK (Bell and Garland, 2008), 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 non-zero 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 (Kreutzer et al., 2014; Monakov et al., 2010)), the overhead is no longer determined by the matrix row containing the largest number of non-zeros, but is determined by the row with the largest number of non-zero 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 (Bell and Garland, 2008) 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 requires storing some additional zeros, the comparison between the formats in Figure 3 reveals that the blocking strategy may still render significant memory savings compared with the plain ELLPACK (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. This 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 makes a

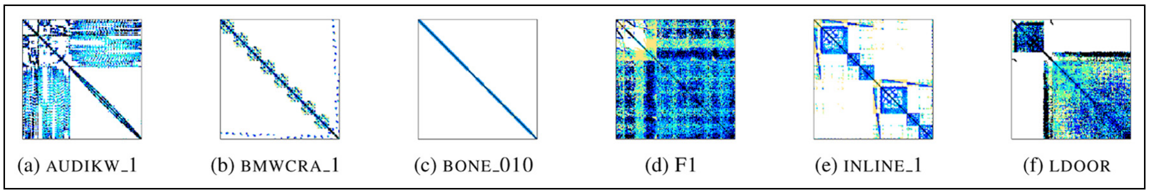

Figure 2. Sparsity plots of selected test matrices.

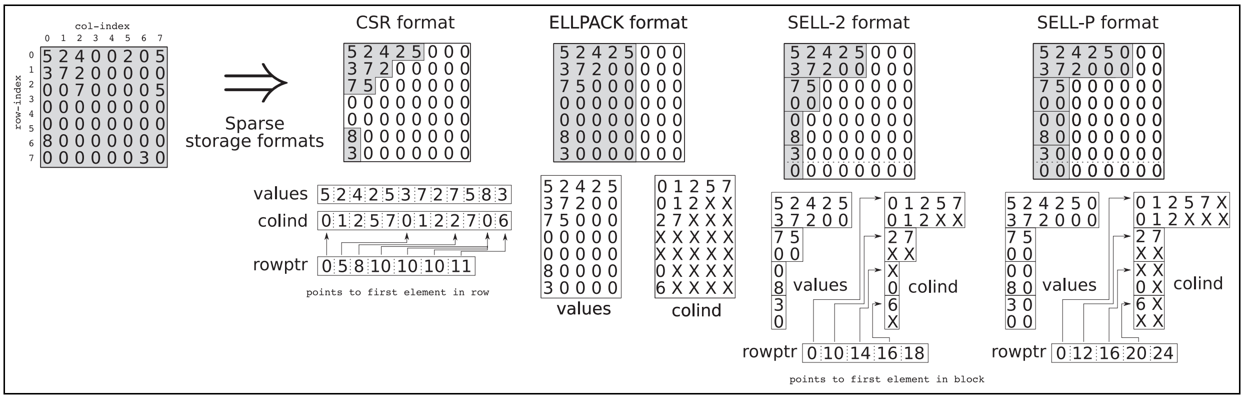

Figure 3. Visualizing the CSR, ELLPACK, SELL-C, and SELL-P formats. The memory demand corresponds to the gray areas. Note that choosing the block size 2 for SELL-C (SELL-2) and SELL-P requires adding a zero row to the original matrix. Furthermore, padding the SELL-P format to a row length divisible by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

preprocessing step transposing the dense matrix from column to row-major format necessary, 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 Anzt et al. (2014). 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 provides incentive 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 straightforward implementation, keeping the set of vectors in texture memory makes an enhanced approach more appealing: In CUDA (version 6.0) every texture read fetches 16 bytes, corresponding to two IEEE DP 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 (DP 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 (NVIDIA, 2009). 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.

5.2 Performance of `SpMM` for SELL-P

For the runtime analysis we use a Kepler K40 GPU which is newer than the K20 GPUs employed in the Piz Daint supercomputer. We benchmarked the SELL-P `SpMM` for a set of test matrices taken from the University of Florida Matrix Collection (UFMC).[9]. 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. The host system has a theoretical peak of 333 GFlop/s in DP and is equipped with 64 GB of main memory that can be accessed at a theoretical bandwidth of up to 51 GB/s. The K40 GPU has a theoretical peak performance of 1682 GFlop/s in DP. On the card, 12 GB of main memory are accessed at a theoretical bandwidth of 288 GB/s. Using bandwidth tests provided by NVIDIA we achieved a bandwidth of 180 GB/s. The implementation of all GPU kernels is realized in CUDA (NVIDIA, 2009) version 6.0,[10] and we also include in performance comparisons with routines taken from NVIDIA's cuSPARSE library. On the CPU, Intel's MKL (Intel Corporation, 2007) is used (version 11.0, update 5).

In Figure 5, we visualize the performance scaling of the previously described `SpMM` kernel for different test matrices 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. Note in particular the very good performance for the cases where the column-count equals a multiple of 16. Using the `SpMM` kernel versus a set of consecutive `SpMV` sparse products (that typically achieve less than 25 GFlop/s

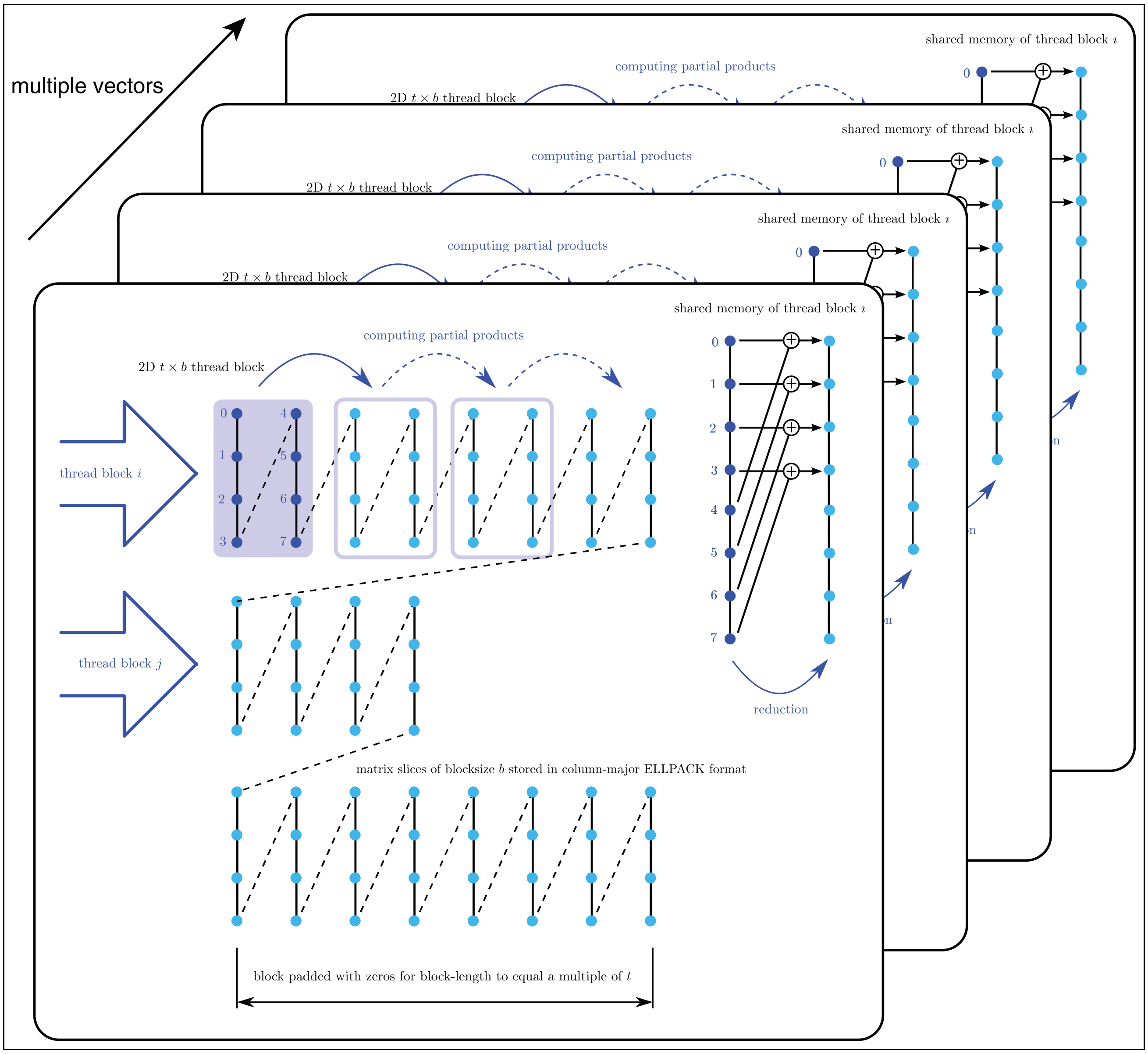
multiple vectors
shared memory of thread block i
computing partial products
2D t × b thread block
thread block i
thread block j
reduction
matrix slices of blocksize b stored in column-major ELLPACK format
block padded with zeros for block-length to equal a multiple of t

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.

on this architecture (Anzt et al., 2014); see Figure 1, right) results in speedup factors of around 5 ×. Similar performance improvement (up to 6.1 ×) is observed on CPUs when replacing consecutive MKL `SpMV` kernels with the MKL `SpMM` routine, see MKL results in Table 3.

While these results are obtained by assuming the performance-beneficial row-major storage of the tall-and-skinny dense matrix, 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 and the cuSPARSE CSRSpMM (Naumov, 2012) with unblocked `SpMV` counterparts, the performance reported in Table 3 includes the time needed to transpose the tall-and-skinny dense matrix composed of the vectors.

The performance results reveal that the SELL-P based `SpMM` kernel outperforms the cuSPARSE counterpart for all tested matrices. The speedup over the fastest SpMV ranges between 3.8 × and 5.4 ×. The CPU results (using Intel's MKL) were obtained by running the CPU in the `numactl -interleave=all` policy, which is well known to improve performance. The reported performance is consistent with

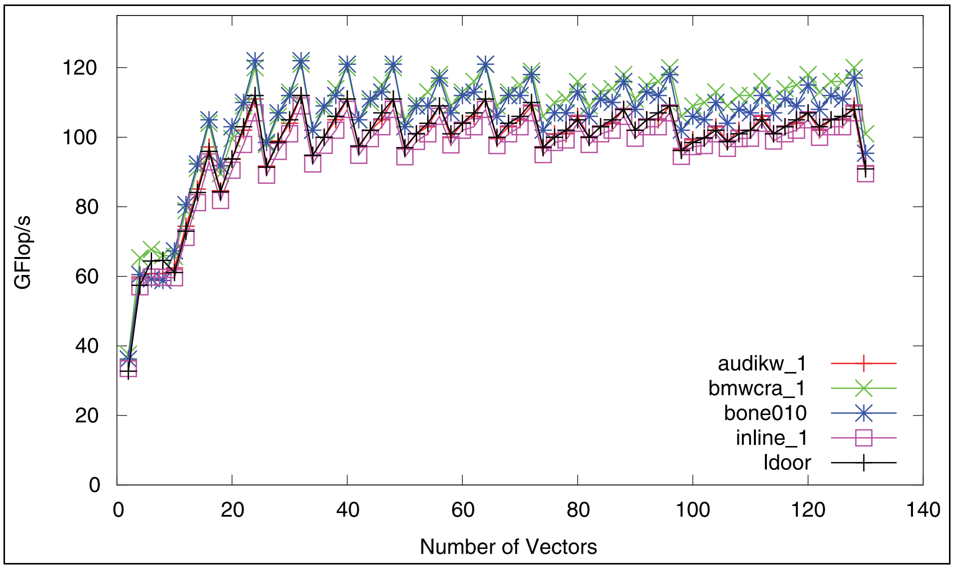

Figure 5. DP performance scaling with respect to the vector count of the SpMM kernel for the matrices listed in Table 1.

Table 3. Asymptotic DP performance [GFlop/s] for a large number of vectors using consecutive SpMV s (mkl_dcsrmv, cuSPARSE CSR, cuSPARSE HYB, MAGMA SELL-P SpMV) or a blocked SpMV kernels (mkl_dcsrmm, cuSPARSE SpMM, MAGMA SpMM). The last three columns is the speedup of the MAGMA SpMM against the best SpMV and the mkl_dcsrmm, respectively. See Table 1 for the matrix characteristics.

	Intel MKL		NVIDIA cuSPARSE			MAGMA		speedup	
Matrix	mkl_dcsrmv	mkl_dcsrmm	CSR	HYB	SpMM	SELL-P	SpMM	*vs.*best SpMV	*vs.* mkl_dcsrmm
AUDI	7.24	22.5	21.9	17.7	104.5	22.1	111.3	5.0	3.1
BMW	6.86	32.2	22.3	24.2	112.5	23.6	122.0	3.8	4.7
BONE010	7.77	30.5	15.5	25.2	108.3	22.3	121.6	4.1	3.9
F1	5.64	20.1	19.3	16.9	99.6	19.6	106.3	5.4	3.6
INLINE	8.10	28.9	20.7	19.1	102.0	21.1	108.8	3.8	3.6
LDOOR	6.78	41.5	14.9	19.3	99.4	20.7	111.2	5.4	6.1

benchmarks provided by Intel (Intel Corporation, 2014). The results reveal a 3 × to 5 × acceleration from CPU to GPU implementation, which was expected from the compute and bandwidth capabilities of the two architectures.

6 Experiment framework

The Piz Daint Supercomputer at the Swiss National Computing Centre in Lugano was listed in November 2014 as the sixth fastest supercomputer according to the TOP500, while its energy efficiency ranked number nine in the Green500. 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 (CSCS, 2014). Piz Daint has 5272 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 5272 GPUs. The peak performance is 7.8 Petaflops (CSCS, 2014). 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 (Fourestey et al., 2014). 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.

7 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. 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

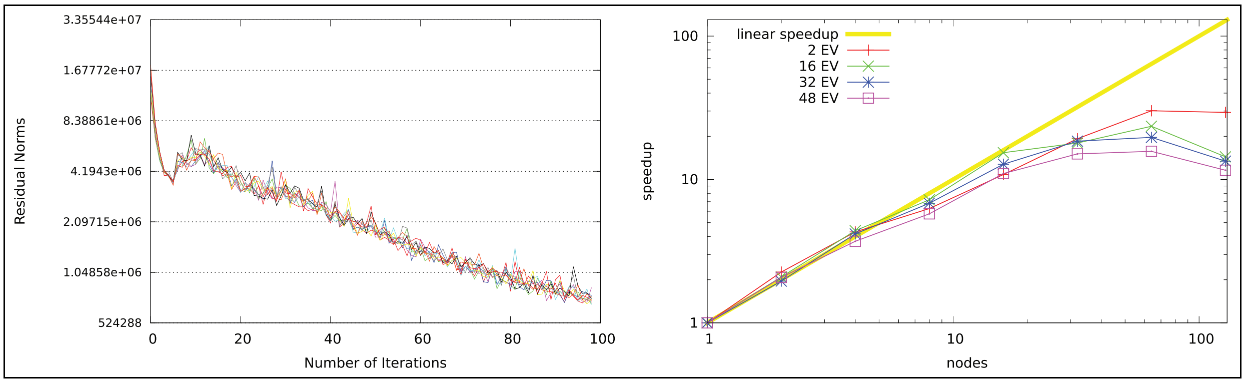

Figure 6. Visualization of the LOBPCG convergence computing 10 eigenvectors for the Audi test matrix (left) and the corresponding scaling of the Hypre code (right).

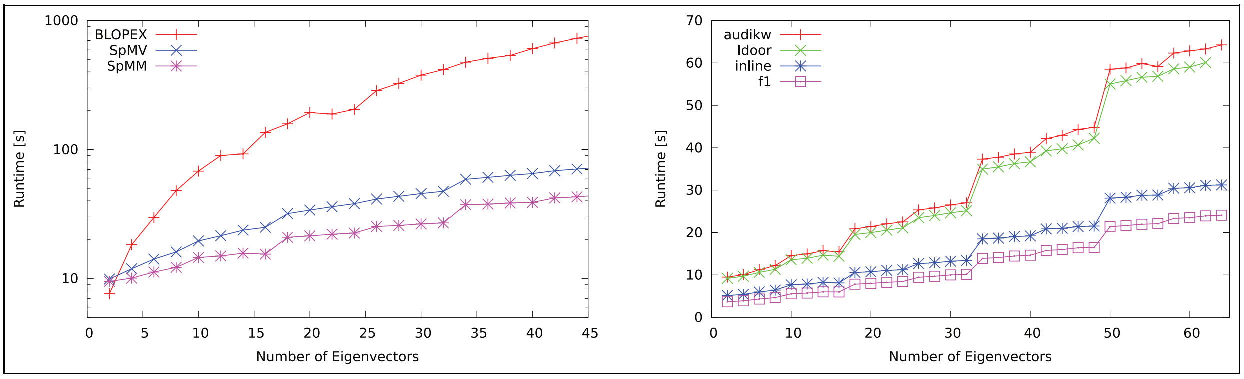

Figure 7. Left: Runtime to complete 100 DP iterations on the Audi problem using either the BLOPEX via the Hypre interface on 4 nodes (32 threads), or the GPU implementation using either a sequence of `SpMV` s or the `SpMM` kernel. Right: Runtime needed to complete 100 iterations in DP using the LOBPCG GPU implementation based on `SpMM`.

to 128 nodes (1024 cores) of the hardware platform listed in Section 6. 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. On the left-hand side of Figure 6 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 right-hand graph of Figure 6 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 moderate node counts.

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

$$2 \cdot n_z \cdot n_v + 36 \cdot n \cdot n_v^2 \tag{3}$$

where n_z 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 on the left-hand graph of Figure 7 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 `SpMV`s 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

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) with respect to performance (left) or energy efficiency (right).

Matrix	Best performance			Best energy efficiency		
	Nodes	Time [s]	Energy [J]	Nodes	Time [s]	Energy [J]
AUDI	64	8.88	73,956	8	24.21	27,611
BMW	8	3.59	3713	2	7.57	2291
BONE010	64	6.06	38,656	8	15.33	18,078
F1	32	8.12	32,179	4	17.20	9587
INLINE	64	6.22	46,744	4	18.08	10,558
LDOOR	64	7.14	56,041	8	14.41	16,976

Matrix	Best performance			Best energy efficiency		
	Nodes	Time [s]	Energy [J]	Nodes	Time [s]	Energy [J]
AUDI	32	34.86	151,800	2	307.20	96,060
BMW	8	14.41	14,564	2	30.20	9215
BONE010	64	23.85	192,492	2	258.93	81,873
F1	32	23.86	101,979	1	206.64	31,659
INLINE	32	24.31	105,512	8	36.73	42,382
LDOOR	32	24.93	107,172	2	233.35	74,638

Table 5. Runtime and energy balance of 100 iterations of the GPU-accelerated LOBPCG in DP. The last column relates the GPU energy to the total energy.

Matrix	EVs	Time [s]	Energy CPU [J]	Energy GPU [J]	GPU versus total energy [%]
AUDI	16	14.69	1075.00	1664.00	60.75
	32	19.47	1377.00	2539.00	64.84
BMW	16	2.54	182.00	232.00	56.04
	32	3.37	241.00	375.00	60.88
BONE010	16	14.58	996.00	1609.00	61.77
	32	18.97	1307.00	2456.00	65.27
F1	16	5.57	390.00	581.00	59.84
	32	7.40	536.00	958.00	64.12
INLINE	16	7.71	602.00	906.00	60.08
	32	10.11	709.00	1248.00	63.77
LDOOR	16	13.87	951.00	1537.00	61.78
	32	18.39	1283.00	2368.00	64.86

consecutive SpMVs with the SpMM, as this kernel also promotes certain column-counts of the tall and skinny dense matrix. The right-hand side of Figure 7 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 × 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 respective improvements of 30% and 20% reflect that only 12.5% and 8.7% of the overall LOBPCG flops are in SpMVs for the 32 and 48 eigenvector problems, respectively (see equation (3) 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

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

Nodes	Speedup						Greenup					
	AUDI	BMW	BONE010	F1	INLINE	LDOOR	AUDI	BMW	BONE010	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
Nodes	Speedup						Greenup					
	AUDI	BMW	BONE010	F1	INLINE	LDOOR	AUDI	BMW	BONE010	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

also promotes certain column-counts of the tall and skinny dense matrix, as shown on the right of Figure 7.

On the right-hand side of Figure 6 we identified almost linear scaling of the Hypre implementation. 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 computed. 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 nonlinear 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. The results in Table 5 indicate that handling a larger set of eigenvalues increases the GPU energy need in relation to the total energy budget. Depending on the system and the number of eigenvectors, up to 65% of the energy is consumed by the GPU. 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 115 W (1.44 GFlop/W), whereas the K20X GPU consumes 225 W when providing 1311 GFlop/s (5.83 GFlop/W) (Fourestey et al., 2014). Hence, assuming full load, adding the accelerator theoretically increases the power need per node by a factor of around 1.67. However, the GPU-accelerated LOBPCG also uses the CPU for solving the Rayleigh–Ritz problem independent of the system matrix. 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 difference 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 top left 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 top right 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

Table 7. Runtime and energy balance of 100 iterations of the Jacobi-preconditioned LOBPCG in DP.

Matrix	EVs	Time [s]	Energy CPU [J]	Energy GPU [J]	GPU versus total energy [%]
AUDI	16	22.69	1735.00	3098.00	64.10
	32	53.10	3694.00	7637.00	67.40
BMW	16	4.54	363.00	463.00	56.05
	32	11.55	849.00	1247.00	59.49
BONE010	16	22.38	1565.00	2906.00	65.00
	32	52.98	3526.00	7439.00	67.84
F1	16	9.04	690.00	1155.00	62.60
	32	21.72	1522.00	2888.00	65.49
INLINE	16	12.14	1053.00	1658.00	61.16
	32	28.04	2092.00	3861.00	64.86
LDOOR	16	21.72	1480.00	2798.00	65.40
	32	51.49	3461.00	7200.00	67.54

Table 8. Runtime and energy balance of 100 iterations of the ILU-preconditioned LOBPCG in DP.

Matrix	EVs	Time [s]	Energy CPU [J]	Energy GPU [J]	GPU versus total energy [%]
AUDI	16	71.97	5475.00	8276.00	60.18
	32	121.84	8730.00	15,425.00	63.86
BMW	16	10.47	884.00	1123.00	55.95
	32	21.12	1632.00	2331.00	58.82
BONE010	16	39.94	2863.00	5020.00	63.68
	32	84.02	5709.00	11,306.00	66.45
F1	16	26.89	2142.00	3201.00	59.91
	32	49.34	3597.00	6151.00	63.10
INLINE	16	35.09	2819.00	4172.00	59.68
	32	63.25	4642.00	7990.00	63.25
LDOOR	16	51.33	3667.00	5770.00	61.14
	32	96.83	6696.00	12,241.00	64.64

when computing 16 eigenvectors. 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 Table 3, the expectation for 16 vectors is that an optimized CPU code (blocking the `SpMV`s) is 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 eigenvectors (e.g. 32, shown in Table 6, bottom) reduces the fraction of `SpMV` flops to the total flops (see equation (3)), 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 $\times$, indicates that there are other missed optimization opportunities in the CPU implementation. For large eigenvector counts, the majority of flops are in GEMMs used for assembling the matrix representations for the local Rayleigh–Ritz minimizations and the orthogonalizations. In the GPU-accelerated LOBPCG implementation, these routines are highly optimized. In particular the GEMMs are optimized for the tall-and-skinny shape of the matrices involved (Nath et al., 2010). More specifically, the matrix multiplication $A^{\mathrm{T}} \cdot B$ is split into a set of smaller GEMMs that are tuned for the particular sizes. A batched routine is used for simultaneous execution of the small GEMMs (Yamazaki et al., 2015), and the sum of the partial products is formed in a post-processing step.

We now enhance the LOBPCG with a preconditioner. For the developed GPU-accelerated version we analyze in Tables 7 and 8 the runtime and energy when using the Jacobi and ILU preconditioner available in the MAGMA software package (Innovative Computing Laboratory, UTK, 2014a). As expected, enhancing the LOBPCG with a preconditioner increases runtime and energy consumption of every iteration. In a scientific application, the preconditioner's efficiency is determined by the tradeoff between increased resource usage and improved convergence behavior, which is outside the focus of this study. By comparing with the non-preconditioned LOBPCG in

Table 9. Speedup and greenup of the MAGMA Jacobi-LOBPCG *vs.* the CPU Jacobi-LOBPCG when computing 16 (top) or 32 (bottom) eigenvectors.

Nodes	Speedup						Greenup					
	AUDI	BMW	BONE010	FI	INLINE	LDOOR	AUDI	BMW	BONE010	FI	INLINE	LDOOR
1	6.53	3.45	7.10	7.87	6.10	16.99	4.87	3.01	5.61	6.04	4.35	13.25
2	4.04	1.50	3.28	3.75	3.10	8.06	6.00	2.47	5.31	5.77	4.49	12.74
4	1.57	0.91	1.64	1.07	1.15	4.05	4.64	2.70	5.11	3.17	3.16	11.76
8	0.69	1.11	0.70	0.89	0.78	1.84	3.90	6.15	4.02	4.93	4.01	7.92
16	0.50	1.25	0.35	0.59	0.47	0.97	4.63	7.47	3.65	6.25	4.65	8.65
Nodes	Speedup						Greenup					
	AUDI	BMW	BONE010	FI	INLINE	LDOOR	AUDI	BMW	BONE010	FI	INLINE	LDOOR
1	12.02	6.12	10.12	25.65	8.78	20.25	9.01	5.35	7.82	19.57	6.68	15.29
2	6.82	2.37	4.73	5.30	5.19	9.85	10.22	4.10	7.53	8.37	7.91	15.13
4	3.20	1.28	2.43	1.92	2.47	5.11	9.33	3.95	7.40	5.79	7.19	14.31
8	2.48	2.90	1.48	1.26	1.46	3.25	13.67	11.98	8.63	7.14	7.75	17.33
16	0.85	1.29	0.56	0.88	0.75	1.30	8.59	14.95	6.05	9.51	7.71	10.52

Table 5, we notice that the GPU fraction in the overall energy balance increases for the Jacobi preconditioner and decreases for the ILU preconditioner. This indicates that applying the Jacobi preconditioner is a compute-intensive operation increasing the pressure on the GPU, while the less compute-intense forward–backward triangular solves in the ILU-preconditioned LOBPCG cause a low GPU utilization and emphasize the CPU contributions to the energy balance.

In Table 9 we report the speedup and greenup of the Jacobi-preconditioned MAGMA LOBPCG over the CPU counterpart. Although the speedups are typically smaller than for the unpreconditioned LOBPCG, the BLOPEX LOBPCG used via the Hypre framework does for no configuration achieve the energy efficiency level of the GPU-accelerated implementation.

As a bottomline, 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.

8 Summary and outlook

In this study we 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 provided a comprehensive description of the GPU-kernel used to generate the block-Krylov subspace via an `SpMM` product. As this building block also serves as the backbone for other block-Krylov methods, we included a runtime analysis that reveals significant speedups over a set of consecutive SpMVs, and an equivalent routine provided in Intel's MKL and NVIDIA's cuSPARSE library. We compared runtime and energy efficiency of the GPU-accelerated LOBPCG with the BLOPEX LOBPCG implementation used via the Hypre interface. Running the CPU code on one node we reported more than an order of magnitude lower resource utilization for the GPU-accelerated version. We showed that increasing the hardware resources for the BLOPEX code succeeds in improving the runtime performance, but results in ever higher energy usage. This indicates that with tuning GPU code it is possible to transfer the resource efficiency of GPU supercomputers into improved energy efficiency of the scientific applications. Future work includes simplifying the process of evaluating a system's resource efficiency when running scientific applications by creating a set of basic benchmark kernels, and extrapolating to a more complex algorithm.

Acknowledgement

The authors would like to thank the Swiss National Computing Centre for granting access to their system, and support in deploying the energy measurements.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This material is based upon work supported by the National Science Foundation (Grant number ACI-1339822), Department of Energy (Grant number DE-SC0010042), and NVIDIA. The work was also funded in part by the Russian Scientific Foundation (agreement N14-11-00190).

Notes

1. See http://www.green500.org/
2. See http://www.top.org/
3. See https://code.google.com/p/blopex/
4. The authors would like to thank the CSCS for access to Piz Daint and the support in running the experiments.
5. See http://docs.nvidia.com/cuda/cusparse/
6. See https://developer.nvidia.com/amgx
7. See http://www.cise.ufl.edu/research/sparse/matrices/
8. See http://www.mathworks.com/matlabcentral/fileexchange/48-lobpcg-m
9. See http://www.cise.ufl.edu/research/sparse/matrices/
10. See https://developer.nvidia.com/cuda-toolkit-60

References

Agullo E, Augonnet C, Dongarra J, et al. (2011) QR factorization on a multicore node enhanced with multiple GPU accelerators. In: *Proceedings of IPDPS*. IEEE, pp. 932–943.

Aliaga JI, Anzt H and Castillo M (2015) Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors. *Concurrency and Computation: Practice and Experience* 27(4): 885–904.

Anderson E, Bai Z, Bischof C, et al. (1999) *LAPACK Users' Guide*, 3rd edn. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Anderson M, Ballard G, Demmel J and Keutzer K (2010) Communication-avoiding QR decomposition for GPUs. Technical Report UCB/EECS-2010-131, EECS Department, University of California, Berkeley.

Anzt H, Tomov S and Dongarra J (2014) Implementing a sparse matrix vector product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. Technical Report ut-eecs-14-727, University of Tennessee.

Anzt H, Tomov S and Dongarra J (2015) Energy efficiency and performance frontiers for sparse computations on GPU supercomputers. In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '15)*. New York: ACM Press, pp. 1–10.

Arbenz P and Geus R (2005) Multilevel preconditioned iterative eigensolvers for Maxwell eigenvalue problems. *Applied Numerical Mathematics* 54(2): 107–121.

Baker CG, Hetmaniuk UL, Lehoucq RB and Thornquist HK (2009) Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software* 36(3): 13:1–13:23.

Barrett R, Berry M, Chan TF, et al. (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd edn. Philadelphia, PA: SIAM.

Bell N and Garland M (2008) Efficient sparse matrix–vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004.

Benner P and Mach T (2011) Locally optimal block preconditioned conjugate gradient method for hierarchical matrices. *PAMM* 11(1): 741–742.

Castro A, Appel H, Oliveira M, et al. (2006) octopus: a tool for the application of time-dependent density functional theory. *physica status solidi (b)* 243(11): 2465–2488.

CSCS (2014) Piz Daint Computing Resources. URL http://user.cscs.ch/computing_systems/piz_daint/index.html.

Charles J, Sawyer W, Dolz MF and Catalán S (2015) Evaluating the performance and energy efficiency of the COSMO-ART model system. *Computer Science - Research and Development* 30(2): 177–186.

Demmel J, Grigori L, Hoemmen M and Langou J (2008) Communication-avoiding parallel and sequential qr factorizations. *CoRR* abs/0806.2159.

Dongarra J, Beckman P, Moore T, et al. (2011) The international ExaScale software project roadmap. *The International Journal of High Performance Computing Applications* 25(1): 3–60.

Dongarra J and Heroux MA (2013) Toward a new metric for ranking high performance computing systems. SANDIA REPORT SAND2013-4744.

Donofrio D, Oliker L, Shalf J, et al. (2009) Energy-efficient computing for extreme-scale science. *Computer* 42(11): 62–71.

Fourestey G, Cumming B, Gilly L and Schulthess TC (2014) First experiences with validating and using the Cray power management database tool. In: *Proceedings of the 2014 Cray User Group (CUG) user meeting*, Lugano, Switzerland.

Golub GH and Van Loan CF (1996) *Matrix Computations*, 3rd edn. Baltimore, MD: Johns Hopkins University Press.

Gonze X, Beuken JM, Caracas R, et al. (2002) First-principles computation of material properties: the ABINIT software project. *Computational Materials Science* 25(3): 478–492.

Heroux M, Bartlett R, Hoekstra VHR, et al. (2003) An overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories.

Hetmaniuk U and Lehoucq R (2006) Basis selection in LOBPCG. *Journal of Computational Physics* 218(1): 324–332.

Hoemmen M (2010) *Communication-avoiding Krylov subspace methods*. PhD Thesis, EECS Department, UC Berkeley.

Innovative Computing Laboratory, UTK (2014a) Software distribution of MAGMA version 1.5. http://icl.cs.utk.edu/magma/.

Innovative Computing Laboratory, UTK (2014b) Software distribution of MAGMA version 1.6.1. http://icl.cs.utk.edu/magma/.

Intel Corporation (2007) Intel® Math Kernel Library for Linux* OS. Document Number: 314774-005US. Intel Corporation.

Intel Corporation (2014) Intel® Math Kernel Library. Sparse BLAS and Sparse Solver Performance Charts: DCSRGEMV and DCSRMM. https://software.intel.com/en-us/intel-mkl. Intel Corporation.

Jiménez V, Gioiosa R, Kursun E, et al. (2010) Trends and techniques for energy efficient architectures. In: *2010 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC)*, pp. 276–279. DOI:10.1109/VLSISOC.2010.5642673.

Jones E, Oliphant T, Peterson P, et al. (2001–) SciPy: Open source scientific tools for Python. Available at: http://www.scipy.org/.

Kestor G, Gioiosa R, Kerbyson D and Hoisie A (2013) Quantifying the energy cost of data movement in scientific applications. In: *2013 IEEE International Symposium on*

Workload Characterization (IISWC), pp. 56–65. DOI:10.1109/IISWC.2013.6704670.

Knote Cea (2011) Towards an online-coupled chemistry–climate model: evaluation of trace gases and aerosols in COSMO-ART. *Geoscientific Model Development* 4(4): 1077–1102.

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

Knyazev AV (1998) Preconditioned eigensolvers - an oxymoron? *Electronic Transactions on Numerical Analysis* 7: 104–123.

Knyazev AV (2001) Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing* 23: 517–541.

Knyazev AV, Argentati ME, Lashuk I and Ovtchinnikov EE (2007) Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc. *SIAM Journal on Scientific Computing* 29(5): 2224–2239.

Kolev TV and Vassilevski PS (2006) Parallel eigensolver for H(curl) problems using H1-auxiliary space AMG preconditioning. Technical Report UCRL-TR-226197, Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

Kreutzer M, Hager G, Wellein G, Fehske H and Bishop AR (2014) A unified sparse matrix data format for efficient general sparse matrix–vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36(5): C401–C423.

Krueger J, Donofrio D, Shalf J, et al. (2011) Hardware/software co-design for energy-efficient seismic modeling. In: *Proceedings of SC'11*. New York: ACM Press, pp. 73:1–73:12.

Monakov A, Lokhmotov A and Avetisyan A (2010) Automatically tuning sparse matrix–vector multiplication for GPU architectures. In: *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'10)*. Berlin: Springer-Verlag, pp. 111–125.

Nath R, Tomov S and Dongarra J (2010) An improved Magma Gemm For Fermi graphics processing units. *The International Journal of High Performance Computing Applications* 24(4): 511–515.

Naumov M (2012) Preconditioned block-iterative methods on GPUs. *PAMM* 12(1): 11–14.

NVIDIA (2009) NVIDIA CUDA *Compute Unified Device Architecture Programming Guide*, 2.3.1 edition. NVIDIA Corporation.

Padoin E, Pilla L, Boito F, Kassick R, Velho P and Navaux P (2013) Evaluating application performance and energy consumption on hybrid CPU + GPU architecture. *Cluster Computing* 16(3): 511–525.

Stathopoulos A and Wu K (2002) A block orthogonalization procedure with constant synchronization requirements. *SIAM Journal on Scientific Computing* 23: 2165–2182.

Tomov S, Langou J, Dongarra J, Canning A and Wang LW (2006) Conjugate-gradient eigenvalue solvers in computing electronic properties of nanostructure architectures. *International Journal of Computational Science and Engineering* 2(3/4): 205–212.

Vömel C, Tomov SZ, Marques OA, Canning A, Wang LW and Dongarra JJ (2008) State-of-the-art eigensolvers for electronic structure calculations of large scale nano-systems. *Journal of Computational Physics* 227(15): 7113–7124.

Wittmann M, Hager G, Zeiser T and Wellein G (2013) An analysis of energy-optimized lattice-Boltzmann CFD simulations from the chip to the highly parallel level. *CoRR*. Available at: https://pdfs.semanticscholar.org/7c0c/02245f6704a00800a43ecf7d87f0e977ff7e.pdf

Yamada S, Imamura T, Kano T and Machida M (2006) High-performance computing for exact numerical approaches to quantum many-body problems on the Earth simulator. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. New York: ACM Press.

Yamada S, Imamura T and Machida M (2005) 16.447 TFlops and 159-billion-dimensional exact-diagonalization for trapped Fermion–Hubbard model on the Earth simulator. In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. Washington, DC: IEEE Computer Society, p. 44.

Yamazaki I, Anzt H, Tomov S, Hoemmen M and Dongarra J (2014) Improving the performance of CA-GMRES on multicores with multiple GPUs. In: *Proceedings of IPDPS'14*. Washington, DC: IEEE Computer Society, pp. 382–391.

Yamazaki I, Tomov S, Dong T and Dongarra J (2015) Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs. In: Daydé M, et al. (eds), *VECPAR 2014* (*Lecture Notes in Computer Science*, vol. 8969). New York: Springer, pp. 17–30.

Author biographies

Hartwig Anzt is a research scientist in Jack Dongarra's Innovative Computing Lab (ICL) at the University of Tennessee. He received his PhD in mathematics from the Karlsruhe Institute of Technology (KIT) in 2012. His research interests include simulation algorithms, sparse linear algebra (in particular, iterative methods and preconditioning), hardware-optimized numerics for GPU-accelerated platforms, and power-aware computing.

Stanimire Tomov is a research director in the Innovative Computing Laboratory (ICL) at the University of Tennessee. His research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). He has been involved in the development of numerical algorithms and software tools in a variety of fields ranging from scientific visualization and data mining to accurate and efficient numerical solution of PDEs. Currently, his work is concentrated on the development of numerical linear algebra libraries for emerging architectures for HPC, such as heterogeneous multicore processors, GPUs, and Intel Xeon Phi processors.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE IPDPS Charles Babbage Award; and in 2013 he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.