Accelerating the LOBPCG method on GPUs using a blocked Sparse Matrix Vector Product

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

***Abstract*—**

This paper presents a heterogeneous CPU-GPU algorithm design and optimized implementation for an entire sparse iterative eigensolver – the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) – starting from low-level GPU data structures and kernels to the higher-level algorithmic choices and overall heterogeneous design. Most notably, the eigensolver leverages the high-performance of a new GPU kernel developed for the simultaneous multiplication of a sparse matrix and a set of vectors (SpMM). This is a building block that serves as a backbone for not only block-Krylov, but also for other methods relying on blocking for acceleration in general. The heterogeneous LOBPCG developed here reveals the potential of this type of eigensolver by highly optimizing all of its components, and can be viewed as a benchmark for other SpMM-dependent applications. Compared to non-blocked algorithms, we show that the performance speedup factor of SpMM vs. SpMV-based algorithms is up to six on GPUs like NVIDIA's K40. In particular, a typical SpMV performance range in double precision is 20 to 25 GFlop/s, while the SpMM is in the range of 100 to 120 GFlop/s. Compared to highly-optimized CPU implementations, e.g., the SpMM from MKL on two eight-core Intel Xeon E5-2690s, our kernel is 3 to $5\times$. faster on a K40 GPU. For comparison to other computational loads, the same GPU to CPU performance acceleration is observed for the SpMV product, as well as dense linear algebra, e.g., matrix-matrix multiplication and factorizations like LU, QR, and Cholesky. Thus, the modeled GPU (vs. CPU) acceleration for the entire solver is also 3 to $5\times$. In practice though, currently available CPU implementations are much slower due to missed optimization opportunities, as we show.

I. Introduction

The main challenges often associated with numerical linear algebra are the fast and efficient solution of large, sparse linear systems, and the appertaining eigenvalue problems. While linear solvers often serve as a backbone of simulation algorithms based on the discretization of partial differential equations, eigensolvers play a central role, e.g., in quantum mechanics, where eigenstates and molecular orbitals are defined by eigenvectors, or principle component analysis. With increasing system size and sparsity, dense linear algebra routines, usually based on direct solvers like LU factorization, or, in the case of an eigenvalue problem, Hessenberg decomposition [1], become less suitable as the memory demand and computational cost may exceed the available resources. Iterative methods providing solution approximations often become the method of choice. However, as their performance is, at least in case of sparse linear systems, usually memory bound, leveraging the computing power of today's supercomputers, often accelerated by coprocessors like graphics processing units (GPUs), becomes challenging.

While there exist numerous efforts to adapt iterative linear solvers like Krylov subspace methods to coprocessor technology, sparse eigensolvers have so far remained outside the main focus. A possible explanation is that many of those combine sparse and dense linear algebra routines, which makes porting them to accelerators more difficult. Aside from the power method, algorithms based on the Krylov subspace idea are among the most commonly used general eigensolvers [1]. When targeting symmetric positive definite eigenvalue problems, the recently developed Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG, see [2]) belongs to the most efficient algorithms. LOBPCG is based on maximizing the Rayleigh Quotient, while taking the gradient as the search direction in every iteration step. Iterating several approximate eigenvectors, simultaneously, in a block in a similar locally optimal fashion, results in the full block version of the LOBPCG. 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 [3], [4]. One of the performance-crucial key elements is a kernel generating the Krylov search directions via computing the product of the sparse system matrix and a set of vectors. With the sparse matrix vector product performance traditionally limited by memory bandwidth, LOBPCG, depending on this routine, has for a long time been considered unsuitable for GPU acceleration.

In this paper we present an LOBPCG implementation for graphics processing units able to efficiently leverage the accelerator's computing power. For this purpose, we employ a sophisticated sparse matrix data layout, and develop a kernel specifically designed to efficiently compute the product of a sparse and a tall-and-skinny dense matrix composed of the block of eigenvector approximations. As this kernel also is an integral part of other block-Krylov solvers, the significance of its performance carries beyond the example integration into LOBPCG we present in this paper. We benchmark the routine against a similar implementation provided in Intel's MKL [5] and NVIDIA's cuSPARSE [6] library, and analyze the improvement it renders to the performance of the LOBPCG GPU implementation. Finally, we also compare it to the state-of-the-art multi-threaded CPU implementations of LOBPCG based on the BLOPEX [7] code for which the software libraries *PETSc* and *hypre* provide an interface [8]. For matrices taken

from the University of Florida Matrix Collection, we achieve significant acceleration when computing a set of the respective eigenstates.

The rest of the paper is structured as follows. We first revise the LOBPCG algorithm that serves not only as a motivation to develop a building block generating the product of a sparse matrix and multiple vectors, but also as a framework revealing the performance improvements. After listing some key characteristics of the hardware platform and the test matrices we target in Section IV, we provide a detailed description of the sparse matrix storage format and the processing idea of the kernel in Section V. We investigate the performance depending on the number of vectors and compare against both the MKL and cuSPARSE equivalent, and implementations based on consecutive sparse matrix vector products. In Section VI we compare the runtime of the LOBPCG GPU implementation using either consecutive matrix vector products or the developed blocked variant. We also include a brief performance comparison against the state-of-the-art CPU implementation available in the BLOPEX software package. We conclude in Section VII by summarizing the findings and listing algorithms that can potentially be accelerated in a similar fashion.

II. Related Work

Blocked sparse matrix vector product: As there exists significant need for blocked sparse matrix vector products, NVIDIA's cuSPARSE library provides this routine for the CSR format [6]. 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 row-major matrix as input, that can be used in combination with a preprocessing step transposing the matrix to achieve significantly higher performance [9]. The blocked sparse matrix vector product we propose in this paper not only outperforms the cuSPARSE implementations for our test cases, but the detailed description also allows porting it to other architectures.

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 [10] and orthogonalizations based on LAPACK (xGEQRF + xUNGQR) [11] and correspondingly MAGMA for GPUs [12], [13], recent work includes communication-avoiding QR [14], also developed for GPUs [15], [16]. For tall and skinny matrices these orthogonalizations are in general memory bound. Higher performance, using Level 3 BLAS operations, 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). These were developed for GPUs in MAGMA, including a mixed-precision Cholesky QR that removes the square by selectively using higher than the working precision arithmetic [17] (also applied to a CA-GMRES for GPUs).

For the LOPBPCG method, the most time consuming operation after the SpMM kernel is the orthogonalization. There are two sets of orthogonalizations of m vectors per iteration (see Section III).

LOBPCG implementations: The BLOPEX software package maintained by Andrew Knyazev may be considered as state-of-the-art for CPU implementations of LOBPCG, as the popular software libraries PETSc and hypre provide an interface [8]. Also Scipy [18], octopus [19] and Anasazi [20] part of the Trilinos library [21] feature LOBPCG implementations. The first implementation of LOBPCG able to utilize a GPU's computing power by has been available since 2011 in the ABINIT material science software package [22]. The implementation, realized in fortran90, benefits from utilizing the generic linear algebra routines available in the CUDA [23] and MAGMA [12], [13] GPU libraries. More recently, NVIDIA anounced that LOBPCG will be included in the GPU-accelerated Algebraic Multigrid Accelerator AmgX [1].

III. LOBPCG

LOBPCG stands for Locally Optimal Block Preconditioned Conjugate Gradient method [2], [24]. It is designed to find 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:

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

which results in finding the smallest eigenstates of the original problem. In the LOBPCG method 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 RayleighRitz method. This is summarized by the pseudo-code on Figure 1.

```
1 do i=1, niter
2     R = P(AX_i − λX_i)
3     check convergence criteria
4     [X_i, λ] = Rayleigh-Ritz on span{X_i, X_{i−1}, R}
5 end do
```

Fig. 1: LOBPCG algorithm

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 small, and therefore special care and modifications must be taken (see [2], [25]). While the LOBPCG convergence characteristics usually benefit from using an an application-specific preconditioner [26], [27], [28], [29], [30], we refrain from including preconditioners as we are particularly interested in the performance of the top-level method. Our implementation is hybrid, using both the GPUs and CPUs available. In particular, all data resides on the GPU

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

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 done on the GPU. The small and not easy to parallelize Rayleigh-Ritz eigenproblem is done on the CPU using vendor-optimized LAPACK. More specifically, to find

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

the Rayleigh-Ritz method first accumulates the matrices 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 [2]. 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 VI). More stable versions, including Cholesky/SVD QR iterations and the mixed-precision Cholesky QR [17], as well as LAPACK/MAGMA based, CGS, and MGS for GPUs are also an option that we provide.

IV. Experiment Framework

The hardware we use in this paper is a two socket Intel Xeon E5-2670 (Sandy Bridge) platform accelerated by an NVIDIA Tesla K40c GPU with a theoretical peak performance of 1,682GFLOP/s. The host system has a theoretical peak of 333GFLOP/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 [23], version 5.5 [31], while we also include in the performance comparisons routines taken from NVIDIA's cuSPARSE [6] library. On the CPU, Intel's MKL [5] is used in version 11.0, update 5. Note that the CPU-based implementations use the "numactl –interleave=all" option when beneficial.

The matrix problems we target are taken from the University of Florida Matrix Collection (UFMC)[3]. With some key characteristics collected in Table I, and sparsity plots for selected matrices shown in Figure 2, we tried to cover a large variety of systems common in scientific computing.

V. Sparse Matrix-Vector-Block Product

A key building block for the LOBPCG algorithm and other block-Krylov solvers is a routine generating the Krylov search directions by computing the product of a sparse matrix and a set of vectors. This routine can obviously be implemented as a set of consecutive sparse matrix vector products; however, the interpretation as a product of a sparse matrix and a tall-and-skinny dense matrix composed of the distinct vectors may promote a different approach (sparse matrix dense matrix product, `SpMM`). In particular, already cached data of the sparse matrix may be reused when processing multiple vectors simultaneously. This would render performance improvement to the memory-bound kernel. In the GPU implementation of LOBPCG, we realize this routine by handling the sparse matrix using the recently proposed SELL-P format (padded sliced ELLPACK format [32]). In the following we first describe the SELL-P format, provide details on how we implement the `SpMM` kernel, and then analyze its performance by comparing against the CSRSpMM taken from NVIDIA's CUSPARSE library [6].

matrix	nonzeros (nnz)	Size (n)	nnz/n
AUDIKW_1	77,651,847	943,645	82.28
BMW3_2	11,288,630	227,362	49.65
BMWCRA_1	10,641,602	148,770	71.53
BONE_010	47,851,783	986,703	48.50
BONE_S10	40,878,708	914,898	44.68
CANT	4,007,383	62,451	64.17
CRANKSEG_2	14,148,858	63,838	221.63
F1	26,837,113	343,791	78.06
FAULT_639	27,245,944	638,802	42.65
HOOK_1498	59,374,451	1,498,023	39.64
INLINE_1	36,816,170	503,712	73.09
LDOOR	42,493,817	952,203	44.63
PWTK	11,524,432	217,918	52.88
SHIPSEC1	3,568,176	140,874	25.33
STOC_1465	21,005,389	1,465,137	14.34
XENON_2	3,866,688	157,464	24.56

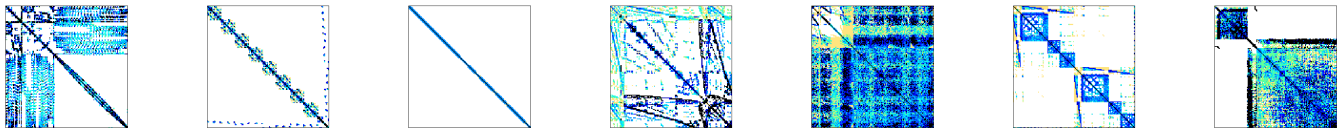
TABLE I: Description and properties of the test matrices.

Implementation of `SpMM` for SELL-P

While for dense matrices it is usually reasonable to store all matrix entries in consecutive order, sparse matrices are characterized by a large number of zero elements, and storing those is not only unnecessary, but would also incur significant storage overhead. Different storage layouts exist that aim to reducing the memory footprint of the sparse matrix by storing only a fraction of the elements explicitly, and anticipating all other elements to be zero, see [33], [34], [35]. In the CSR format [33], this idea is taken to extremes, as only nonzero entries of the matrix are stored. In addition to the array **values** containing the nonzero elements, two integer arrays **colind** and **rowptr** are used to locate the elements in the matrix, see Figure 3. While this storage format is suitable when computing a sparse matrix vector product on processors with a deep cache-hierarchy, as it reduces the memory requirements to a minimum, it fails to allow for high parallelism and coalesced memory access when computing on streaming-processors like GPUs. On those, the ELLPACK-format, padding the different rows with zeros for a uniform row-length, coalesced memory access, and instruction parallelism may, depending on the matrix characteristics, outperform the CSR format [36]. However, the ELLPACK format incurs a storage overhead for the general case, which is determined by the maximum number of nonzero elements aggregated in one row and the average number of nonzeros per row (see Table II). Depending on the associated

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

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



(a) AUDIKW_1 (b) BMWCRA_1 (c) BONE_010 (d) CRANKSEG_2 (e) F1 (f) INLINE_1 (g) LDOOR

Fig. 2: Sparsity plots of selected test matrices.

memory and computational overheads, using ELLPACK may result in poor performance, despite that coalesced memory access is highly favourable for streaming processors.

A workaround to reduce memory and computational overhead is to split the original matrix into row blocks before converting these into the ELLPACK format. In the resulting sliced ELLPACK format (SELL or SELL-C where C denotes the size of the row blocks [37], [38]), the overhead is no longer determined by the matrix row containing the largest number of nonzeros, but by the row with the largest number of nonzero elements in the respective block. While sliced SELL-C reduces the overhead very efficiently (i.e., choosing C=1 results in the storage-optimal CSR format), assigning multiple threads to each row requires padding the rows with zeros, such that each block has a rowlength 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 [36] with the rowlength of each block being padded a multiple of the number of threads assigned to each row when computing a matrix vector or matrix multi-vector 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 compared to ELLPACK (also see Table II), which directly translate into reduced computational cost for the `SpMV` kernel. For the design of the `SpMM` routine it is not sufficient to reduce the computational overhead, as performance also depends on the memory bandwidth. Therefore, it is essential to optimize the memory access pattern, which requires the accessed data to be aligned in memory whenever possible [23]. For consecutive memory access, and with the motivation of processing multiple vectors simultaneously, we implement the `SpMM` assuming the tall-and-skinny dense matrix composed of the vectors being stored in row-major order. Although this requires a preprocessing step transposing the dense matrix prior to the `SpMM` call, the more appealing aligned memory access to the vector values may compensate for the extra work.

The `SpMM` kernel then arises as a natural extension of the `SpMV` routine for the SELL-P format proposed in [32]. 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 (respectively the tall-and-skinny dense matrix), in texture memory, makes an enhanced approach more appealing. The motivation is that in CUDA (version 5.5) 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 [23]. 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.

Performance of `SpMM` for SELL-P

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 `SpMM` kernel instead of a set of consecutive sparse matrix vector products, that typically achieve less than 25 GFLOP/s on this architecture [32], results in speedup factors of up to 5.4 on GPUs, see Table III. 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 IV.

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 against the cuSPARSE CSRSpMM [6], 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 library also includes a highly tuned version assuming, like the MAGMA implementation, row-major storage [9]. Combining this with a preprocessing

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%
CRANK	CRANKSEG_2	14,148,858	63,838	221.63	3423	218,517,474	93.53%	15,991,232	11.52%
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 II: Matrix characteristics and storage overhead for selected test matrices when using ELLPACK, or SELL-P format. 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).

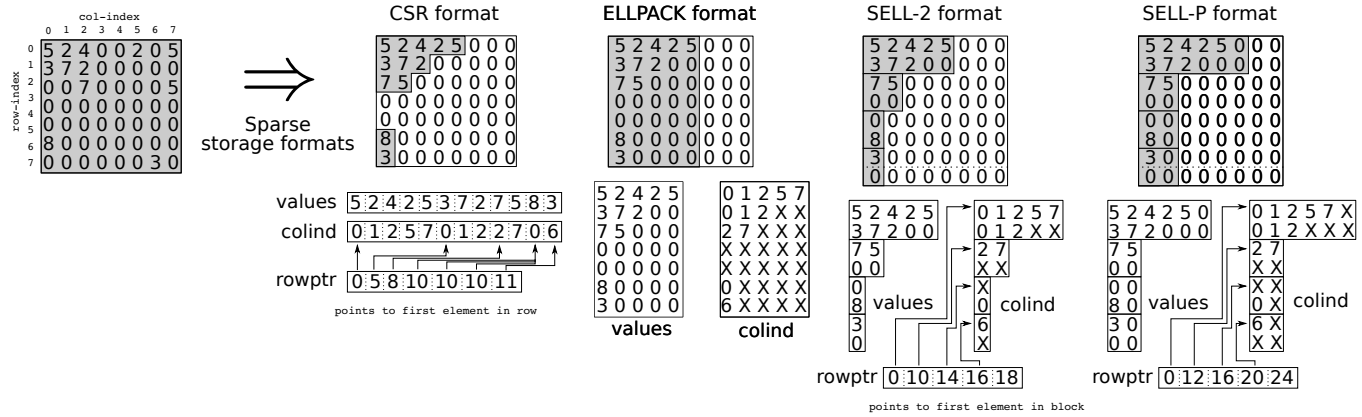


Fig. 3: Visualizing the CSR, ELLPACK, SELL-C, SELL-P formats. The memory demand corresponds to the grey areas. Note that choosing the block size 2 for SELL-C (SELL-2) and SELL-P requires adding a zero row to the original matrix. Furthermore, padding the SELL-P format to a row length divisible by 2 requires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

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 III: Asymptotic DP performance [GFLOP/s] of sparse test matrices and a large number of vectors with a set of 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 kernel against the respective best SpMV. See Table I for the respective matrix characteristics.

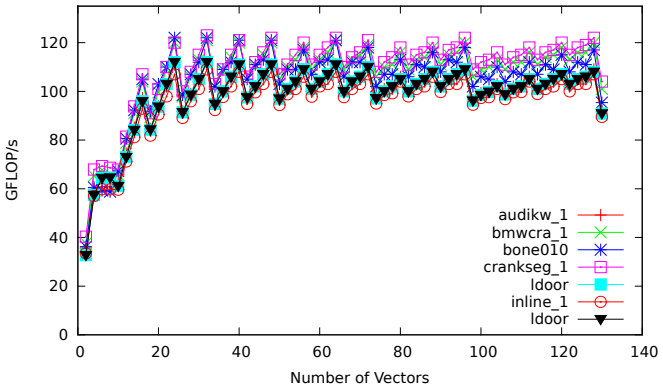


Fig. 5: Performance scaling with respect to the vector count of the SpMM kernel for selected matrices.

step of 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 as "cuSPARSE CSRSpMM v2"in Figure 6). The developed 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 highly tuned cuSPARSE SpMM ranges between 13% and 41% (see results for CANT and CRANK, respectively).

Figure 7 compares our SpMM kernel on a K40 with the DCSRMM kernel from Intel's MKL (routine mkl_dcsrmm) on two eight-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. The row-major storage allows on both architectures for significantly higher GFLOP rates. The CPU runs are using the `numactl --interleave=all` policy, which is well known to improve performance. The

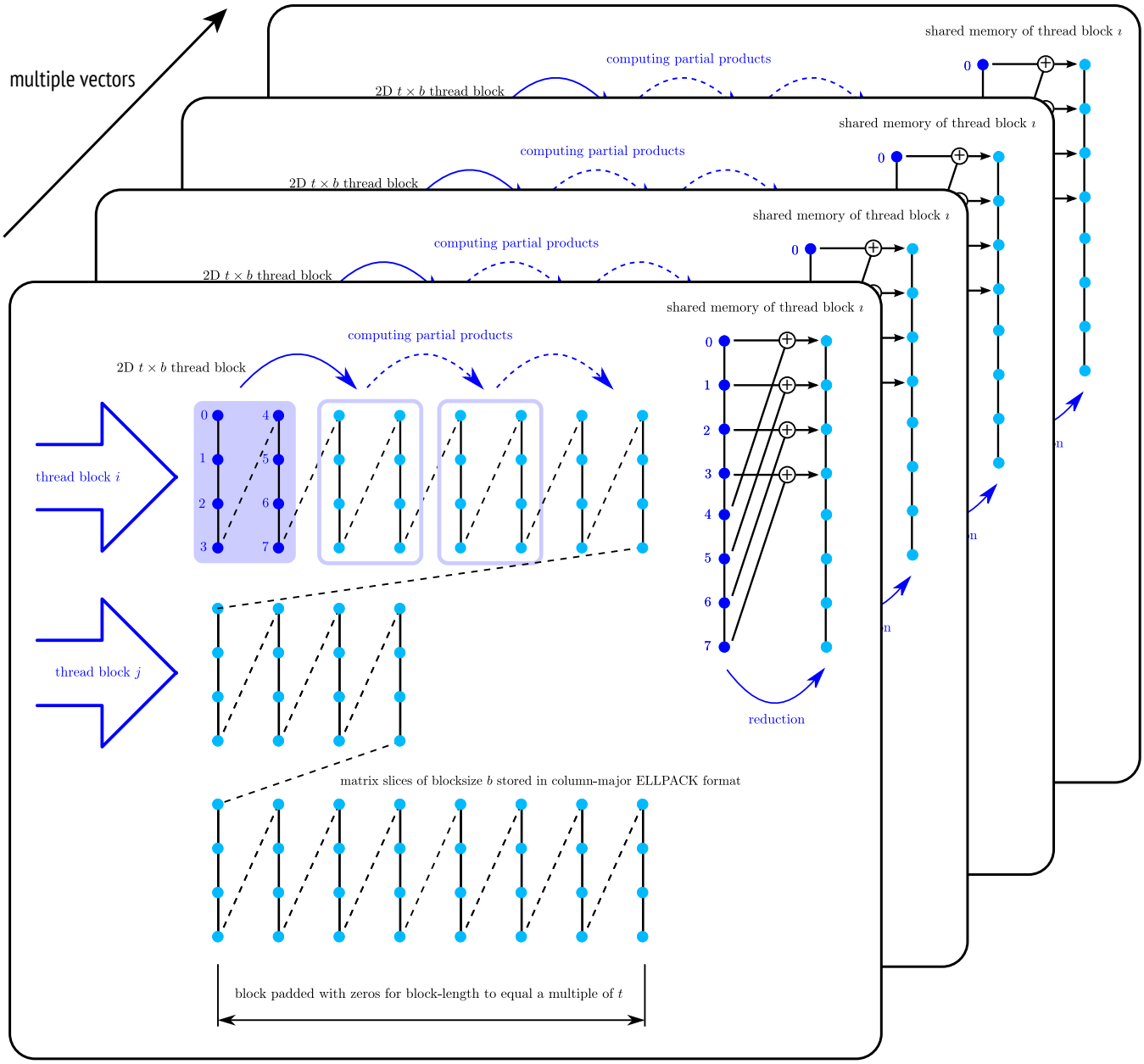


Fig. 4: Visualization of the SELL-P memory layout and the SELL-P SpMM kernel including the reduction step using the blocksize $b = 4$ (corresponding to SELL-4), and always assigning two threads to every row ($t = 2$). Adding a z-dimension to the thread-block allows to process multiple vectors simultaneously.

performance obtained is consistent with benchmarks provided by Intel [39]. 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.

VI. LOBPCG GPU Performance

Finally, we want to quantify how the developed SpMM improves the performance of the LOBPCG GPU implementation. For this purpose, we benchmark two versions of the LOBPCG implementation, one using a set of consecutive SpMVs to generate the search directions, and one where we integrate the developed SpMM kernel. Furthermore, we compare against the multithreaded CPU implementation of LOBPCG provided by Andrew Knyazev in the BLOPEX package [7]. As popular software libraries like PETSc [40] and Hypre [41] provide interfaces to this implementation [8], we may consider this code as the state-of-the-art CPU implementation of LOBPCG. For the benchmark results, we used the BLOPEX code via the Hypre interface on the hardware platform listed in Section IV. For optimal utilization of the Sandy Bridge architecture, we enable hyperthreading and execute the eigensolver using 32 OpenMP threads.

The LOBPCG implementation in BLOPEX is matrix free, i.e., the user is allowed to provide their choice of Sp-

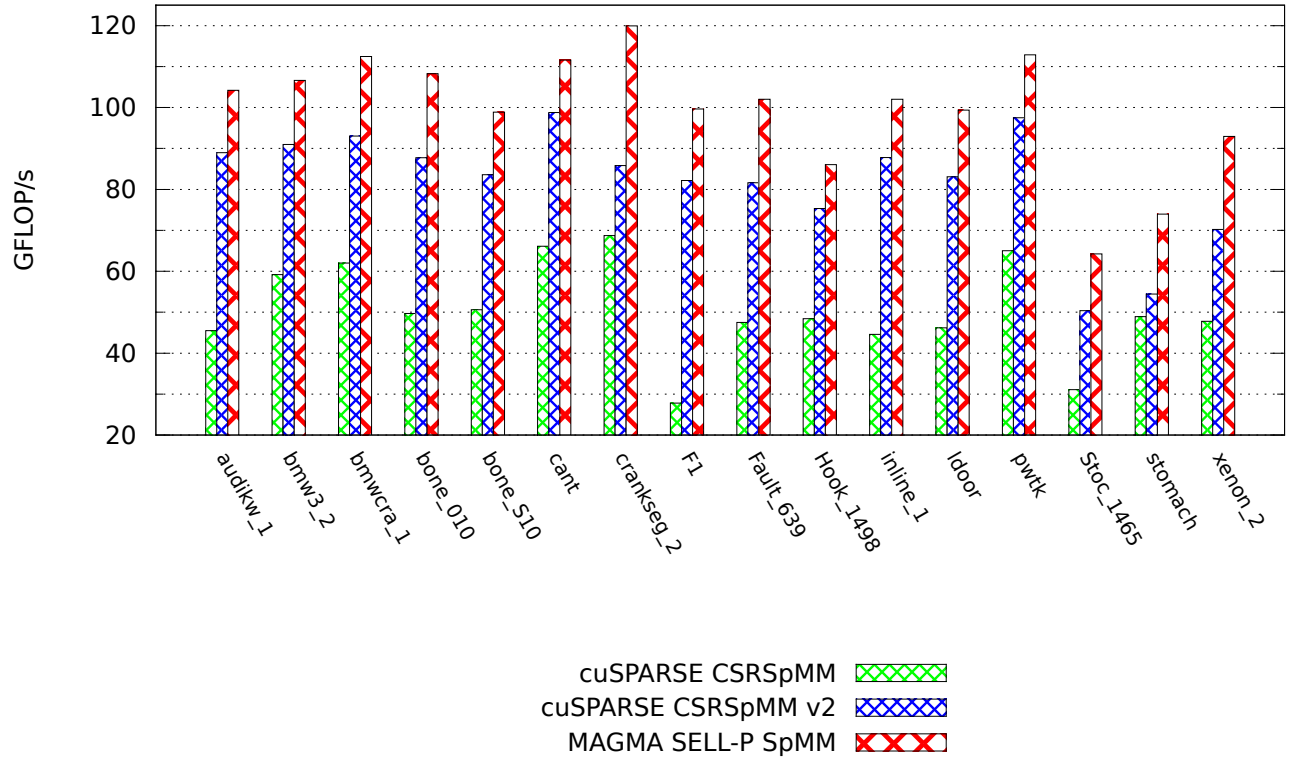


Fig. 6: Performance comparison between the developed `SpMM` kernel and the CSRSpMM kernels provided by NVIDIA for selected matrices and a set of 64 vectors.

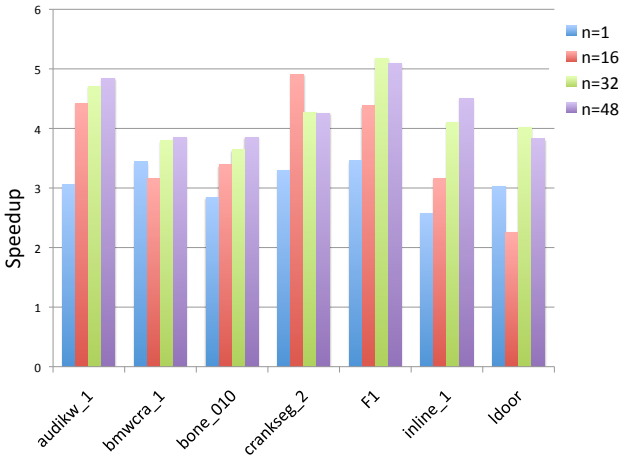


Fig. 7: Speedup of the developed `SpMM` kernel on a K40 *vs.* the DCSRMM kernel provided by Intel's MKL on two eight-core Intel Xeon E5-2690s for selected matrices and number of vectors n.

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

MV/SpMM implementation. In these experiments we use the Hypre interface to BLOPEX, linked with the MKL library.

The validity of the results is ensured as the convergence of the eigenvectors is matching the BLOPEX convergence. In Figure 8 we visualize the convergence of 10 eigenvectors for the AUDI test matrix.

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 nonzeros 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

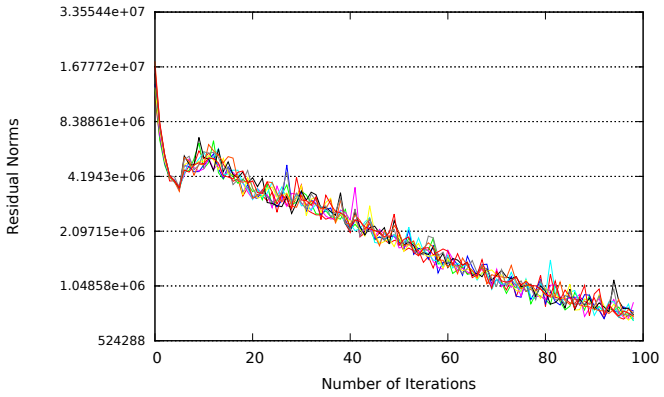


Fig. 8: Visualizing the convergence of 10 eigenvectors when applying the developed GPU implementation of LOBPCG based on the SpMM kernel to the AUDI test matrix.

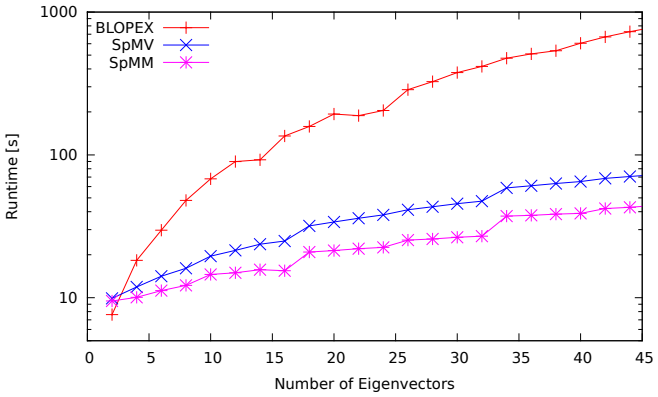


Fig. 9: Runtime needed to complete 100 iterations on the AUDI problem using either the BLOPEX code via the Hypre interface or the GPU implementation using either a sequence of SpMVs or the SpMM kernel to generate the search directions.

dense matrix). The left part of the sum reflects the SpMM operation generating the Krylov vectors, the right part contains the remaining operations including the orthogonalization of the search directions. Due to the n_v^2 term, we may expect the runtime to increase superlinearly with the number of vectors, which 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 or the GPU implementation using either a sequence of SpMVs or the SpMM kernel to generate the search directions. 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 improvement of correspondingly 30% and 20% reflects 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 (1) and the matrix specifications in Table I). 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, see Figure 10 showing the runtime needed by the SpMM-based GPU implementation of LOBPCG to complete 100 iterations for different test matrices.

To complete the performance analysis, we report in Figure 11 the speedup factors of the GPU LOBPCG *vs.* the BLOPEX code via its Hypre interface. We observe that as soon as 16 eigenvectors are needed, the GPU implementation using the consecutive SpMVs outperforms the CPU code 5×, while for the SpMM-based algorithm the acceleration is 10×. The 5× speedup when using the consecutive SpMVs on the GPU indicates that the Hypre interface to LOBPCG is not blocking the SpMVs. Based on the kernels' analysis, the expectation is that an optimized CPU code (blocking the SpMVs) would achieve about the same performance as the GPU LOBPCG without blocking, and would be about 3 to 5× slower than the blocked

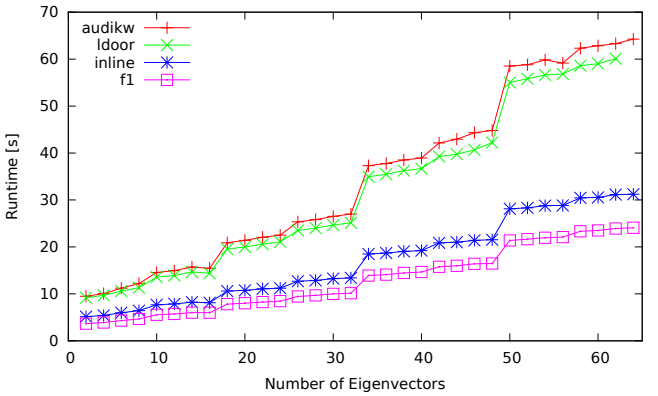


Fig. 10: Runtime needed to complete 100 iterations using the LOBPCG GPU implementation based on SpMM.

version. Computing more vectors reduces the fraction of SpMV flops to the total flops (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 20 and up to 35× for 48 vectors, shows that there are other missed optimization opportunities in the CPU implementation. In particular, these 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 squere resulting matrices A^TB – required modifications to the standard GEMM algorithm for large matrices [42]. What worked very well is splitting the A^TB GEMM into smaller GEMMs based on tuning the MAGMA GEMM [42] for the particular small sizes, all grouped for execution into a single batched GEMM, followed by addition of the local results [17]. The acceleration factor against a similarly optimized CPU code, based on our performance analysis, must be in the range of 3 to 5×.

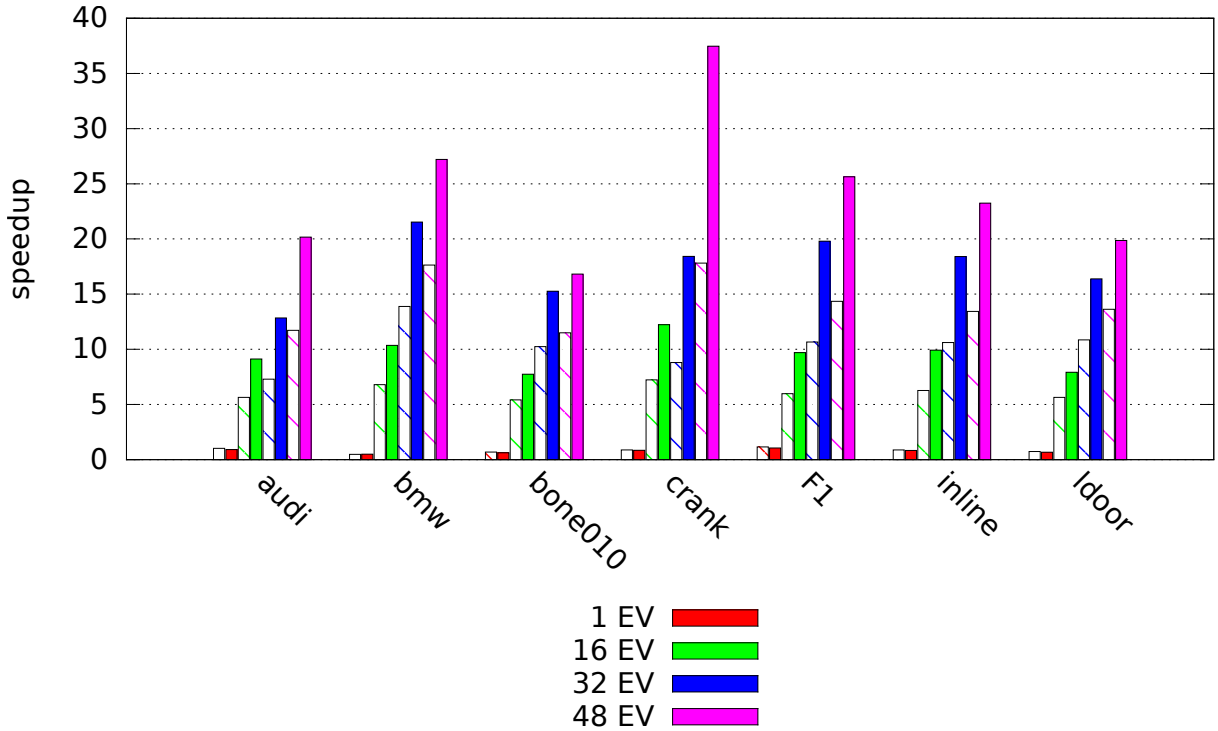


Fig. 11: Speedup of the GPU implementation of LOBPCG over the BLOPEX CPU code (Hypre interface to BLOPEX linked with MKL) using consecutive SpMVs (striped) and the SpMM kernel (solid), respectively.

VII. Summary and Outlook

In this paper we have presented a heterogeneous CPU-GPU algorithm design and optimized implementation of the LOBPCG eigensolver. The benefit of using blocking routines like the LOBPCG is based on a more efficient use of hardware. As opposed to running at the low performance of a SpMV kernel, which is typical for Krylov subspace methods, the LOBPCG runs at the speed of a SpMM kernel that is up to $6\times$ faster on GPUs such as the NVIDIA's K40. Instead of the standard Krylov subspace methods' memory-bound performance of 20 to 25 GFlop/s (in double precision on a K40), the LOBPCG computes a small set of eigenstates at a typical rate of 100 to 140 GFlop/s. We detailed the data structures, algorithmic designs, and optimizations needed to reach this performance. Most notably, these were the designs and optimizations for the SpMM kernel, the specific GEMM, and orthogonalization routines needed. Compared to CPUs, one K40 outperforms two eight-core Intel Sandy Bridge cores by 3 to $5\times$. In practice, our heterogeneous LOBPCG outperformed the Hypre interface of the BLOPEX CPU implementation by more than an order of magnitude when computing a small set of eigenstates. This shows that even for multicore CPUs, where the HPC software stack is considered to be better established than the HPC software stack for the more recent GPU architectures, there are many missed optimization opportunities. There is a compelling need to build on this work to provide users with the full potential of blocking algorithms.

The developed SpMM routine, the specific GEMMs, and orthogonalizations, are building blocks that serve as the foundation for not only block-Krylov, but also for many other methods relying on blocking for acceleration in general. As such, there is increasing interest in both the kernels and the methods that can use them. For this reason, future research will focus on how to efficiently integrate and further extend these building blocks into other block-Krylov methods.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, Department of Energy grant No. DE-SC0010042, and NVIDIA Corporation.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[2] 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.

[3] 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 *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 44–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.1

[4] 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 *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188504

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

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

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

[8] 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. [Online]. Available: http://dblp.uni-trier.de/db/journals/siamsc/siamsc29.html#KnyazevALO07

[9] M. Naumov, "Preconditioned block-iterative methods on gpus," *PAMM*, vol. 12, no. 1, pp. 11–14, 2012. [Online]. Available: http://dx.doi.org/10.1002/pamm.201210004

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

[11] 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.

[12] Innovative Computing Laboratory, UTK, "Software distribution of MAGMA version 1.5," http://icl.cs.utk.edu/magma/, 2014.

[13] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parellel Comput. Syst. Appl.*, vol. 36, no. 5-6, pp. 232–240, 2010, http://dx.doi.org/10.1016/j.parco.2009.12.005DOI: 10.1016/j.parco.2009.12.005.

[14] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-avoiding parallel and sequential qr factorizations," *CoRR*, vol. abs/0806.2159, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0806.html#abs-0806-2159

[15] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding qr decomposition for gpus," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-131, Oct 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-131.html

[16] 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. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2011.html#AgulloADFLTT11

[17] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra, "Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs," *VECPAR 2014*, jan 2014.

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

[19] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, E. K. U. Gross, and A. Rubio, "octopus: a tool for the application of time-dependent density functional theory," *phys. stat. sol. (b)*, vol. 243, no. 11, pp. 2465–2488, 2006. [Online]. Available: http://dx.doi.org/10.1002/pssb.200642067

[20] 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 Trans. Math. Softw.*, vol. 36, no. 3, pp. 13:1–13:23, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1527286.1527287

[21] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, "An Overview of Trilinos," Sandia National Laboratories, Tech. Rep. SAND2003-2927, 2003.

[22] X. Gonze, J.-M. Beuken, R. Caracas, F. Detraux, M. Fuchs, G.-M. Rignanese, L. Sindic, M. Verstraete, G. Zerah, F. Jollet, M. Torrent, A. Roy, M. Mikami, P. Ghosez, J.-Y. Raty, and D. Allan, "First-principles computation of material properties: the ABINIT software project," *Computational Materials Science*, vol. 25, no. 3, pp. 478 – 492, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0927025602003257

[23] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2nd ed., NVIDIA Corporation, August 2009.

[24] A. V. Knyazev, "Preconditioned eigensolvers - an oxymoron?" *Electronic Transactions on Numerical Analysis*, vol. 7, pp. 104–123, 1998.

[25] U. Hetmaniuk and R. Lehoucq, "Basis selection in LOBPCG," *Journal of Computational Physics*, vol. 218, no. 1, pp. 324 – 332, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999106000866

[26] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168927404001990

[27] P. Benner and T. Mach, "Locally optimal block preconditioned conjugate gradient method for hierarchical matrices," *PAMM*, vol. 11, no. 1, pp. 741–742, 2011. [Online]. Available: http://dx.doi.org/10.1002/pamm.201110360

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

[29] D. Kressner, M. Pandur, and M. Shao, "An indefinite variant of lobpcg for definite matrix pencils," *Numerical Algorithms*, pp. 1–23, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11075-013-9754-3

[30] 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. [Online]. Available: http://books.google.com/books?id=i5TzGgAACAAJ

[31] N. Corp., *NVIDIA CUDA TOOLKIT V5.5*, July 2013.

[32] 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.

[33] 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.

[34] S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc, "Sparse matrix vector multiplication on multicore and accelerator systems," in *Scientific Computing with Multicore Processors and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, 2010.

[35] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proc. IPDPS*, 2011, pp. 721–733.

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

[37] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11515-8_10

[38] 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.

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

[40] S. Balay, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2014. [Online]. Available: http://www.mcs.anl.gov/petsc

[41] R. D. Falgout and U. M. Yang, "Hypre: A Library of High Performance Preconditioners," in *Computational Science - ICCS 2002, International Conference, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III*, ser. Lecture Notes in Computer Science, P. M. A. Sloot, C. J. K. Tan, J. Dongarra, and A. G. Hoekstra, Eds., vol. 2331. Springer, 2002, pp. 632–641.

[42] 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.