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

Hartwig Anzt
University of Tennessee
hanzt@icl.utk.edu

Stanimire Tomov
University of Tennessee
tomov@icl.utk.edu

Jack Dongarra
University of Tennessee
dongarra@icl.utk.edu

ABSTRACT

This paper presents a heterogeneous CPU-GPU implementation for a sparse iterative eigensolver – the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG). For the key routine generating the Krylov search spaces via the product of a sparse matrix and a block of vectors, we propose a GPU kernel based on a modified sliced ELLPACK format. Blocking a set of vectors and processing them simultaneously accelerates the computation of a set of consecutive `SpMV`s significantly. Comparing the performance against similar routines from Intel's MKL and NVIDIA's cuSPARSE library we identify appealing performance improvements. We integrate it into the highly optimized LOBPCG implementation. Compared to the BLOBEX CPU implementation running on two eight-core Intel Xeon E5-2690s, we accelerate the computation of a small set of eigenvectors using NVIDIA's K40 GPU by typically more than an order of magnitude.

ACM Classification Keywords

G.1.3 Numerical Analysis: Numerical Linear Algebra—*Eigenvalues and eigenvectors*

Author Keywords

LOBPCG eigensolver, GPU acceleration, SpMV, SpMM

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 principal 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 [38], 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.

SpringSim'15, April 13-16, 2015, Alexandria, VA.

DOI string from ACM form confirmation

While there exist numerous efforts to adapt iterative linear solvers 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 [38]. When targeting symmetric positive definite eigenvalue problems, the recently developed Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG, see [27]) 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 [42][41]. 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 [1] and NVIDIA's cuSPARSE [35] 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 [24] code for which the software libraries *PETSc* and *hypre* provide an interface [28]. For matrices taken from the University of Florida Matrix Collection, we achieve significant acceleration when computing a set of the respective eigenstates.

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 [35]. 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 row-major matrix as input, that can be used in combination with a preprocessing step transposing the matrix to achieve significantly higher performance [34]. 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. There has been extensive research on both its acceleration and stability. Besides the classical and modified Gram-Schmidt orthogonalizations [19] and orthogonalizations based on LAPACK (xGEQRF + xUNGQR) [4] and correspondingly MAGMA for GPUs [22][39], recent work includes communication-avoiding QR [16], also developed for GPUs [5][3]. 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 [43] (also applied to a CA-GMRES for GPUs).

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

LOBPCG implementations: The BLOPEX package maintained by A. 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 [28]. Also Scipy [23], octopus [14] and Anasazi [8] part of the Trilinos library [20] feature LOBPCG implementations.
The first GPU implementation of LOBPCG is from 2011 in the ABINIT material science package [17]. The implementation, realized in fortran90, benefits from utilizing the generic linear algebra routines available in the CUDA [37] and MAGMA [22][39] GPU libraries. More recently, NVIDIA anounced that LOBPCG will be included in the GPU-accelerated Algebraic Multigrid Accelerator AmgX [1].

LOBPCG

LOBPCG stands for Locally Optimal Block Preconditioned Conjugate Gradient method [27][26]. 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 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 small, and therefore special care and modifications must be taken (see [27][21]). While the LOBPCG convergence characteristics usually benefit from using an an application-specific preconditioner [7][12][29][30][25], 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 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. For stability, various orthogonalizations are performed, following the LOBPCG Matlab code from A. Knyazev [2]. We used our highly optimized GPU

[1] https://developer.nvidia.com/amgx
[2] http://www.mathworks.com/matlabcentral/fileexchange/48-lobpcg-m

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). More stable versions, including Cholesky/SVD QR iterations and the mixed-precision Cholesky QR [43], as well as LAPACK/MAGMA based, CGS, and MGS for GPUs are also an option that we provide.

SPMM 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 [6]). 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 [35].

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 [10][40134013]. In the CSR format [10], 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. 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 [11]. However, the ELLPACK format incurs a storage overhead for the general case: The maximum number of nonzero elements aggregated in one row determines how many elements are stored per row – eventually filled with explicit zeros. Hence, the overhead is determined by the maximum number of nonzeros in one row and the average number of nonzeros per row (see Table 1). Depending on the associated 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 [32][31]), 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 [11] 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 Table 1 reveals that the blocking strategy may still render significant memory savings compared to ELLPACK, 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 [37]. 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 [6]. 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 1. 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

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 1: Matrix characteristics and storage overhead for selected test matrices from the Tim Davis Matrix Collection [15] 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).

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

EXPERIMENTAL RESULTS

Hardware Setup: We use two Intel Xeon E5-2670 (Sandy Bridge) CPUs 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 [37], version 5.5 [36], while we also include in the performance comparisons routines taken from NVIDIA's cuSPARSE [35] library. On the CPU, Intel's MKL [1] is used in version 11.0, update 5. Note that the CPU-based implementations use the "numactl –interleave=all" option when beneficial.

Performance of `SpMM` for SELL-P: In Table 2 we compare the asymptotic performance achieved by the `SpMV` and `SpMM` kernels taken from Intel's MKL library [1] and NVIDIA's cuSPARSE library [35] with the developed SELL-P `SpMV` and `SpMM` kernels that are available in the MAGMA open source software stack [22].

Both GPU implementations for the `SpMM` (cuSPARSE and MAGMA) assume the vectors to be stored in row-major data format. This typically results in significantly higher performance [34]. As most algorithms require column-major storage, and for a fair comparison to the CPU implementation, the transposition operation is included in the performance of those kernels. The CPU runs are using the `numactl --interleave=all` policy, which is well known to improve performance. The performance obtained for the MKL routines is consistent with benchmarks provided by Intel [2].

Matrix	Intel MKL CSR	Intel MKL `SpMM`	NVIDIA cuSPARSE CSR	NVIDIA cuSPARSE HYB	NVIDIA cuSPARSE `SpMM`	MAGMA SELL-P	MAGMA `SpMM`
AUDI	7.24	22.5	21.9	19.3	88.95	22.1	104.21
BMW	6.86	32.2	17.7	16.9	93.04	23.6	112.46
BONE010	7.77	30.5	22.3	20.7	87.71	22.3	108.26
F1	5.64	20.1	24.2	19.1	82.15	19.6	99.63
INLINE	8.10	28.9	15.5	14.9	87.76	21.1	102.00
LDOOR	6.78	41.5	25.2	19.3	83.09	20.7	99.37

Table 2: Asymptotic DP performance [GFLOP/s] of sparse test matrices and a large number of vectors with a set of consecutive `SpMV`s (MKL CSR, cuSPARSE CSR, cuSPARSE HYB, MAGMA SELL-P `SpMV`) and the respective `SpMM` kernels. See Table 1 for the respective matrix characteristics.

Depending on the matrix characteristics, the GPU `SpMV`s are between 2 and 4× faster than MKL. This is expected from the compute and bandwidth capabilities of the two architectures. Comparing the performance of the GPU kernels, the SELL-P kernel is the winner in 4 or 6 cases. On the CPU, the MKL `SpMM` routine achieves abut 4.1× better performance than the `SpMV` kernel. Similar improvements can be observed on the GPU: The performance of the cuSPARSE `SpMM` kernel is 4.1× higher than the CSR kernel, and 4.7× higher than the HYB kernel; The MAGMA `SpMM` kernel achieves 4.8× better performance than the `SpMV` kernel. Comparing CPU with GPU, the cuSPARSE `SpMM` is on average 3× faster than the MKL `SpMM`, however outperformed for all test matrices by the developed SELL-P `SpMM` kernel, accelerating the computation by factors between 2.5 and 5. With an average speedup of 3.6 over the MKL, the SELL-P `SpMM` performance typically exceeds 100 GFLOP/s.

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 `SpMV`s

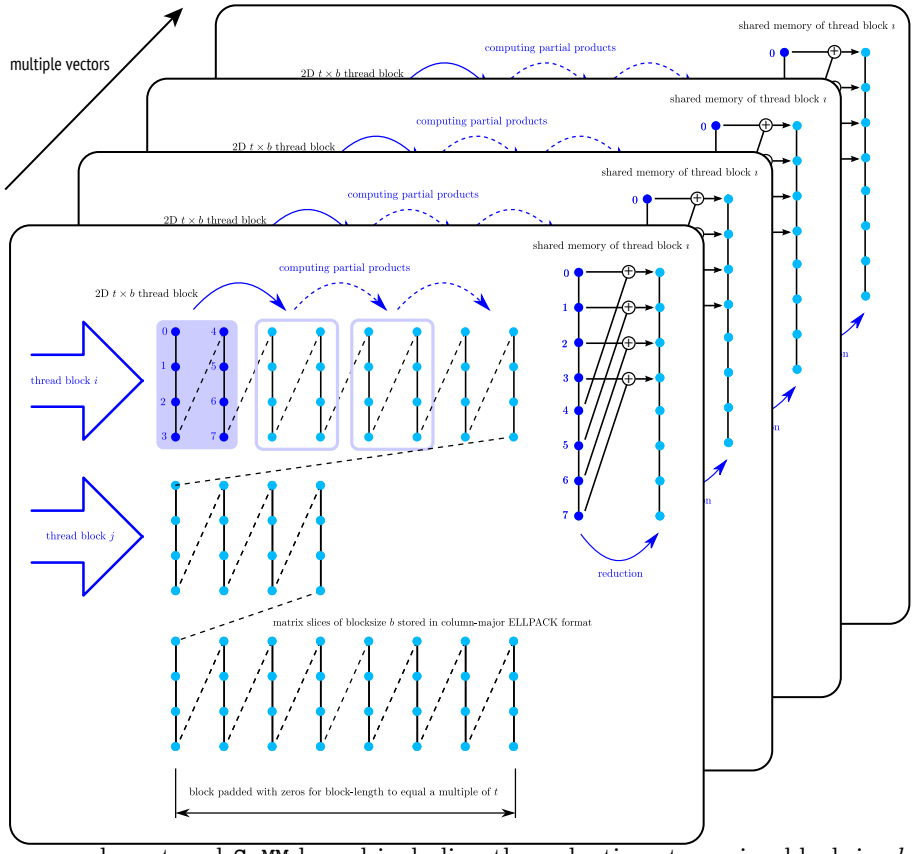

Figure 1: SELL-P memory layout and `SpMM` kernel including the reduction step using blocksize $b = 4$ (SELL-4), and assignement of two threads to row ($t = 2$). Adding a z-dimension to the thread-block allows to process multiple vectors simultaneously.

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 [24]. As popular software libraries like PETSc [9] and Hypre [18] provide interfaces to this implementation [28], 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 its the Hypre interface. 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 SpMV/SpMM implementation. In these experiments we use the Hypre interface to BLOPEX, linked with the MKL library.

The convergence on the GPU is matching the BLOPEX convergence. 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 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 2 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 `SpMV`s 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 `SpMV`s for the 32 and 48 eigenvector problems, respectively (see equation (1) and the matrix specifications in Table 1). While the BLOPEX implementation also shows some variances for different numbers of vectors, the runtime pattern of the GPU LOBPCG reflects the efficiency of the orthogonalization routines favoring cases where 16, 32, or 48 vectors are processed. This pattern is amplified when

replacing the consecutive SpMVs with the SpMM, as this kernel also promotes certain column-counts of the tall and skinny dense matrix.

To complete the performance analysis, we report in Figure 3 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 by more than 5×, while for the SpMM-based algorithm the acceleration is on average close to 8×. 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, which is about 3 to 5× slower than the blocked 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 [33]. Benefits are in particular drawn from splitting the A^TB GEMM into smaller GEMMs based on tuning the MAGMA GEMM [33] for small sizes. The execution is then grouped into a single batched GEMM, followed by addition of the local results [43]. Based on our performance analysis, the acceleration factor against a similarly optimized CPU code would be in the range of 2.5 to 5×.

SUMMARY AND OUTLOOK

We presented a heterogeneous CPU-GPU algorithm design for 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 presented LOBPCG runs at the speed of a SpMM kernel designed for the SELL-P format. On NVIDIA's K40 GPU, this kernel outperforms Intel's SpMM on two eight-core Intel Sandy Bridge cores by 2.5 to 5×. 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. 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 reveals that

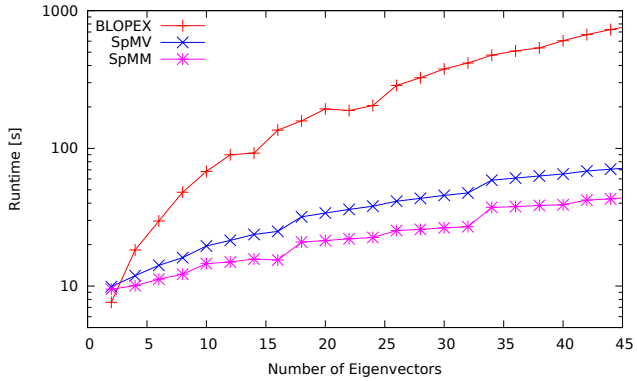

Figure 2: Runtime to complete 100 iterations on the AUDI problem using either the BLOPEX code via the Hypre interface or the GPU implementation using either a sequence of SpMVs or the SpMM kernel to generate the search directions.

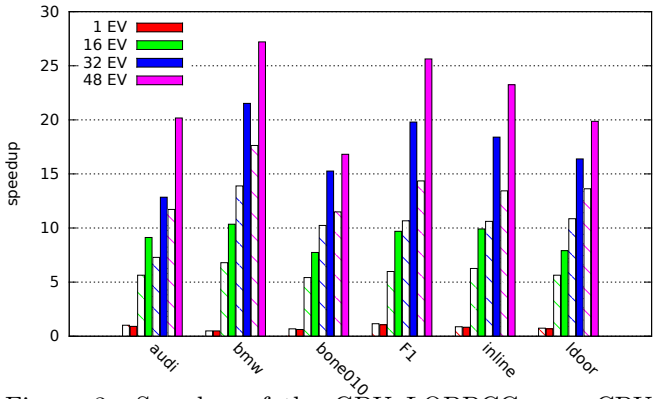

Figure 3: Speedup of the GPU LOBPCG over CPU BLOPEX (Hypre interface linked with MKL) using consecutive SpMVs (striped) and the SpMM kernel (solid), respectively.

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. The developed SpMM routine, the specific GEMMs, and orthogonalizations, serve as key building blocks not only block-Krylov, but also for other methods that rely on blocking strategies. Hence, the kernel design and findings presented in this paper may be used to accelerate other methods in a similar fashion.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, Department of Energy grant No. DE-SC0010042, NVIDIA, and the Russian Scientific Fund (Agreement N14-11-00190).

REFERENCES

1. Intel® Math Kernel Library for Linux* OS. Document Number: 314774-005US, October 2007. Intel.
2. Intel® Math Kernel Library. Sparse BLAS and Sparse Solver Performance Charts: DCSRGEMV and DCSRMM, October 2014. Intel Corporation.
3. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., and Tomov, S. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *IPDPS*, IEEE (2011), 932–943.
4. Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. *LAPACK Users' Guide (Third Ed.)*. SIAM, Philadelphia, PA, USA, 1999.
5. Anderson, M., Ballard, G., Demmel, J., and Keutzer, K. Communication-avoiding qr decomposition for gpus. Tech. Rep. UCB/EECS-2010-131, EECS Department, UC Berkeley, Oct 2010.
6. Anzt, H., Tomov, S., and Dongarra, J. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. Tech. Rep. ut-eecs-14-727, University of Tennessee, March 2014.
7. Arbenz, P., and Geus, R. Multilevel preconditioned iterative eigensolvers for maxwell eigenvalue problems. *Applied Numerical Mathematics 54*, 2 (2005), 107 – 121. 6th IMACS International Symposium on Iterative Methods in Scientific Computing.
8. Baker, C. G., Hetmaniuk, U. L., Lehoucq, R. B., and Thornquist, H. K. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Trans. Math. Softw. 36*, 3 (July 2009), 13:1–13:23.
9. Balay, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., and Zhang, H. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2014.
10. Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
11. Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on CUDA, Dec. 2008.
12. Benner, P., and Mach, T. Locally optimal block preconditioned conjugate gradient method for hierarchical matrices. *PAMM 11*, 1 (2011), 741–742.
13. Buluç, A., Williams, S., Oliker, L., and Demmel, J. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS* (2011), 721–733.
14. Castro, A., Appel, H., Oliveira, M., Rozzi, C. A., Andrade, X., Lorenzen, F., Marques, M. A. L., Gross, E. K. U., and Rubio, A. octopus: a tool for the application of time-dependent density functional theory. *phys. stat. sol. (b) 243*, 11 (2006), 2465–2488.
15. Davis, T. A., and Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw. 38*, 1 (Dec. 2011), 1:1–1:25.
16. Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. Communication-avoiding parallel and sequential qr factorizations. *CoRR abs/0806.2159* (2008).
17. et al., X. G. First-principles computation of material properties: the ABINIT software project. *Computational Materials Science 25*, 3 (2002), 478 – 492.
18. Falgout, R., and Yang, U. Hypre: A Library of High Performance Preconditioners. In *ICCS 2002, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III*, P. M. A. Sloot, C. J. K. Tan, J. Dongarra, and A. G. Hoekstra, Eds., vol. 2331 of *Lecture Notes in Computer Science*, Springer (2002), 632–641.
19. Golub, G. H., and Van Loan, C. F. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
20. Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A. An Overview of Trilinos. Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.
21. Hetmaniuk, U., and Lehoucq, R. Basis selection in LOBPCG. *Journal of Computational Physics 218*, 1 (2006), 324 – 332.
22. Innovative Computing Laboratory, UTK. Software distribution of MAGMA version 1.5. `http://icl.cs.utk.edu/magma/`, 2014.
23. Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–.
24. Knyazev, A. https://code.google.com/p/blopex/.
25. Knyazev, A., and Neymeyr, K. *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, 2001.
26. Knyazev, A. V. Preconditioned eigensolvers - an oxymoron? *Electronic Transactions on Numerical Analysis 7* (1998), 104–123.

27. Knyazev, A. V. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput 23* (2001), 517–541.

28. Knyazev, A. V., Argentati, M. E., Lashuk, I., and Ovtchinnikov, E. E. Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc. *SIAM J. Scientific Computing 29*, 5 (2007), 2224–2239.

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

30. Kressner, D., Pandur, M. M., and Shao, M. An indefinite variant of lobpcg for definite matrix pencils. *Numerical Algorithms* (2013), 1–23.

31. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. A unified sparse matrix data format for modern processors with wide simd units. *CoRR abs/1307.6209* (2013).

32. Monakov, A., Lokhmotov, A., and Avetisyan, A. 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, Springer-Verlag (Berlin, Heidelberg, 2010), 111–125.

33. Nath, R., Tomov, S., and Dongarra, J. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl. 24*, 4 (Nov. 2010), 511–515.

34. Naumov, M. Preconditioned block-iterative methods on gpus. *PAMM 12*, 1 (2012), 11–14.

35. NV. *CUSPARSE LIBRARY*, July 2013.

36. NVIDIA. *NVIDIA CUDA TOOLKIT V5.5*, July 2013.

37. NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 ed., August 2009.

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

39. Tomov, S., Dongarra, J., and Baboulin, M. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl. 36*, 5-6 (2010), 232–240. http://dx.doi.org/10.1016/j.parco.2009.12.005DOI: 10.1016/j.parco.2009.12.005.

40. Williams, S., Bell, N., Choi, J., Garland, M., Oliker, L., and Vuduc, R. 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.

41. Yamada, S., Imamura, T., Kano, T., and Machida, M. 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, ACM (New York, NY, USA, 2006).

42. Yamada, S., Imamura, T., and Machida, M. 16.447 tflops and 159-billion-dimensional exact-diagonalization for trapped fermion-hubbard model on the earth simulator. SC '05, IEEE Computer Society (Washington, DC, USA, 2005), 44–.

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