Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs

Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
cbrow216@vols.utk.edu, {ahmad,tomov,dongarra}@icl.utk.edu

***Abstract*—Dense linear algebra (DLA) has historically been in the vanguard of software that must be adapted first to hardware changes. This is because DLA is both critical to the accuracy and performance of so many different types of applications, and because they have proved to be outstanding vehicles for finding and implementing solutions to the problems that novel architectures pose. Therefore, in this paper we investigate the portability of the MAGMA DLA library to the latest AMD GPUs. We use auto tools to convert the CUDA code in MAGMA to the Heterogeneous-Computing Interface for Portability (HIP) language. MAGMA provides LAPACK for GPUs and benchmarks for fundamental DLA routines ranging from BLAS to dense factorizations, linear systems and eigen-problem solvers. We port these routines to HIP and quantify currently achievable performance through the MAGMA benchmarks for the main workload algorithms on MI25 and MI50 AMD GPUs. Comparison with performance roofline models and theoretical expectations are used to identify current limitations and directions for future improvements.**

***Index Terms*—Numerical Linear Algebra, HPC, GPU Computing, AMD GPUs, HIP Runtime, Portability**

I. Introduction and Related Work

GPUs are quickly evolving. Traditionally used for graphics rendering, requiring high data throughput, GPUs excelled in delivering very high bandwidth that is fundamentally needed in computations well beyond graphics. The latest A100 GPU from Nvidia, for example, features an unprecedented theoretical bandwidth of 1.6 TB/s. Along with bandwidth, compute capabilities have also grown, reaching an outstanding 19.5 Tflop/s peak in FP64 arithmetic for a single A100 GPU. A well developed memory hierarchy has enabled data reuse that alleviates the gap between data transfer rates and compute capabilities, allowing many applications, e.g., based on matrix-matrix products, to reach performances close to the peak. By covering the entire spectrum from bandwidth-limited to compute-bound computations, GPUs have become a commodity hardware that is widely used in HPC. Currently, more than 20% of the Top500 supercomputers use GPUs [1], and future plans point to a trend that GPUs are here to stay. The Summit and Sierra supercomputers at Oak Ridge and Livermore Lab respectively, are the top supercomputers in USA and both feature Nvidia GPUs. The first two Exascale supercomputers, Frontier and Aurora at Oak Ridge and Argonne Labs respectively, are slated to appear by the end of 2021 and be based on GPUs from AMD and Intel, respectively [2]. Another boost in the use of GPUs is AI and ML, which have influenced current GPU designs to boost low-precision calculations. For example, FP16 calculations are hardware accelerated on the latest Nvidia GPUs using so called Tensor Cores, and are able to reach 624 TFlop/s. These developments have put pressure on developers to not only port their DLA on different GPUs but also to support and update for the latest GPU changes, e.g., most notably now with new capabilities enabling the use of low-precision arithmetic in HPC.

The quickly evolving GPUs have posted challenges on how to program and maintain functional and performance-portable codes across different vendors and generations of GPUs. There is a lack of well adopted standards on how to code them. Indeed, while there are standards like OpenCL, OpenMP, OpenACC, and SYCL, vendors prefer to push quickly with proprietary innovations, e.g., as in CUDA from Nvidia, that can percolate into the standards later. Software solutions, like RAJA [3] or Kokkos [4] for example, provide "wrappers" on top of the plethora of vendor solutions and standards in an effort to provide unified programming environment, but in general they are not a substitution for HPC libraries, and also must be updated and supported on a case-by-case base.

There has been a lot of work on specific algorithms, concentrated on particular GPUs, and programming model. These range from matrix-vector [5]–[7], matrix-matrix [8]–[11], and other BLAS operations to higher level routines, like linear system, eigen-problems and SVD solvers, etc. In this paper, we are interested to port an entire library that contains all these DLA building blocks; benchmark the performance obtained; and assess the library portability accros GPUs. We do this for the MAGMA library [12], originally designed and implemented to support Nvidia GPUs. MAGMA has been ported before to OpenCL [13] and Intel Xeon Phi [14]. Here we assess its portability to the AMD's Radeon Open Compute platforM (ROCm), using HIP [15].

Portability of DLA libraries, like LAPACK [16] and ScaLAPACK [17], has been achieved through the portability of the BLAS standard [18]. The libraries are coded using BLAS, and vendors or community provides high-performance BLAS implementations for different architectures. Thus, BLAS is the abstraction allowing these DLA libraries not to be developed from scratch, while still being functionally and performance-portable to new architectures. This is the design in MAGMA for its high-level routines. However, MAGMA also provides

BLAS and auxiliary routines, e.g., to supplement vendor BLAS and discover better performing building blocks, when needed, and these require low-level programing. Fundamental routines like matrix-matrix products (e.g., GEMM for general matrices, etc.) have always been of high interest as many algorithms use them, or modify them, e.g., fusing them with other operations, to derive desired functionalities. Therefore, it is important to have these in source code as well vs. just a typical assembly implementation that can get close to peak performance by bypassing compiler limitations.

II. The Software Architecture of hipMAGMA

A. Overview

MAGMA has grown significantly over the years. While LAPACK has approximately 320 functions, amounting to about 1,300 routines when counting all precisions, MAGMA provides most of these routines in LAPACK compliant interface where both input and output matrices are on the CPU memory, as well as in GPU interface, where both input and outpus are on the GPU memory. Many of the routines also have different versions, there are a number of main routines for multi-GPUs [19], out-of-GPU-memory algorithms [20], [21], and mixed-precision solvers [19], [22], [23]. In addition, MAGMA also has batched routines [24], [25], and a sparse linear algebra component [26]. BLAS itself, on the other hand, has about 57 functions, or about 170 routines when counting the precisions. This has created an enormous code base that would be challenging to support and port across different GPUs and programming models, unless properly designed and maintained with portability in mind.

Figure 1 shows the MAGMA software stack. MAGMA provides LAPACK interfaces, which are implemented using BLAS. The BLAS implementations for specific hardware reside under the 'BLAS' layer. Thus, BLAS provides an abstraction layer that shields the top-level MAGMA routines from platform-specific implementation details. The BLAS interface is either implemented via the vendor libraries, e.g., cuBLAS or hipBLAS, or by a custom (MAGMA) implementation. These are the routines written in CUDA that we aim to port using auto tools. The LAPACK layer is backend-independent and remains unchanged (by design).

B. Translation & Generation

The HIP backend is automaticly generated using a modified version of the 'hipify-perl' script provided by AMD [15] to translate individual files from CUDA code (i.e., `.cu` files) into HIP code (MAGMA uses the `.hip.cpp` extension to distinguish from normal, non-HIP C++ code). Almost all functions and data types in CUDA have a direct one-to-one counterpart in HIP [15], and so text substitution works to translate most CUDA code into compilable HIP code.

We found that this porting methodology worked very well, providing both functional and performance portable code. A few of the issues enountered came from:

- Deprecated CUDA and NVIDIA library APIs (which typically do not exist at all in their HIP counterparts). These must be either removed or updated to use a currently supported API in CUDA.
- Oddities arising from the CUDA launch syntax (i.e. `kernel<<<blk, thd>>>(A, B, ...)`) and expressions for `blk` or `thd`; Expressions must be wrapped in more parentheses than neccessary, e.g., the CUDA code should read: `kernel<<<(blk), (thd)>>>(A, B, ...)`;
- CUDA's method of dynamic shared memory (**`extern`** `__shared__ type_t ptr;`) allows for some declarations of `ptr` outside of a device function, residing in the global scope. In HIP, all such declarations must be made inside of and per each device function which uses dynamic shared memory.
- Unimplemented functionality in HIP libraries and the compiler itself. Most problems or bugs that have been found while converting MAGMA have already been fixed in a ROCm release, so only temporary adjustments must be made in the meantime. For example, MAGMA includes macro definitions for unimplemented functions, so they may compile and run, but any program that uses them will indicate the unreliability of the results.

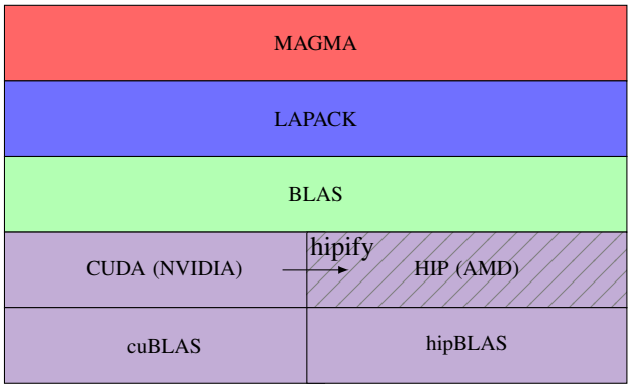

Fig. 1. Software Architecture of MAGMA. Top layers are the most abstracted, while lower-level layers are more hardware specific.

These issues were corrected by amending the base code to include blocks of code in **`#ifdef`** guards. So, in some occurrences, HIP code is included in the base code, but is never compiled on an NVIDIA platform due to the macros defined. From there, a direct translation is performed, copying directories of source code and then translating each file into valid HIP code via the tools mentioned. The build system must then be directed to compile the new, generated files instead of original sources. Other than that, the new MAGMA build system was largely unchanged.

III. BLAS Design for AMD GPUs

MAGMA uses high level interfaces that abstract the vendor supplied BLAS and LAPACK routines. Since MAGMA, in general, develops hybrid CPU-GPU LAPACK algorithms, it requires standard BLAS and LAPACK for executing workloads on the CPU (e.g., using MKL or OpenBLAS as backends). A similar mode of operation exists on the GPU side, using either cuBLAS or hipBLAS as backends. However,

MAGMA also provides its own set of BLAS routines, which can replace specific vendor routines that lack optimization or are missing. For NVIDIA GPUs, most of the BLAS required by MAGMA exists in cuBLAS, and achieves very good performance. However, few of these routines are slower than the ones developed in MAGMA, e.g., the batch triangular solve. For AMD GPUs, hipBLAS is still actively developed, and some of its BLAS routines are either missing or trail the corresponding MAGMA BLAS routines in performance.

This section highlights some of our BLAS designs and optimizations for AMD GPUs in order to complement hipBLAS. Our strategy takes advantage of the well optimized routines in hipBLAS, e.g., GEMM, to provide other BLAS functionalities that leverage the GEMM performance. Some BLAS routines, like matrix-vector products, cannot take advantage of the GEMM performance, and therefore are developed based on our experience with the CUDA backend.

A. Matrix Multiplication (GEMM)

The GEMM kernel is arguably the most important DLA operation. Not surprisingly, it has been the focus of many research efforts [9], [10]. Vendor libraries usually provide highly optimized GEMM implementations that are written in a low-level language [27] in order to overcome some limitations imposed by the compiler and the hardware scheduler. As an example, assembly implementations [28], [29] are available today in cuBLAS, with the ability to achieve a performance that is very close to the GPU's theoretical peak.

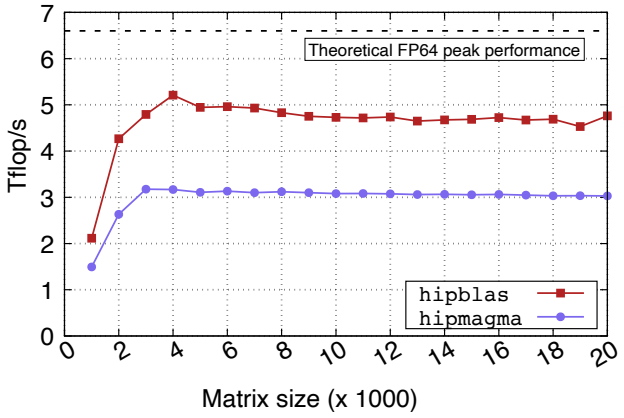

Fig. 2. Performance of the GEMM operation in double precision on the Mi50 GPU. Results are shown for square sizes using ROCm 3.5.

A similar behavior is observed on AMD GPUs. The hipBLAS library provides a high performance GEMM implementation that is written in a low level language that skips the limitations imposed by the `hipcc` compiler. Therefore, the hipBLAS GEMM routine outperforms the MAGMA kernel, which is written using the HIP kernel language based on the design by Nath et al. [10]. Figure 2 shows the DGEMM performance on an Mi50 GPU. The hipBLAS routine is about 55% faster than MAGMA. While this illustrates functional and performance portability, as the performance obtained is quite reasonable, it is important analyze and put these results in perspective with respect to the theoretical peak performance and the corresponding performance on the CUDA backend. While hipBLAS DGEMM is asymptotically at 72% of the theoretical peak of the Mi50 GPU, the cuBLAS GEMM routine is about 93% of the peak on the V100 GPU. More importantly, the MAGMA DGEMM kernel scores 63% of the hipBLAS performance, while it usually scores 82% of the cuBLAS performance on the V100 GPU. That means (1) the achievable performance of compute-bound BLAS on AMD GPUs needs to be improved, even for low-level implementations, and (2) there are more overhead imposed by the HIP compiler/runtime than the CUDA compiler/runtime.

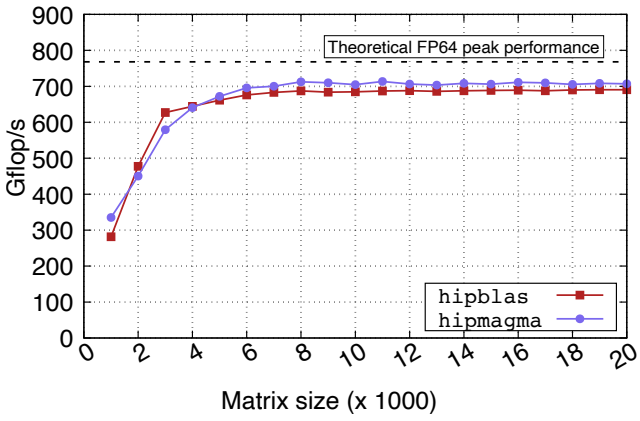

Fig. 3. Performance of the GEMM operation in double precision on the Mi25 GPU. Results are shown for square sizes using ROCm 3.5.

The MAGMA GEMM has been incorporated in cuBLAS for GPUs when assembly was not yet needed in order to get close to peak performance. The design features a few levels of blocking for the different memory hierarchies of the GPUs and is parametrized for subsequent autotuning [25], [30], [31]. Theoretically the design allows for communication and computations to be overlapped and the inner-most loops are blocked to reduce data traffic to a level where the kernel is compute-bound, and thus achieve close to peak performance. This is illustrated for example on Figure 2 for the Mi25 GPU. Here MAGMA HIP version actually outperforms the assebly kernel in hipBLAS and reaches 94% of the theoretical peak. These results, as well as discussions with AMD, lead us to conlcude that:

- The HIP compiler (`hipcc`) can be further improved in regards to register use, e.g., spilling or promoting local 2D arrays to registers. MAGMA uses register blocking to reduce data trafic and the compiler is crucial in enabling this to happen. Since the gap between bandwidth and compute peak on the Mi50 is much higher than the one for the Mi25, the difference between code in assembly and compiler-dependent HIP is more prominent.
- We intentionally did not find the optimal configuration sizes for MAGMA's kernels in order to illustrate the performance of the direct auto port. Changing different

tuning parameters yields better performance, e.g., up to 3.5 Tflop/s on the Mi50. Further improvements will come from the compiler and we plan to research this further at a later ROCm release.

B. Symmetric Compute-bound BLAS

Unlike GEMM, some important BLAS routines involve symmetric matrices as inputs or outputs. Such routines are important for high level algorithms such as the Cholesky factorization or the symmetric eigensolvers. Routines like the symmetric rank-k and rank-2k updates (SYRK and SYR2K, respectively) compute only the lower or the upper triangular part of the output symmetric matrix. The SYRK operation is defined as $(C = \alpha A \times A^T + \beta C)$ or $(C = \alpha A^T \times A + \beta C)$. The SYR2K update is defined as $(C = \alpha A \times B^T + \alpha B \times A^T + \beta C)$ or $(C = \alpha A^T \times B + \alpha B^T \times A + \beta C)$. Both α and β are scalars. A and B are input matrices of size $n \times k$ or $k \times n$, depending on the transposition of the operation.

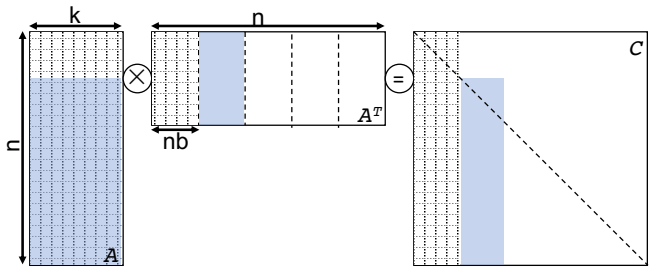

Fig. 4. Symmetric Rank-k Update using GEMM

A SYRK or SYR2K kernel can be written as a variation of the MAGMA GEMM kernel. However, based on the results of Figure 2, the performance of upper-bound for such kernels would be at 3 Tflop/s. This is why we focus on a more efficient approach that takes advantage of the hipBLAS GEMM performance. Figure 4 shows how the non-transposed SYRK operation can be implemented using a a sequence of GEMM calls. In order to update the lower triangular part of C using $A \times A^T$, the matrix A^T is subdivided into block columns of size `nb`. A sequence of GEMM calls then multiplies a submatrix of A with the corresponding block-column of A^T. The submatrix of A shrinks by `nb` from the top at each call, which avoids computing the upper off-diagonal blocks of C. The drawback of using GEMM appears in performing extra computations, which overwrite the upper triangular part of the diagonal blocks of C. However, this is a small tradeoff that leads to a high performance. The SYR2K operation can be implemented using two calls to a customized SYRK that accepts two different input matrices A and B instead of one. The customized SYRK routine is used to implement the standard SYRK operation, as well as the standard SYR2K operation.

Figure 5 shows the performance of the DSYRK operation. Thanks to a GEMM-based implementation, the DSYRK routines in MAGMA performs half the operation count of GEMM

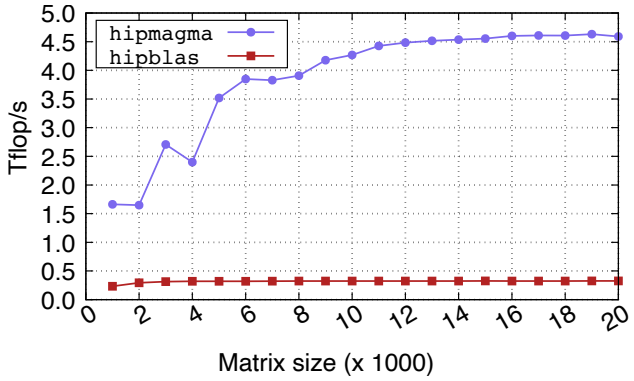

Fig. 5. Performance of the SYRK operation in double precision on the Mi50 GPU. Results are shown for square sizes using ROCm 3.5.

in half the time, which leads to nearly the same FLOP rate. On the other hand, the DSYRK in hipBLAS is $14\times$ slower, which is one of the motivations to add symmetric BLAS to hipMAGMA. The performance of the DSYR2K routine is almost identical to Figure 5, and is, therefore, not shown.

Another important BLAS operation is the symmetric matrix-multiply (SYMM), which is defined as $C = \alpha A \times B + \beta C$, or $C = \alpha B \times A + \beta C$. The input matrix A is symmetric, while B and C are assumed to be general matrices. Unfortunately, there is no straightforward way to utilize GEMM out of the box in the SYMM operation, since the symmetric matrix A is now an input (like SYRK and SYR2K, which accept general matrices and produce a symmetric output). This is why we designed the SYMM kernel similarly to the GEMM, except for the way the matrix A is read. Figure 6 shows the performance of the DSYMM kernel against the corresponding routine from the Intel MKL library, since hipBLAS does not yet provide a SYMM routine. The DSYMM performance is within 70% of the MAGMA GEMM kernel. This is because of a similar design that takes care of the symmetry of A. The asymptotic performance is about 45% faster than the MKL routine.

C. Triangular Matrix Multiplication (TRMM)

This routine is important for algorithms like the QR factorization and the symmetric eigensolvers. The TRMM operation is an in-place update, as per the standard BLAS definition. It computes product $B \leftarrow A \times B$ or $B \leftarrow B \times A$ for a triangular matrix A and a general matrix B. The in-place update imposes certain challenges on a parallel implementation. For example, GEMM is an out-of-place update that can be implemented as a single kernel call with parallel independent thread-blocks. This is not a suitable strategy for the TRMM operation.

Instead, we adopt a recursive implementation [32] that takes advantage of the hipBLAS GEMM routine. We begin by a "small TRMM" kernel, which works only on small triangular matrices that fit in the GPU shared memory. Independent thread-blocks read the small A matrix in shared memory, and perform the in-place update for independent sublocks of the B matrix. These sub-blocks are also cached in registers to be

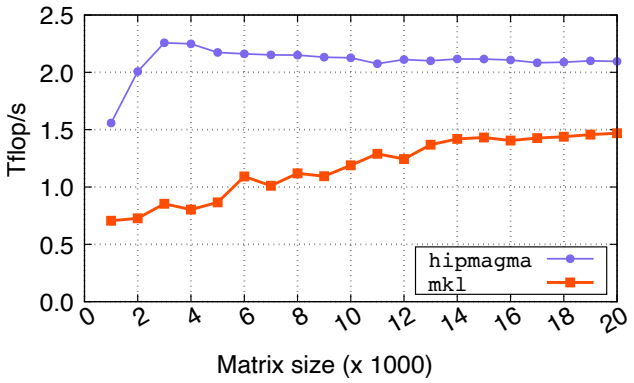

Fig. 6. Performance of the SYMM operation in double precision on the Mi50 GPU. Results are shown for square sizes using ROCm 3.5 against the MKL library. The CPU is an Intel Skylake processor (2×18-core Intel Xeon Gold 6140, running at 2.3 GHz)

updated in-place. This kernel is invoked only if the size of A is $\leq$ nb, which is a tuning parameter.

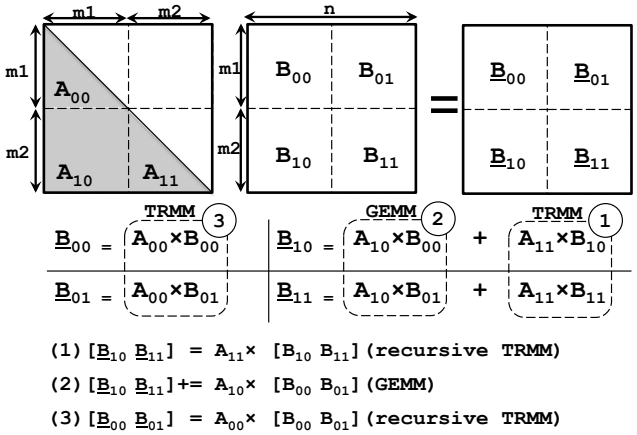

Fig. 7. The recursive TRMM design.

The recursive implementation is shown in Figure 7. It subdivides A as shown in the figure until the size of the triangular submatrix is $\leq$ nb, where the small TRMM kernel is called. The high-level TRMM begins by calling itself recursively with respect to A_{11}, and updates B_{1x} in-place (step 1). Then it calls the hipBLAS GEMM routine to compute the remaining portion for B_{1x} (step 2). The final step is to invoke the recursive TRMM routine with respect to A_{00} and update B_{0x} in-place. Note that these three steps must be performed in the specified order.

Figure 8 shows the performance of the DTRMM routine. The figure shows a huge asymptotic speedup that exceeds $60\times$ in favor of hipMAGMA. Our profiling of the hipBLAS TRMM kernel shows a sequence of calls to matrix-vector product kernels, which are memory bound. This explains the huge performance gap with respect to hipMAGMA, which leverages the high performance of the DGEMM kernel.

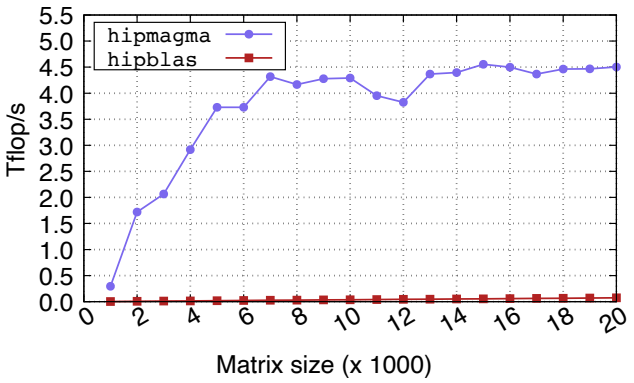

Fig. 8. Performance of the TRMM operation in double precision on the Mi50 GPU. Results are shown for square sizes using ROCm 3.5

D. Memory-bound BLAS

Matrix vector product (GEMV) and its symmetric variant (SYMV) are perhaps the most important memory-bound kernels in BLAS. The latter is a crucial component in the tridiagonal reduction stage of the symmetric eigensolver (DSYEVD), which we highlight later in this paper. Unlike compute-bound codes which provide a reasonable performance but seem to relatively underperform (e.g., the MAGMA GEMM and HEMM kernels in Figures 2 and 6, respectively), the hipified GEMV kernel achieves close to peak performance. As shown in Figure 9, the MAGMA kernel reaches up to 187 Gflop/s, which translates to achieving 748 GB/s of memory bandwidth. This is about 89% of the peak memory bandwidth scored by a STREAM benchmark ($\approx$ 840 GB/s). The more complicated SYMV kernel uses the symmetry to reduce data trafic in half (vs. GEMV), making it about $5\times$ faster than the hipBLAS kernel (Figure 10), which seems to underperform. The effective asymptotic bandwidth of the kernel is 516 GB/s, which is 61% of the sustained peak bandwidth. To put these numbers in more perspective, the same GEMV and SYMV implementations reach about 97% and 89.5% of the peak bandwidth on the V100 GPU, respectively.

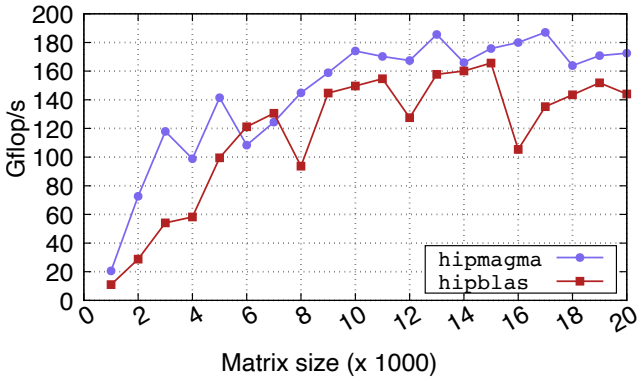

Fig. 9. Performance of the GEMV operation in double precision on the Mi50 GPU. Results are shown for square sizes using ROCm 3.5

So far, the memory-bound hipified kernels (e.g., GEMV) perform close to peak. The other, compute-bound kernels shown here (e.g., GEMM, SYMM), give reasonable performance but seem to relatively underperform. Among the four kernels discussed, the GEMV implementation is the only one that does not require register blocking. The rest require caching blocks of the matrix. SYMV transposes the cached off-diagonal blocks of the matrix to account for the untouched triangular part of the matrix. Kernels like GEMM and SYMM cache two blocks of A and B at a time in shared memory, where their product is accumulated to a block of C in registers.

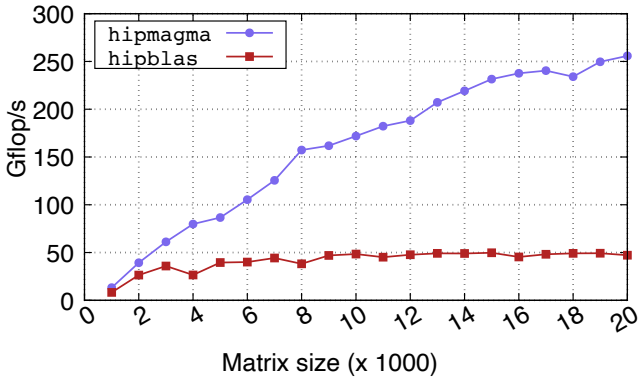

Fig. 10. Performance of the SYMV operation in double precision on the Mi50 GPU. Results are shown for square sizes using ROCm 3.5

IV. Impact on High-level Algorithms

Here we illustrate the portability of the high-level MAGMA routines, which are independent of the BLAS backend, and therefore remain unchanged in the port. Still, we show the impact of the optimized BLAS routines, and in this case we highlight the symmetric eigenvalue problem. The MAGMA implementation is hybrid CPU-GPU that uses the GPU to first reduce the input matrix to a tridiagonal form. Next, the tridiagonal matrix is sent to the CPU to find the eigenvalues. The computation of the eigenvectors, if required, is done of the GPU, since it mostly consists of level-3 BLAS routines.

Figures 11 and 12 show the impact of three BLAS routines (SYMV, SYR2K, and TRMM) on the performance of the DSYEVD solver. The solver also requires the SYMM routines, but since hipBLAS does not provide one, we always use MAGMA as the provider of this routine. In other words, any difference in performance will be due to the improvement in the SYMV, SYR2K, and TRMM kernels.

If no eigenvectors are required (Figure 11), then the `magmablas` backend provides a substantial speedup over the hipblas backend. In fact, the `magmablas` backend reduces the time-to-solution by factors between 13% to 73%. If the eigenvectors are required, the speedup is even more. Since the back transformations required to compute the eigenvectors rely on the TRMM routine, the time-to-solution is reduced by at least 58%, and goes up to 75%. This performance level is very reasonable.

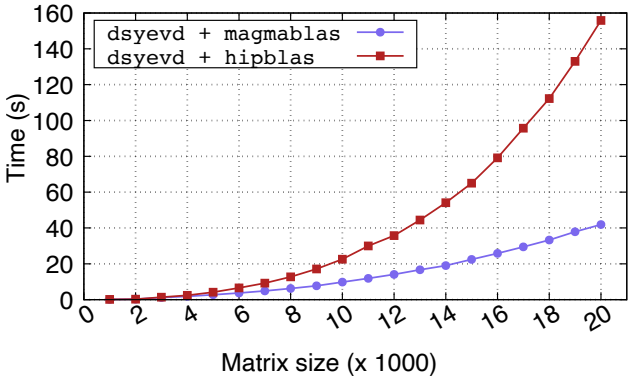

Fig. 11. Performance of the single stage symmetric eigensolver in double precision (DSYEVD). Results are shown for solving the eigenvalues only using the Mi50 GPU and ROCm 3.5

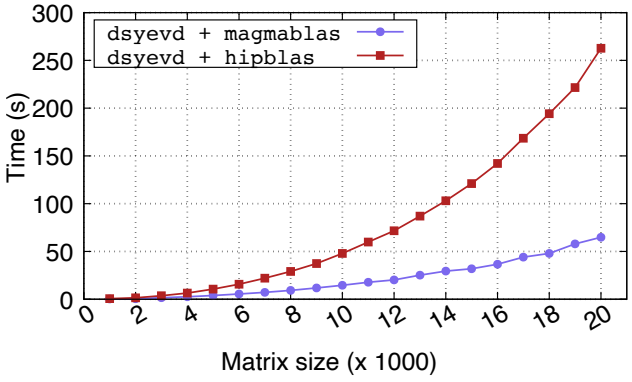

Fig. 12. Performance of the single stage symmetric eigensolver in double precision (DSYEVD). Results are shown for solving the eigenvalues and the eigenvectors using the Mi50 GPU and ROCm 3.5

V. Conclusion and Future Work

We investigated the portability of high-performance DLA to AMD GPUs using the HIP programming model. By allowing MAGMA to use different backends of BLAS routines and auto-source translation tools, we were able to easily provide an almost complete functional as well as performance portable port (of about 2,000 routines) to AMD GPUs. The developments are open source and are currently available though the hipMAGMA v1.0 release [33]. However, we also identified a few areas that can be further improved. The HIP framework is under continuous improvements, and we expect future ROCm version will further improve both BLAS and the HIP compiler. Thanks to a flexible backend switching in MAGMA, we are able to selectively exclude the currently slow BLAS routines and replace them with MAGMA's own BLAS kernels. The improved BLAS has a significant impact on high-level algorithms, like the dense symmetric eigensolver that we choose to highlight, whose time-to-solution is reduced by up to 75%. Future directions include continuous improvement of other high-level algorithms, and performing comprehensive tuning sweeps for the hipified kernels to achieve their best performances across AMD GPUs.

ACKNOWLEDGEMENT

This research was supported by AMD and the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations (the Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

REFERENCES

[1] "TOP500," https://www.top500.org/.

[2] T. Trader, "Cray, AMD to Extend DOEs Exascale Frontier," https://www.hpcwire.com/2019/05/07/cray-amd-exascale-frontier-at-oak-ridge/.

[3] "RAJA Performance Portability Layer," https://github.com/LLNL/RAJA.

[4] H. Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, Dec. 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.07.003

[5] H. H. B. Sørensen, "High-performance matrix-vector multiplication on the gpu," in *Euro-Par 2011: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 377–386.

[6] R. Nath, S. Tomov, T. Dong, and J. J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on GPUs," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, 2011, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063392

[7] A. Abdelfattah, D. Keyes, and H. Ltaief, "Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators," *ACM Trans. Math. Softw.*, vol. 42, no. 3, May 2016. [Online]. Available: https://doi.org/10.1145/2818311

[8] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS 04. New York, NY, USA: Association for Computing Machinery, 2004, p. 133137. [Online]. Available: https://doi.org/10.1145/1058129.1058148

[9] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, 2008, p. 31. [Online]. Available: http://doi.acm.org/10.1145/1413370.1413402

[10] R. Nath, S. Tomov, and J. Dongarra, "An Improved Magma Gemm For Fermi Graphics Processing Units," *The International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010. [Online]. Available: https://doi.org/10.1177/1094342010385729

[11] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. J. Dongarra, "High-Performance Matrix-Matrix Multiplications of Very Small Matrices," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 659–671. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_48

[12] S. Tomov, J. J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010. [Online]. Available: https://doi.org/10.1016/j.parco.2009.12.005

[13] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming," *Parallel Comput.*, vol. 38, no. 8, pp. 391–407, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.002

[14] A. Haidar, J. Dongarra, K. Kabir, M. Gates, P. Luszczek, S. Tomov, and Y. Jia, "Hpc programming on intel many-integrated-core hardware with magma port to xeon phi," *Scientific Programming*, vol. 23, 01-2015 2015.

[15] AMD, *AMD ROCm Platform*, 2020. [Online]. Available: https://rocmdocs.amd.com/en/latest/index.html

[16] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. D. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, Pennsylvania, USA: SIAM, 1999.

[17] L. S. Blackford, J. Choi, A. Cleary, E. DAzeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, and J. J. Dongarra, *ScaLAPACK Users Guide*. USA: Society for Industrial and Applied Mathematics, 1997.

[18] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[19] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, April 19-23 2010, pp. 1–8, DOI: 10.1109/IPDPSW.2010.5470941.

[20] I. Yamazaki, S. Tomov, and J. Dongarra, "One-sided dense matrix factorizations on a multicore with multiple gpu accelerators," in *in Proc. of the 2012 International Conference on Computational Science*, 2012.

[21] A. Haidar, K. Kabir, D. Fayad, S. Tomov, and J. Dongarra, "Out of memory svd solver for big data," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[22] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-precision Iterative Refinement Solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 47:1–47:11. [Online]. Available: https://doi.org/10.1109/SC.2018.00050

[23] A. Abdelfattah, S. Tomov, and J. Dongarra, "Investigating the benefit of fp16-enabled mixed-precision solvers for symmetric positive definite matrices using gpus," in *Computational Science – ICCS 2020*. Cham: Springer International Publishing, 2020, pp. 237–250.

[24] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon, "A Proposed API for Batched Basic Linear Algebra Subprograms," Manchester Institute for Mathematical Sciences, Tech. Rep., April 2016, [MIMS Preprint]. [Online]. Available: http://eprints.maths.manchester.ac.uk/id/eprint/2464

[25] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for gpus," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[26] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, and J. Dongarra, "Optimizing krylov subspace solvers on graphics processing units," in *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*, IEEE. Phoenix, AZ: IEEE, 05-2014 2014.

[27] G. Tan, L. Li, S. Triechle, E. H. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, 2011, pp. 35:1–35:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063431

[28] J. Lai and A. Seznec, "Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494986

[29] S. Gray, "A full walk through of the SGEMM implementation," https://github.com/NervanaSystems/maxas/wiki/SGEMM, 2015.

[30] Y. Li, J. Dongarra, and S. Tomov, "A Note on Auto-tuning GEMM for GPUs," in *Computational Science – ICCS 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 884–892.

[31] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM Kernels for the Fermi GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, Nov 2012.

[32] A. Charara, H. Ltaief, and D. E. Keyes, "Redesigning Triangular Dense Matrix Computations on GPUs," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 477–489.

[33] C. Brown, A. Abdelfattah, S. Tomov, and J. Dongarra, "hipMAGMA v1.0," Mar. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3908549