A note on the GPU acceleration of eigenvalue computations

K. Rupp[*,†], Ph. Tillet[**], B. F. Smith[*], T. Grasser[‡] and A. Jüngel[†]

[*]*Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, USA*
[†]*Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8–10, A-1040 Wien, Austria*
[**]*Institute of Electrical and Computer Engineering, National Chiao Tung University, Hsinchu City, Taiwan 300*
[‡]*Institute for Microelectronics, TU Wien, Gußhausstraße 27–29, A-1040 Wien, Austria*

Abstract. Eigenvalue computations for large sparse matrices such as the Lanczos method are commonly based on Krylov subspace techniques. One of the dominant operations in such algorithms are iterated computations of inner products with the same vector in order to preserve orthogonality of the Krylov basis. These operations can be accelerated by existing BLAS functionality using GPUs. However, this is not fully efficient due to unnecessary memory transfers. We present improved implementations in CUDA and OpenCL, which are now available in ViennaCL, PETSc and SLEPc, and demonstrate an up to two-fold performance gain over existing GPU vendor libraries.

Keywords: Linear Algebra, Eigenvalues, Krylov methods, GPU, CUDA, OpenCL, ViennaCL, PETSc, SLEPc
PACS: 02.10.Ud, 02.60.Dc

INTRODUCTION

Eigenvalue computations are at the heart of quantum mechanics: The fundamental stationary Schrödinger equation is naturally formulated as an eigenvalue problem. Frameworks built on top of the Schrödinger equation such as the density functional theory preserve the need for computing eigenvalues.

Depending on the respective setting, either the full eigenvalue problem for a dense system matrix A needs to be solved, or only certain eigenvalues (usually either the largest or the smallest) of a sparse matrix A are of interest. We also note that the general eigenvalue problem of finding z such that

$$Az = \lambda Bz \tag{1}$$

for eigenvalues λ and eigenvectors z can be reduced to the standard eigenvalue problem if B is positive definite and either real symmetric or complex Hermitian: Using the Cholesky factorization $B = LL^T$ (or $B = LL^H$ in the Hermitian case), one can rewrite (1) to

$$(L^{-1}AL^{-T})(L^T z) = \lambda L^T z \, , \tag{2}$$

hence denoting $C := L^{-1}AL^{-T}$ and $y = L^T z$ one observes that the eigenvalue problem $Cy = \lambda y$ is equivalent to (1) [1].

If A is dense, standard basic linear algebra subroutines (BLAS) supplemented by additional library functionality in e.g. LAPACK [2] can be used [3]. With the advent of accelerators, particularly graphics processing units (GPUs) for general purpose computations, the actual mapping of these standard linear algebra routines becomes challenging and is subject of active research [4, 5, 6]. In this work, however, we focus on the case of a sparse matrix A. Eigenvalue methods for sparse matrices predominantly, yet not exclusively, rely on orthogonal Krylov bases, which are commonly dealt with by using Gram-Schmidt methods or Householder reflections [7]. Even though Householder reflections provide better round-off properties, Gram-Schmidt methods are often preferred in practice because of the lower number of arithmetic operations and the higher amount of parallelism involved. Given an orthonormal basis $(v_1, \ldots, v_N)$ and a vector w not in the space spanned by the basis, the basis extension with respect to w is computed by

$$v_{N+1} \leftarrow w - \sum_{i=1}^{N} \langle w, v_i \rangle v_i \, , \qquad v_{N+1} \leftarrow v_{N+1} / \|v_{N+1}\| \tag{3}$$

where we refer to the parallel computation of the scalars $\langle w, v_i \rangle$ in (3) as *mdot* (multiple dot) operation. If A stems from a stencil discretization and the Krylov basis reaches sizes of more than, say, 50 vectors, the *mdot* operation becomes the dominating factor for the overall performance.

GPU IMPLEMENTATIONS

In the following we compare three possible implementations for the computation of multiple inner products with one common argument in parallel:

Ddot Assuming that repeated loads of w from global memory is cheap or ignoring the costs of memory loads, the first implementation is an iterated call of the function for dot product.

Dgemv In order to reuse the common vector w, the second implementation we consider is to copy the vectors $\{v_i\}_{i=1}^{N}$ into a matrix V and obtain the results $\langle w, v_i \rangle$ from the resulting vector of the matrix-vector multiplication $V \times w$. In the context of a Gram-Schmidt orthogonalization one could also augment V with additional vectors in each call, but this is not a viable approach for an actual software implementation because it requires to pass V along.

mdot The third implementation is an optimized custom implementation, avoiding most spurious memory loads. For portability reasons, both a CUDA and an OpenCL version were implemented. We implemented kernels `dot`, `dot2`, `dot3`, `dot4`, `dot8` for dealing with one, two, three, four, and eight vectors $\{v_i\}$, respectively. In the following we discuss `dot4` in more detail, other kernels follow the same pattern.
Thread i loads the entry of `w[i]` into a local variable `w_i`. Then, the scalars

```
alpha1 = w_i * v1[i]
alpha2 = w_i * v2[i]
alpha3 = w_i * v3[i]
alpha4 = w_i * v4[i]
```

are computed. If threads need to deal with more than a single entry, they continue with summing contributions into `alpha1`, `alpha2`, `alpha3`, and `alpha4`. A parallel reduction in shared on-chip memory for each thread group leads to partial results `beta1`, `beta2`, `beta3`, and `beta4` which contain the sum over all `alpha`-variables of each thread. These partial results are written back to a vector z in global memory and are then copied back to the host. Note that the size of the vector z equals the number of independent thread groups and is much smaller than the input vectors w and $\{v_i\}$. The host finally sums these partial results to obtain the results of $\langle w, v_1 \rangle$, $\langle w, v_2 \rangle$, $\langle w, v_3 \rangle$, and $\langle w, v_4 \rangle$.
If the number of vectors in the set $\{v_i\}$ does not exactly map to one of the *dotX*-kernels ($X \in 1,2,3,4,8$), then the largest suitable *dotX*-kernels are iterated until all inner products have been computed. For example, an *mdot* operation acting on 21 vectors calls `dot8` twice, `dot4` once, and `dot2` once. Hence, values from w may still be loaded multiple times, but the asymptotic overhead is only 12.5 percent. Kernels dealing with a higher number of vectors $\{v_i\}$ are possible, but the price to pay is two-fold: First, increased maintenance effort is required for the higher number of kernels. Second, the higher number of vectors leads to increased register and on-chip shared memory pressure in the kernels, ultimately either reducing the effective memory bandwidth obtained for the kernel or exceeding the on-chip resources available.

While the first two implementations are motivated by making use of the tuning effort put into vendor libraries, our third approach is entirely concerned about minimizing memory transfer. Even though we focus on GPUs here, such a reuse of data is also important for good performance on conventional processors. As long as the programming models, here CUDA and OpenCL, are able to load data near peak memory bandwidth, our third approach can also be justified from the software maintenance point of view.

BENCHMARK RESULTS

We compare our implementation of *mdot* with the two implementations discussed in the previous section. Our implementations are freely available through the GPU linear algebra package ViennaCL [8], the solver package PETSc [9, 10], and hence the eigenvalue package SLEPc [11]. The latter two have been reported to be in use for grid-based quantum computing just recently [12].

Our experiments were run on an NVIDIA GeForce GTX 470 using CUDA 5 platform libraries and an AMD Radeon HD7970 using clAmdBlas 1.10.321. Both systems are Linux-based, hence the latency overheads reported here may be slightly different on a Windows-based machine. All operations are carried out using double precision arithmetics.

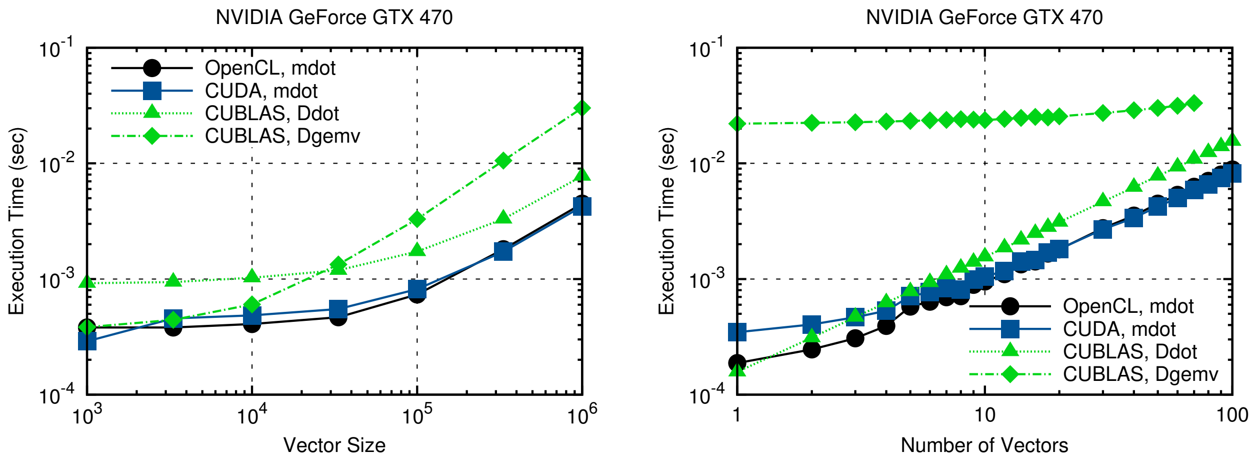

FIGURE 1. Comparison of execution times for inner products $\langle w, v_i \rangle$, $i = 1, \ldots, N$ using double precision arithmetics on NVIDIA hardware. Left: $N = 50$, varying vector size. Right: Vector size 10^6, varying N. A performance gain of up to a factor of two over CUBLAS (CUDA 5.0) is obtained.

Matrix setup times for the *Dgemv* implementations are included in our timings. Since all operations considered are entirely memory bandwidth limited, our results are directly transferable to workstation hardware (NVIDIA Tesla series, AMD FirePro series). We used 64 threads for each of the 256 thread groups for both platforms. In order to judge the expected performance on a broad range of machines, no autotuning was applied to the kernels. Conversely, target- and size-specific optimizations may further improve the performance by a few percent [6].

Both the CUDA and OpenCL implementation of *mdot* are depicted with the respective alternatives based on vendor-libraries for the NVIDIA GPU in Fig. 1. Comparing the performance for different vector sizes and a fixed number of 50 vectors $\{v_i\}$, the overhead of iterated kernel launches for *Ddot* becomes apparent at small vector sizes. The matrix-vector product implementation using *Dgemv* is faster than *Ddot* for small vector sizes, but is still unable to outperform our *mdot* implementation. It should be noted that for vector sizes below about 10^4, the PCI-Express communication overhead makes a purely CPU-based implementation the better choice over a GPU implementation anyway. For large vector sizes, *Dgemv* performs poorly and is by an order of magnitude slower than our *mdot* implementation. The *Ddot* function stays within a factor of two when compared to *mdot* at large problem sizes, which is expected as it needs to read almost twice as much data from global GPU memory. There are no significant differences in performance between our CUDA and OpenCL implementation of *mdot*. We assume that the slightly better performance of the OpenCL version is due to better device-specific optimization possibilities of the just-in-time OpenCL compiler.

For a fixed vector size 10^6, the comparison of performance for a varying number of vectors v_i on the right of Fig. 1 shows a large overhead of the *Dgemv* implementation. Moreover, due to the additional memory needed for the matrix, the GPU runs out of memory if more than 70 vectors $\{v_i\}$ are involved. Both of our *mdot* implementation outperforms the *Ddot*-based approach as soon as the number of vectors exceeds two and gradually increases to a factor of 1.9 for 100 vectors, which is almost the theoretically possible maximum speed-up. Again, there is no significant difference in performance for our CUDA and OpenCL implementations.

Execution times for the AMD GPU are compared in Fig. 2. Our *mdot* implementation is outperformed by the *Dgemv*-based approach for small vector sizes up to 10^4. As mentioned earlier, this regime is of little relevance in practice, hence we refrained from a separate optimization for this case. For larger vector sizes, however, the *Dgemv*-based implementation quickly shows a large overhead of more than an order of magnitude. Also, the slope of the benchmark curves when comparing different vector sizes suggests that the communication overhead is still significant even at a vector size of 10^6. Larger vector sizes, however, exceed the available GPU RAM limitations, because up to 101 of them need to reside there at the same time. This is again reflected by the missing data points for the *Dgemv*-implementation when using more than 80 vectors. The *Ddot*-based implementation shows the anticipated factor of slightly below two in execution time with respect to our *mdot* implementation. This factor is fairly independent of the number of vectors $\{v_i\}$, except for the case of a single dot product.

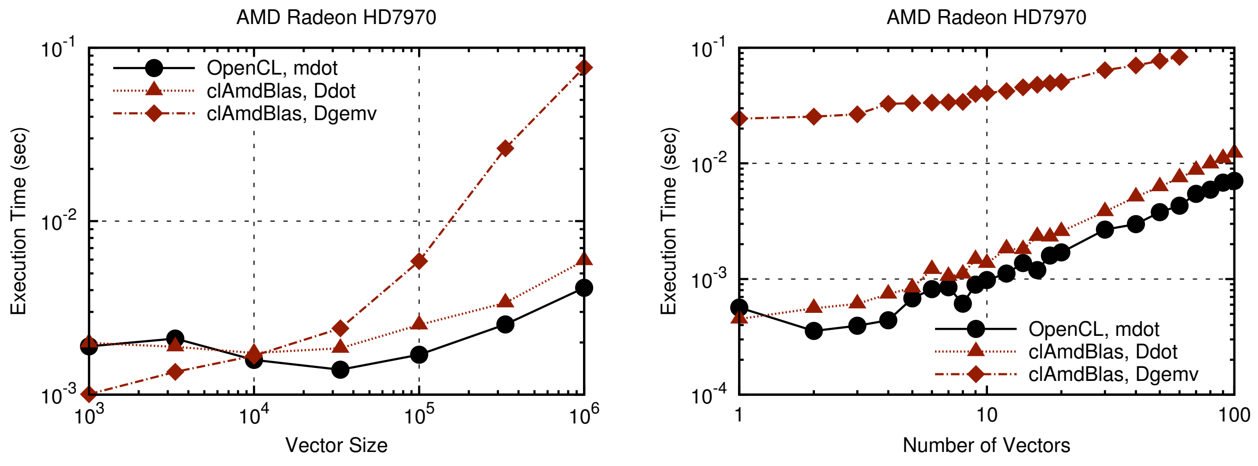

FIGURE 2. Comparison of execution times for inner products $\langle w, v_i \rangle$, $i = 1, \ldots, N$ using double precision arithmetics on AMD hardware. Left: $N = 50$, varying vector size. Right: Vector size 10^6, varying N. A performance gain of up to a factor of two over clAmdBlas 1.10.321 is obtained.

CONCLUSION

We discussed our implementation of the multiple inner products $\langle w, v_i \rangle$ with a common argument vector w for GPUs from major vendors. A performance benefit of a factor of almost two is obtained by packing multiple inner products into a single kernel and thus avoiding unnecessary multiple reads of the values in w. Our results also confirm that the leading hardware metric for the performance of vector operations on modern hardware is memory bandwidth, not floating point operations per second.

ACKNOWLEDGMENTS

This work has been partially supported by the Austrian Science Fund (FWF), grant P23598, and the FASTMath SciDAC Institute. K. Rupp wishes to thank Paul Mullowney and Jose E. Roman for quick performance feedback on early versions of the implementations.

REFERENCES

1. G. H. Golub, and C. F. Van Loan, *Matrix Computations (3rd ed.)*, Johns Hopkins University Press, 1996.
2. LAPACK - Linear Algebra PACKage (2013), `http://www.netlib.org/lapack/`.
3. A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics*, Texts in Applied Mathematics, Springer, 2007, ISBN 978-3-540-34658-6.
4. J. Kurzak, S. Tomov, and J. Dongarra, *IEEE Transactions on Parallel and Distributed Systems* (2012).
5. K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Implementing a Code Generator for Fast Matrix Multiplication in OpenCL on the GPU," in *6th IEEE International Symposium on Embedded Multicore SoCs (MCSoC-12), 2012*, 2012.
6. P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin, "Towards Performance-Portable, Scalable, and Convenient Linear Algebra," in *Proceedings of HotPar'13*, 2013, pp. 1–8.
7. D. S. Watkins, *The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007, 1 edn., ISBN 0898716411, 9780898716412.
8. Vienna Computing Library - ViennaCL (2013), `http://viennacl.sourceforge.net/`.
9. S. Balay, J. Brown, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. Curfman-McInnes, B. Smith, and H. Zhang, PETSc Web page (2013), `http://www.mcs.anl.gov/petsc`.
10. S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman-McInnes, B. Smith, and H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory (2013).
11. C. Campos, J. E. Roman, E. Romero, and A. Tomas, SLEPc users manual, Tech. Rep. DSIC-II/24/02 - Revision 3.3, D. Sistemes Informàtics i Computació, Universitat Politècnica de València (2012).
12. T. D. Young, E. Romero, and J. E. Roman, *Computer Physics Communications* **184**, 66 (2013).