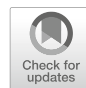
GPU Optimization of Large-Scale Eigenvalue Solver



Pavel Kůs, Hermann Lederer, and Andreas Marek

Abstract We present a GPU implementation of a large-scale eigenvalue solver as a part of the ELPA library. We describe the methodology of utilizing the GPU accelerators within an already well optimized MPI-based code. We present numerical results using two different HPC systems equipped with modern GPU accelerators and show the performance benefits of the GPU version.

1 Introduction

Solving large eigenvalue systems is, apart from being a classical problem of linear algebra with a broad range of applications, a substantial part of many important problems in materials science, computational chemistry, and namely electronic structure theory, where a key task is the solution of Schrödinger-like eigenproblems [1]. Since the solution of the eigenproblem scales as $O(n^3)$, where n is the size of the matrix, it can easily dominate the whole compute-time for large-scale calculations.

The ELPA library [1–3] is a well established eigensolver library used by many computational chemistry and materials science codes. It provides an efficient implementation in distributed memory with good scaling properties for many thousands of CPU cores as well as optimizations targeting various particular architectures. It also contains specific algorithmic advantages for problems where only a certain part of eigenvalues and eigenvectors are sought.

Since the role of accelerated HPC systems with a high peak performance is of growing importance, their efficient usage should be ensured. For this reason, substantial efforts are taken to adapt HPC applications to GPU computing. In this contribution we describe our ongoing effort of improving the performance of the

P. Kůs (✉) · H. Lederer · A. Marek
Max Planck Computing and Data Facility, Garching bei München, Germany
e-mail: pavel.kus@mpcdf.mpg.de; hermann.lederer@mpcdf.mpg.de; andreas.marek@mpcdf.mpg.de

© Springer Nature Switzerland AG 2019

F. A. Radu et al. (eds.), *Numerical Mathematics and Advanced Applications ENUMATH 2017*, Lecture Notes in Computational Science and Engineering 126,
https://doi.org/10.1007/978-3-319-96415-7_9

ELPA library on supercomputers with GPU-equipped nodes. We show multiple ways how GPU support can be employed within an already highly optimized parallel MPI-based code and we discuss their advantages and drawbacks. We comment on both algorithmic and technical aspects of the problem and show performance comparisons.

2 Eigenvalue Solver

We look for the solution of (possibly generalized) eigenvalue problem

$$AV = BV\Lambda.$$

The steps of finding solution to this problem are well known and conceptually simple, see, e.g, [4]. First, if we are to compute a generalized eigenvalue problem, we start by computing the Cholesky decomposition $B = LL^T$ and by transforming the problem to a standard one, $\tilde{A} = \tilde{V}\Lambda$ with

$$\tilde{A} = L^{-1}A(L^{-1})^T, \qquad \tilde{V} = L^T V.$$

The next step is the reduction of the matrix to a tridiagonal form

$$T = Q\tilde{A}Q^T,$$

where $Q = Q_n \cdots Q_2 Q_1$ and $Q^T = Q_1^T Q_2^T \cdots Q_n^T$ are the successive Householder matrices reducing one column of $\tilde{A}$ at a time. The Householder matrices

$$Q_i = I - \beta_i v_i v_i^T$$

are never constructed explicitly, but are always represented only by the Householder vector v_i. In each step, a new Householder vector is computed and stored in place of an eliminated column of A, reducing the memory requirements. The diagonal and sub-diagonal of the resulting matrix are stored separately. Applying transformations on A from both sides complicates blocking and usage of efficient BLAS level 3 kernels. This restriction is alleviated in the two-stage algorithm, which will be briefly described later.

The next step is the solution of the tridiagonal eigenvalue problem

$$T\hat{V} = \hat{V}\Lambda$$

and the final step is the back transformation of the k required eigenvectors

$$\tilde{V} = Q^T\hat{V}.$$

If the original problem was of the generalized form, we have to transform the eigenvectors once more using $\tilde{V} = L^T V$.

The two stage algorithm, featured in ELPA 2, differs from the previous description in performing the conversion to the tridiagonal matrix in two steps. In the first step, the matrix is reduced to a banded matrix. This allows usage of highly optimized BLAS level 3 functions. In the second step, the matrix is further reduced to the tridiagonal form. The two stage tridiagonalization is almost always faster, however, the price to pay is the need to also transform each eigenvector (found by the tridiagonal solver) twice, in order to find the eigenvector of the original system. This makes ELPA 2 an obvious choice when only a small part of the eigenvectors are sought. When, on the other hand, most or all of the eigenvectors are needed, the best choice might depend on other parameters (such as the matrix size, particular hardware, etc) as well. In the rest of this contribution, however, we use only results of the ELPA 1 algorithm.

3 The ELPA Library

Although the basic algorithm is conceptually quite simple, a highly optimized, distributed and scalable implementation using MPI is very challenging. The ELPA library [3] uses a block-cyclic distribution of the matrix, which is well documented and allows codes using the Scalapack library [5] to easily switch to ELPA in order to gain a performance benefit.

ELPA employs very efficient communication patterns in processor rows and columns, benefiting from the block-cyclic distribution of the matrix. Local operations are done by calls to BLAS (or cuBLAS) for parts of local matrices and vectors. Apart from the MPI communication routines the code also contains an OpenMP and a GPU implementation. The complexity is further increased by having both real and complex compute paths as well as single and double precision variants in one code. Moreover, different optimizations are needed for different architectures and thus the code has to be sometimes split on the algorithmic level as well.

4 GPU Computing

Using general purpose graphical processing units (GP-GPU) to accelerate codes in the HPC field is a well established topic. The reason is obvious—the peak-performance of such devices is huge (for example, Nvidia Tesla P100 “Pascal” with a peak-performance of 5.7 TFlop/s is much more powerful than a typical Intel Xeon processor based node with 2 CPUs). It is thus very tempting to use GPUs in HPC calculations. There exist many ongoing projects within the field of linear algebra which try to bring the GPU performance to the users, such as [6] or [7], just to name a few.

There are, however, obstacles caused by a specific computing model of those devices and it is very challenging to reach anything near the theoretical peak-performance of the GPUs. As they were conceived as graphical accelerators, even though they are much more general-purpose nowadays, the performance is still optimal for doing the same operation on large amount of data and it is usually impossible to run the whole HPC calculation on GPUs. The usual approach is to use GPUs as accelerators, when the application runs on CPUs and offloads computationally very intensive (but usually small in terms of code length) parts (called kernels) to the GPU.

There are thus several aspects that have to be taken care of: first, data has to be moved from the CPU memory to the GPU memory (and back). These memories are physically separated and if the memory transfers are not done with care, they can easily degrade the performance of the application. Second, a programming model has to be selected. There are several options. CUDA C offers the best performance, but the code has to be hand-tuned to a specific architecture and is not portable. Furthermore, it works only for Nvidia GPUs. More generic approaches, such as OpenACC have advantages in higher portability and code maintainability (the code can be ported step by step by adding OpenACC directives). The performance, however, is quite often far from optimal. The last option, which is the easiest from the point of view of code development and maintenance is to use GPU kernels from a library, which has been already optimized by the hardware vendor.

We use the latter approach. Our CPU optimization of ELPA works with a distributed matrix, it handles data communications among individual nodes by explicit calls to the MPI library and uses highly optimized BLAS kernels for operations on pieces of the locally stored matrix and vectors. Our GPU implementation is a natural extension of this approach, since we explicitly initiate the data transfers between the main memory and the device memory and then perform local calculations on pieces of the locally stored part of the matrix using calls to the highly optimized cuBLAS library[7]. By this approach, we do not carry the burden of optimizing our code for future GPU architectures, assuming, that such optimization will be done by the hardware vendor.

5 GPU Implementation of ELPA 1

The main focus of the ELPA library is on efficient large-scale calculations, which should remain true for the GPU version as well. We thus still use the MPI-based implementation with a block-cyclic distribution of the matrices. Locally, each MPI task communicates with the GPU to transfer data to and from its memory and to launch the compute kernels on the device. All the memory transfers are done explicitly to avoid unnecessary data movements. We mostly rely on the cuBLAS library, only in ELPA 2 we utilize one kernel, which has been hand-written in CUDA C.

Nowadays, in a typical compute node, the number of CPU cores (several dozens) is much bigger then the number of GPU devices within the node. Thus in a typical setup, multiple MPI tasks have to use the same GPU. In such a setting, to be able to use the GPU efficiently, the Nvidia Multi-process Service [8] has to be used. This daemon has to be started before the run of the application. Its role is to dispatch the requests from individual MPI tasks to the GPU device and to use its streams efficiently. Without the MPS the performance of the application deteriorates significantly.

In general, we tried to keep a single code path for both the CPU and the GPU version of ELPA, with the difference, that instead of calling BLAS operations on parts of the matrices, we call corresponding cuBLAS operations. Obviously, care has to be taken to synchronize the data in GPU memory with the main memory, while keeping the data transfers low. Sometimes, however, more substantial changes in the algorithm have to be done in order to obtain the best performance. The CPU version of ELPA has been highly optimized by keeping the cache reuse in mind. For this reason, many of the algorithms use explicit blocking and try to reuse pieces of the matrices, which are in cache, for multiple operations. This is often not favorable for a GPU, since it cannot benefit from caching, but, rather, it benefits from large amounts of data being processes in one run. Some of the blocking strategies thus had to be changed and the algorithm had to be altered to better suite the GPU.

6 Numerical Results

The success of the ELPA library is caused by a combination of its good node-level performance and its efficient MPI-based communication patterns among multiple nodes within the HPC system. The development of the GPU version, which is described in this contribution, is in principle a node-level optimization, since we do not alter the MPI communication patterns, but only speed-up the local computations by offloading compute intensive kernels to the GPU accelerator. For this reason, we only present performance comparisons using one node, although it can also run on a large HPC system.

We have tested the GPU implementation on two different systems representing two different architectures: the first one is a standard Intel-based system with two Intel Xeon Ivy Bridge processors with 20 cores in total, equipped with two Nvidia Tesla K40m GPUs. The second machine has two IBM Power8 processors and four powerful Nvidia Pascal P100 GPUs. Referred to as the "Minsky" system, it also features the new NVlink interconnect, which should speed-up data transfers between the main and GPU memory. On the Intel system we used the Intel 16 compilers, MKL 2017 (containing the BLAS functions) and CUDA 8 (containing the cuBLAS implementation). On the Minsky machine we used the GNU compilers version 5.3, the ESSL library version 5.5 (containing the BLAS functions) and CUDA 9. We always run our GPU code with the Nvidia Multi process service enabled.

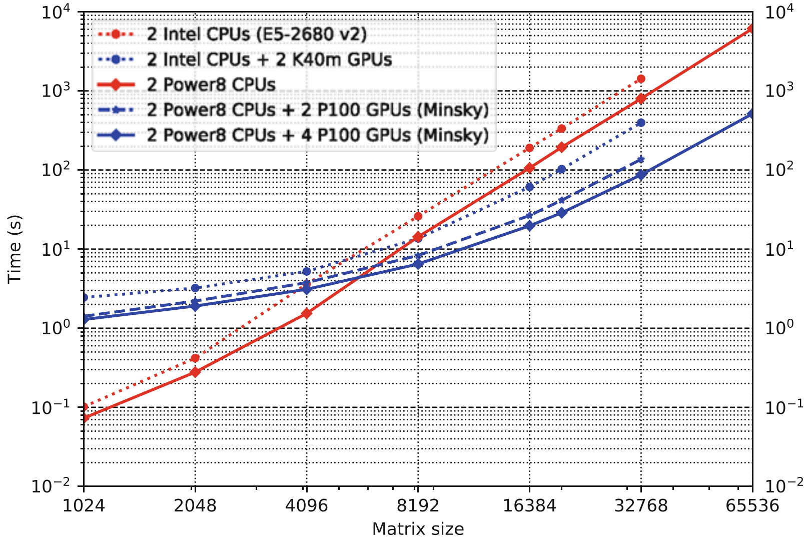


Fig. 1 Total solution times in seconds for CPU only and CPU + GPU versions of ELPA 1. Results from two different architectures are shown for comparison. The problem with the largest matrix is not always computed due to memory limitations

Figure 1 shows the comparison of the total run-time for problems with different matrix sizes. Plotted are the results for both mentioned machines and for both the CPU and the GPU versions of ELPA 1. Comparing the two CPU versions we can see, that the Power8 machine is generally faster, mostly due to higher frequency (4 GHz compared to 2.8 GHz of the Intel machine). For both machines when the matrix is small, the CPU only implementation is significantly faster, since there is not enough workload to saturate the GPUs. From certain threshold on (matrix size of around 5000), however, this behavior changes and the GPU version becomes much faster. In Table 1 we can see timings for a matrix of dimension 32768. We can also compare the achieved speed-ups (bold font) by going from the CPU version to the GPU version on Intel + K40m system (3.6×) and on the Minsky system (5.9× or 9.2×, using 2 or 4 GPUs, respectively).

As it has been described previously, the ELPA 1 algorithm consists of three individual steps, whose performance dependencies are different. In Fig. 2 we show

Table 1 Comparison of solution times for matrix of size 32768 × 32768

	2 CPUs only	2 CPUs + 2 GPUs		2 CPUs + 4 GPUs	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Intel + K40m	1423	396	**3.6**×	–	–
Power8 + P100	798	136	**5.9**×	86.9	**9.2**×

Speed up is with respect to CPU only version

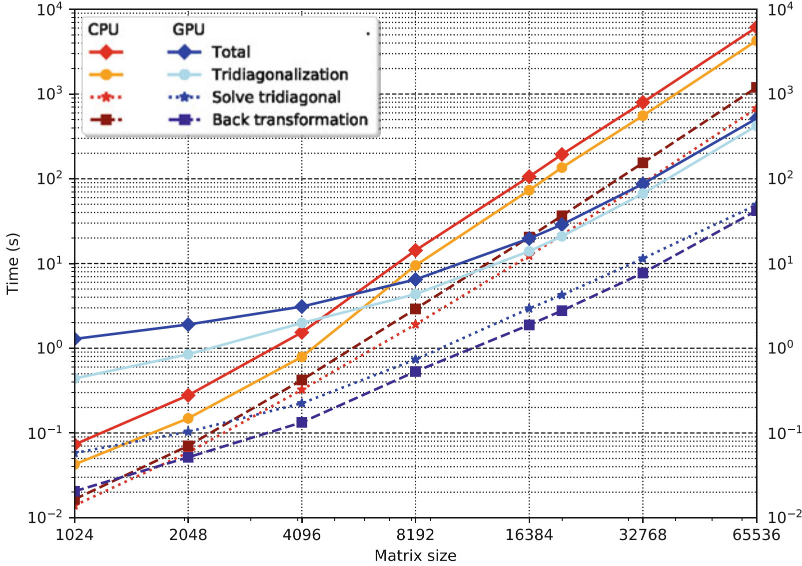


Fig. 2 Detailed comparison of individual parts of the ELPA 1 algorithm on the Minsky node (see in text) using CPU only and CPU and all 4 GPUs

the run times of the individual steps. We selected only two setups for this detailed study, the best performing CPU and GPU variants, i.e., the Minsky CPU and Minsky CPU + 4 GPUs versions.

In all cases, the (one-stage) ELPA 1 compute-time is dominated by the tridiagonalization step. This step contains large BLAS level 2 operations (matrix-vector multiplications), which can not be very efficiently implemented in neither BLAS (CPU) nor cuBLAS (GPU) libraries. Still, since most of the work done in both the tridiagonal solver and the back substitution is hidden in BLAS level 3 operations, which are particularly efficient on GPU, we can see, that the speedup of those functions is higher than the speed-up of the tridiagonalization step and thus the GPU version is even more limited by the BLAS level 2 dominated tridiagonalization.

The solution times for large matrices are listed in detail in Table 2. The first two cases are intended for a comparison between the two tested architectures, since the same matrix size is used, and only 2 GPUs are utilized for the Minsky system. The last case shows the largest possible (due to memory limitations) computed matrix with the full Minsky node (using all 4 GPUs). It is worth noticing, that for the back substitution step we get almost 30× speed-up. This is caused by the efficient implementation using BLAS level 3 operations only. Indeed, even the total speed-up of almost 12× is very good.

Table 2 Performance of individual stages of the ELPA 1 algorithm

Machine	2xIntel+2xK40m			2xPower8+**2x**P100			2xPower8+**4x**P100		
Mat. size	32768			32768			65536		
	CPU	CPU+GPU		CPU	CPU+GPU		CPU	CPU+GPU	
	t (s)	t (s)	s-up	t (s)	t (s)	s-up	t (s)	t (s)	s-up
Total	1424	396	**3.6**×	798	136	**5.9**×	6139	514	**11.9**×
Tridiag.	1108	333	**3.3**×	555	113	**4.9**×	4263	422	**10.1**×
Solve	117	24.1	**4.9**×	88.9	12.9	**6.9**×	678	49.2	**13.8**×3
Back s.	198	36.9	**5.4**×	154	10.3	**15**×	1198	42.1	**28.5**×

Speed up is with respect to CPU only version

7 Conclusions

We have presented a GPU implementation of the ELPA library. We have described the methods used to utilize the potential of GPU accelerators to speed-up large-scale eigenvalue and eigenvector computation, while preserving the ELPA scalability by keeping the block-cyclic distribution of matrices and efficient MPI-based communication patterns. With this GPU implementation, codes using ELPA can significantly reduce their time to solution.

We show, that the achieved speed-up depends on the matrix size and the hardware, but can reach up to 12× on the Minsky system for a large matrix. From the presented results the performance relationships for different hardware can be derived. We also show that the current bottleneck of the ELPA 1 algorithm lies in the tridiagonalization step, because of the necessary matrix-vector multiplications, which can not be implemented very efficiently on neither CPU nor GPU. However, since the other two steps, namely the solution of the tridiagonal system and the back substitution can be computed very efficiently on the GPU, this bottleneck is (relatively) even more severe in the GPU version.

Acknowledgements Part of this work is co-funded by BMBF grant 01IH15001 of the German Government.

References

1. T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krmer, B. Lang, H. Lederer, P.R. Willems, Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Comput. **37**, 783–794 (2011)
2. A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, H. Lederer, The ELPA library - scalable parallel eigenvalue solutions for electronic structure theory and computational science. J. Phys. Condens. Matter **26**, 213201 (2014)
3. ELPA Library, http://elpa.mpcdf.mpg.de
4. G.H. Golub, C.F.V. Loan, *Matrix Computations* (John Hopkins University Press, Baltimore, 2013)

5. ScaLAPACK - Scalable Linear Algebra PACKage, http://netlib.org/scalapack
6. Matrix Algebra on GPU and Multicore Architectures, http://icl.utk.edu/magma
7. CuBLAS Library, https://developer.nvidia.com/cublas
8. Multi-Process Service, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf