Batch QR Factorization on GPUs: Design, Optimization, and Tuning

Ahmad Abdelfattah[1](✉), Stan Tomov[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, USA
{ahmad,tomov,dongarra}@icl.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, USA
[3] University of Manchester, Manchester, UK

Abstract. QR factorization of dense matrices is a ubiquitous tool in high performance computing (HPC). From solving linear systems and least squares problems to eigenvalue problems, and singular value decompositions, the impact of a high performance QR factorization is fundamental to computer simulations and many applications. More importantly, the QR factorization on a batch of relatively small matrices has acquired a lot of attention in sparse direct solvers and low-rank approximations for Hierarchical matrices. To address this interest and demand, we developed and present a high performance batch QR factorization for Graphics Processing Units (GPUs). We present a multi-level blocking strategy that adjusts various algorithmic designs to the size of the input matrices. We also show that following the LAPACK QR design convention, while still useful, is significantly outperformed by unconventional code structures that increase data reuse. The performance results show multi-fold speedups against the state of the art libraries on the latest GPU architectures from both NVIDIA and AMD.

Keywords: Batch linear algebra · QR factorization · GPU computing

1 Introduction and Related Work

In the context of dense linear algebra, a batch routine performs a standard linear algebra algorithm on a batch of relatively small matrices. This kind of workload is quite different from operating on one large matrix. Many software packages, from both the industry and the research community, have been serving the latter form of workloads for many years. Examples include LAPACK [1], PLASMA [13], MAGMA [12], BLIS [19], and Intel's MKL [14]. Batch workloads, however, are relatively recent, and gained a lot of attention in many scientific communities. Applications include quantum chemistry [8], sparse direct solvers [21], astrophysics [16], and signal processing [6]. Vendor software libraries such as Intel's MKL [14], NVIDIA's cuBLAS [17], and AMD's hipBLAS [5] now provide many batch routines for several BLAS and LAPACK operations.

Batch routines often require a different mindset for performance optimization, especially on GPUs. Since we are dealing with small matrices, it is crucial

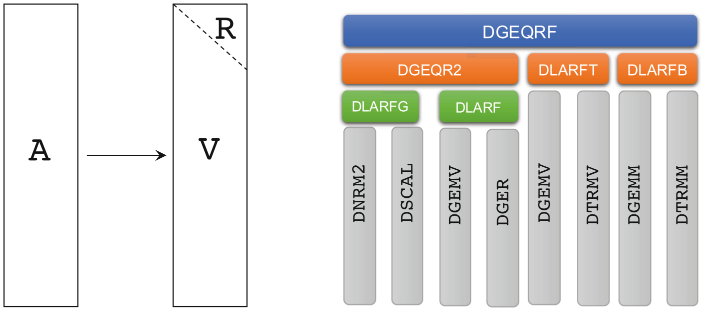
D. Groen et al. (Eds.): ICCS 2022, LNCS 13350, pp. 60–74, 2022.
https://doi.org/10.1007/978-3-031-08751-6_5

to save as much memory traffic as possible. As an example, for very small matrices that fit in the register file of the GPU, fully unrolled and unblocked kernels can achieve a performance that is superior to any other approach [3]. For relatively larger matrices, however, different assumptions must be made in order to maintain a high performance across the size spectrum.

In this paper, we take the batch QR factorization as a case study for optimization on GPUs. We show that there is not a single design strategy that can serve all sizes efficiently. Each design strategy assumes a number of building blocks (e.g., GPU kernels) of the factorization, which might differ from the conventional LAPACK structure. This work is considered an improvement over the work by Haidar et al. [10], which is available in the MAGMA library.

2 Algorithmic Background

The QR factorization decomposes a dense matrix $A_{m\times n}$ into the product $Q_{m\times m} \times R_{m\times n}$, where Q is an orthogonal matrix, and R is upper triangular. Throughout the paper, we assume $m \geq n$. The standard LAPACK implementation does not compute Q explicitly. Upon completion, the matrix A is overwritten by the two matrices V and R, as shown in Fig. 1a. The matrix V is lower triangular with unit diagonals (not stored), such that each column v_i represents an elementary Householder reflector $H_i = I - \tau_i v_i v_i^T$, where τ is a scalar (stored separately). The Q factor is computed as $Q = \prod_i^n H_i$.



(a) QR factorization output (b) Building blocks

Fig. 1. The LAPACK convention of the QR factorization

Assuming double precision, the standard LAPACK implementation is available in the `dgeqrf` routine, which has the building blocks shown in Fig. 1b. Both the `dgeqr2` and `dgeqrf` routines perform the QR factorization. However, `dgeqr2` is an *unblocked* design, meaning that it proceeds one column at a time, building the corresponding elementary reflector (`dlarfg`), and applying it to the rest of the matrix (`dlarf`). Therefore, `dgeqr2` is limited by the memory bandwidth of the hardware, since it relies on vector or matrix-vector operations only (BLAS level 1 and 2). On the other hand, `dgeqrf` is a *blocked* design. It uses `dgeqr2` to factorize a rectangular *panel*. The corresponding block of reflectors are applied to the trailing matrix using matrix-matrix (L3 BLAS) operations. The use of L3 BLAS enables `dgeqrf` to be compute-bound. The application of the block reflectors contains a preparatory stage (`dlarft`), during which a triangular factor T is computed from the V matrix and the scalars $\tau_i, i \in \{1, 2, \cdots, n\}$, such that $Q = I - V \times T \times V^T$. The last equation takes advantage of matrix multiplication (GEMM) when implicitly applying Q to the trailing matrix.

2.1 Nested Blocking

A standard QR factorization directly calls the unblocked panel factorization (`dgeqr2`). For a batch of relatively small matrices, the panel is thin, typically 4–8 in most cases. Thin panels lead to rank-k updates (batch GEMM) that are memory-bound. On the other hand, passing relatively wide panels directly to the memory-bound `dgeqr2` also hinders the performance. The solution to this tradeoff is to use *nested blocking*, which is a well-known approach in LAPACK's blocked algorithms, despite not being used in the standard QR implementation. Figure 2 shows the general idea of nested blocking, where a wide panel is internally split during its factorization. Nested blocking increases the reliance on L3 BLAS operations (batch GEMM).

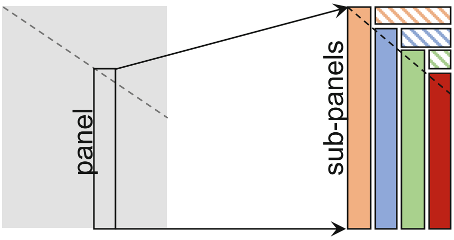


Fig. 2. Nested blocking in the QR panel factorization. The horizontal rectangles refer to parts of the matrix that are touched solely by the update step.

2.2 Computing the Triangular Factor *T*

The original implementation of the `dlarft` routine relies on two memory-bound operations, the matrix-vector product (`dgemv`), and the triangular matrix-vector product (`dtrmv`). For a block-reflector V of width nb, the factor $T_{nb \times nb}$ can be computed recursively as in Algorithm 1, where lines 2 and 3 update the same column of T using `dgemv` and `dtrmv` operations, respectively. However, previous work [11] has shown that all the calls to `dgemv` can be aggregated into one `dgemm` call, while the `dtrmv` calls remain roughly unchanged. While Algorithm 2 clearly shows a performance advantage over Algorithm 1, it needs preprocessing stages that may be costly for small matrices. In order to call `dgemm`, the matrix V must be separated from the R factor (refer to Fig. 1a). This requires (1) copying the R matrix into a workspace (`dlacpy`), (2) setting the upper triangular part of

V to zeros, with units on the diagonal (`dlaset`), and (3) another copy to bring R back on top of V (`dlacpy`). In addition to the overhead of these calls, there is also workspace management overhead. Note that there are other methods for computing the block Householder transformations [18,20], which are beyond the scope of this paper.

Algorithm 1: Classical `dlarft`

```
for $j$=1 to nb do
    $T_{1:j-1,j} = -\tau_j V_{j:n,1:j-1}^T \times V_{j:n,j}$
    $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times T_{1:j-1,j}$
    $T_{j,j} = \tau_j$
end
```

Algorithm 2: Improved `dlarft`

```
$T_{1:nb,1:nb} = V_{1:n,1:n}^T \times V_{1:n,1:n}$
for $j$=1 to nb do
    $T_{1:j-1,j} = -\tau_j T_{1:j-1,1:j-1} \times T_{1:j-1,j}$
    $T_{j,j} = \tau_j$
end
```

3 Experimental Setup

Throughout the paper, we show the incremental performance improvements on a system equipped with an NVIDIA Tesla A100-SXM4 GPU, which is clocked at 1.41 GHz and has 80 GB of memory. The GPU is hosted by an AMD EPYC 7742 64-Core Processor, clocked at 2.25 GHz. The CUDA version is 11.2. The final performance results are collected on this system as well as on another system equipped with an AMD Instinct MI100 GPU, which has 32 GB of memory, and clocked at 1.5 GHz. The ROCM version is 4.5.0. The host CPU is an AMD EPYC 7662 64-Core Processor, running at 3.25 Ghz. All the developments are lined up to be released in the MAGMA library. Our solution will be referenced as "MAGMA" in all the performance results. For NVIDIA GPUs, the performance results are compared against the batch QR factorization in the cuBLAS library, as well as against the open source KBLAS library [9]. For AMD GPUs, the performance is compared against the hipBLAS library.

4 LAPACK-Style Design

Our goal is to maximize the batch QR factorization performance on any matrix size and shape. A straightforward approach is to extend the primitive building blocks in Fig. 1b to support a batch of matrices. There are two advantages to this approach. First, it uses some of the existing batch BLAS routines, like batch GEMM, which are often highly optimized by the vendor libraries, or by open source libraries [2]. For the batch QR factorization in particular, the reliance on an optimized batch GEMM routine guarantees performance portability across different GPU architectures. Second, since the building blocks are assumed to be LAPACK-compliant, the final implementation would support any matrix size and shape. This is unlike some previous efforts that target application-specific range of sizes [7,15].

Our first implementation of batching the building blocks is based on the efforts by Haidar et al. [10]. It is improved by taking into account some of the new features in the vendor libraries, especially the more optimized batch

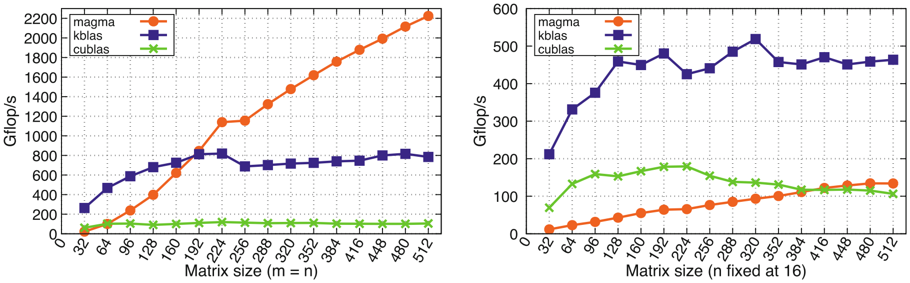


Fig. 3. Batch QR performance in double precision. The MAGMA design is based on batching the building blocks of Fig. 1b. Results are shown for square matrices (left) and tall-skinny matrices (right), using a Tesla A100 GPU.

GEMM kernels. Figure 3 shows the performance results on square and tall-skinny matrices. For square sizes, KBLAS outperforms the original MAGMA design for sizes less than 192. Otherwise, MAGMA has the best performance. Our profiling results show that only MAGMA calls batch GEMM underneath, which explains why its performance scales well, while cuBLAS and KBLAS stagnate.

Table 1. Time breakdown (%) for the original MAGMA design. Results are shown for double precision on the A100 GPU, with 1000 matrices per batch.

Category	Matrix size (m, n)			
	(128,16)	(64,64)	(128,128)	(256,256)
`dgeqr2` kernels	**64.4**	**55.09**	**45.86**	30.28
`dgemm`	12.47	24.68	35.4	**56.23**
Auxiliary (`dlacpy`, `dlaset`, etc.)	21.28	15.93	14.48	10.48
`trmv` (for `dlarft`)	1.85	4.29	4.26	3.01

For tall-skinny matrices, both cuBLAS and KBLAS have a clear advantage over MAGMA. Our conclusion is that the MAGMA design favors wide matrices, where the trailing matrix update is rich in batch GEMM calls. For tall-skinny matrices, such an advantage is absent. In addition, the current panel implementation in MAGMA lacks optimizations for tall-skinny matrices. To emphasize this point, Table 1 shows the percentage of time spent in different parts of the MAGMA design for four selected sizes. It shows that the panel kernels contribute significantly to the total execution time. Therefore, we cannot rely on batch DGEMM alone in order to achieve high performance. The QR panel must undergo an extensive optimization. Since the `dgeqr2` kernels are memory-bound, it is imperative to save memory traffic as much as possible. This can be achieved by merging multiple building blocks into a single execution context, which is often called *kernel fusion*. We use a *multi-level* kernel fusion, in which different parts of the algorithm are fused based on the matrix size.

5 Panel Optimization

We begin by designing a new GPU kernel for the panel factorization. The kernel caches a panel of size $m \times nb$ in the register file of the GPU and implements the unblocked factorization (`dgeqr2`). In a thread block, each thread possesses one row of the panel, so at least m threads are required for each thread block. There are two occasions where we perform a reduction operation across the columns of the panel. The first is during the generation of the Householder reflector, where the norm of the current column is computed. The second is when applying the reflector to the trailing matrix, i.e., $A = (I - \tau v v^T) \times A$. The product $v^T A$ requires a reduction operation across the columns of A. Since these reductions contradict with the thread-per-row assignment, a shared memory workspace is allocated to perform tree reductions. Note that the $v^T A$ product involves a multi-column reduction, for which we re-organize the threads into independent groups, and each group collaboratively reduces the assigned column. A key design aspect of this kernel is the use of compile-time constants. For example, the width of the panel nb must be known at compile time in order to avoid register spilling. It also helps the compiler unroll most of the loops inside the kernel. The kernel is instantiated for $1 \leq nb \leq nb_{max}$, where nb_{max} depends on the GPU resources as well as the compute precision.

The performance of this kernel is dependent on the height of the panel, since it requires one thread per row. For example, a panel of size $512 \times nb$ needs 512 threads. Assuming that all other resources are not a bottleneck, we can schedule four thread blocks at maximum per SM, due to a hardware limitation. Panels taller than 512 would cause at least 25% drop in the thread occupancy per SM. Depending on the width of the panel, other resources could be underutilized as well. The remedy to such a behavior is to relax the constraint on the number of threads. We propose a second kernel that stores the panel in shared memory instead. This enables us use any number of threads to factorize the panel. The proposed kernel assumes $nb \leq$`#threads`$\leq m$. The tree reductions mentioned above are redesigned to work with any number of threads in that range. To prove our point, Fig. 4 shows the performance of the two kernels for a panel of width 4. For the shared memory kernel, we use 32, 64, and 128 threads. The figure shows that there is no clear winner, and that two decisions should be made before the panel factorization: (1) which kernel should be used (register vs. shared memory), and (2) if the shared memory kernel is used, how many threads should be used? All performance graphs in this figure have a staircase-like behavior. As the panel becomes taller, more resources are required, leading to drops in occupancy. In order to select the best performing kernel, we collect offline tuning data based on panel width, precision, and GPU architecture. These data are used to select such a kernel at run time.

Figure 5 shows the updated performance after incorporating the fused `dgeqr2` kernels. For square matrices, the performance of MAGMA is improved by 12.9%–57.1%, while the speedups for the tall-skinny case are in the range 11.9%–41.4%. We generally observe that the smaller the matrix, the larger the speedup. This is expected, since the savings in memory traffic should be more critical for smaller

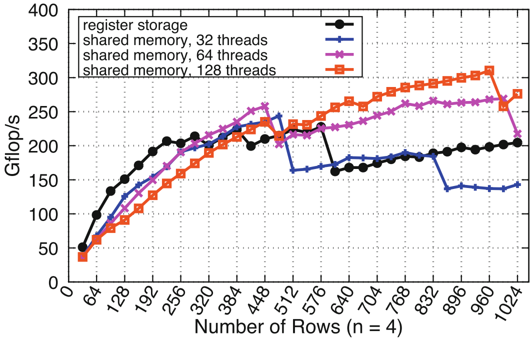


Fig. 4. Comparing different kernels for the fused `dgeqr2` step. Results are shown for double precision using a Tesla A100 GPU, with 1000 per batch.

problems. However, the general behavior against cuBLAS and KBLAS remains the same, except for the slightly earlier intersection points with the MAGMA performance graphs.

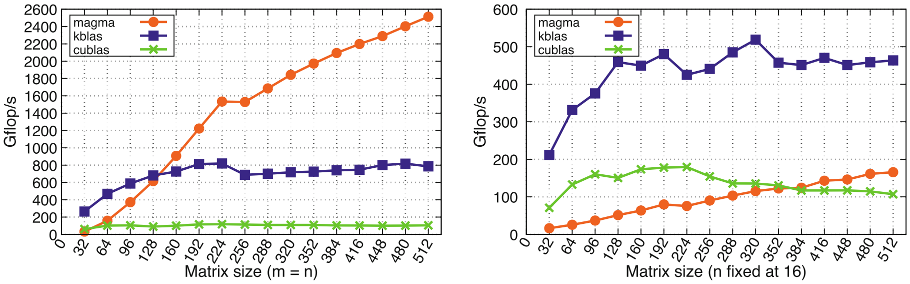


Fig. 5. Batch QR performance in double precision, with the fused panel kernels incorporated into the MAGMA design. Results are shown for square matrices (left) and tall-skinny matrices (right), using a Tesla A100 GPU.

In order to decide where the next optimization should be, we repeat the time breakdown experiment after using the fused panel kernels, which is shown in Table 2. The table shows that the panel kernels are no longer dominant in any of the four sizes. Another positive sign is the increased percentage of the `dgemm` kernel. However, the auxiliary kernels now contribute a noticeable amount of time, and even dominate the execution time for size 128 × 16. Recall that these auxiliary kernels are mostly called in a setup phase for computing the T factor (Sect. 2.2). In addition, if the size of T is small, calling batch GEMM multiple times on small matrices might be inefficient. We need to avoid these auxiliary kernels as much as possible, especially for tall-skinny sizes.

Table 2. Time breakdown (%) for the MAGMA design with fused `geqr2`. Results are shown for double precision on the A100 GPU, with 1000 matrices per batch.

Category	Matrix size (m, n)			
	(128,16)	(64,64)	(128,128)	(256,256)
`dgeqr2` kernels	24.37	11.37	10.26	7.81
`dgemm`	26.39	**48.66**	**58.78**	**74.41**
Auxiliary (`dlacpy`, `dlaset`, etc.)	**45.29**	31.53	23.89	13.81
`trmv` (for `dlarft`)	3.95	8.44	7.07	3.96

6 Optimizing the Trailing Matrix Update

We acknowledge that we cannot use the batch `dgemm` kernel to compute the T factor for relatively thin matrices. At the same time, using the memory bound `dgemv` and `dtrmv` kernels is not expected to be a faster solution. Since this improvement is critical mostly for tall-skinny sizes, a candidate solution is to merge the `dlarft` and the `dlarfb` operations into one GPU kernel. However, since the fused kernels operate on the fastest memory levels of the GPU, the implementation can be simplified into applying the elementary reflectors directly to the trailing matrix (without forming the T factor). This strategy is partially similar to cuBLAS and KBLAS in the sense that they don't use the batch GEMM for the trailing matrix update. However, we limit its use for a certain width, as we discuss later in the paper. Algorithm 3 shows a pseudo code of the proposed kernel. It reads the output of `dgeqr2` into shared memory, setting its upper triangular part to zeros, and its diagonal to ones. The factorized panel remains cached for the lifetime of the kernel. Assuming that the trailing matrix has a width $\bar{n}$, we loop over this width in a small step `ib`, so that the sub-trailing panel (`tA[]`) is cacheable in either the shared memory or the register file. We use a device routine implementation of the `dlarf` routine to apply each reflector in the panel. The `dlarf` routine has an optimized multi-column tree reduction and a parallel rank-1 update, which are the two main components required for the update. Finally, the `tA[]` buffer is written into the main memory, and a new sub-trailing panel is loaded. Similar to the fused panel kernel, there are two implementations of the update kernel, one that uses the register file for storing `tA[]` and uses a restricted number of threads, while the other uses shared memory only, and has a tunable number of threads.

An important point is that the fused panel and update kernels can be used to factorize the entire matrix, without utilizing the batch `dgemm` kernel. This decision is dependent on so many parameters, like the dimensions (m, n) of the matrix, the compute precision, and the GPU architecture. To achieve the best performance, we conducted a set of offline tuning sweeps that discover the *cut-off width*, below which we should use the fused panel/update kernels. The results from the tuning sweeps are stored in lookup tables. While we originally tuned the performance for the A100 GPU, it is straightforward to add lookup tables to

Algorithm 3: Pseudo code for the fused trailing panel update

```
pA[] ← read factorized panel in shared memory
pA[] ← dlaset(pA[], ‘upper’, 0, ‘diag’, 1) //device-routine
for =1 to $\bar{n}$ step ib do
    tA[] ← read the next block of columns from the trailing panel
    for j=1 to ib do
        tA ← dlarf(pA(:,j), tA[]) //device-routine
    end
    write tA[] back into memory
end
```

other GPUs. During the run time, the correct lookup table is used for deciding the best code path to execute. Figure 6 shows the performance of the MAGMA design after incorporating the new update kernel, where MAGMA is now able to outperform both cuBLAS and KBLAS across almost all sizes.

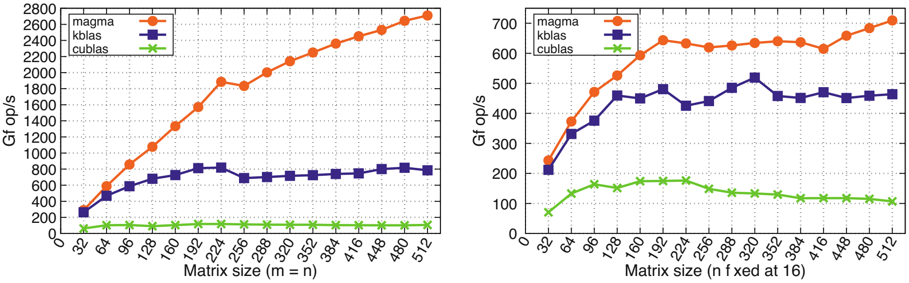


Fig. 6. Batch QR performance in double precision, with both the fused panel and fused update kernels incorporated into the MAGMA design. Results are shown for square matrices (left) and tall-skinny matrices (right), using a Tesla A100 GPU. Batch size = 1k.

7 Optimizations for Sub-warp Dimensions

A final optimization is possible when the entire matrix can be cached in the register file or in the shared memory. At this point, a fully fused and unrolled `dgeqr2` is used. We have addressed this case in a previous work [4], but we discuss it here to complete the scope of the paper. The kernel has some similarities with the one described in Sect. 5, but it has some unique features. First, it uses a serial reduction for computing the norm of a column, and for the $v^T \times A$ product. For sub-warp dimensions, we found out that a serial reduction is often faster than a parallel reduction with repetitive synchronizations. Second, one warp can be involved in factorizing more than one matrix simultaneously. For example, a single warp can factorize four 8×8 matrices at the same time. Third, the code template is instantiated for every possible size. Without the loss of generality, we discuss square sizes up to 32 only. The obvious drawbacks to this approach is its applicability to a restricted range of sizes. It should also be instantiated for every possible (m, n) combination. However, its advantage is clear as shown in Fig. 7. Despite all the optimizations mentioned in the previous sections and in other libraries, the Figure shows significant speedups, up to 3.22× against the best competition.

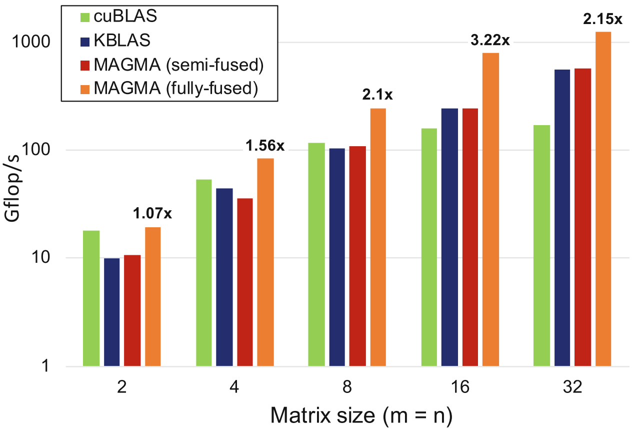


Fig. 7. Batch QR performance in double precision for tiny square matrices. Results are shown for a Tesla A100 GPU. Batch size = 10k. The speedup labels of the fully-fused design are calculated with respect to the best performance of the other three approaches.

8 The Big Picture

The optimized GPU kernels described in Sects. 4 through 7 are now put together into one solution. The factorization begins by a check for tiny matrices, for which the fully unrolled kernel (Sect. 7) can be used. Otherwise, it moves to a decision-making layer that determines whether to use a fused panel/update kernels for the entire factorization (Sects. 5 and 6).

– **If true**, another decision-maker determines which version of the panel/update to be used (in registers or in shared memory). The decision-maker also determines the number of threads in case the shared memory version is preferred.
– **If false**, the factorization proceeds with a LAPACK-style factorization utilizing batch GEMM.

The LAPACK-like implementation has a panel factorization step, during which it checks agains for the feasibility fused panel/update kernel. If they cannot be used (e.g. panel is too large), we fall back to a generic non-fused panel implementation. In either case, the factorization proceeds with computing the T factor and then calling batch GEMM to apply the block reflector to the trailing matrix. All the decision-making layers use a comprehensive set of offline performance benchmark results. The offline data resulting from these benchmarks are tabulated per GPU and per compute precision.

9 Final Performance Results

This section shows the final performance results on both NVIDIA and AMD GPUs. All results are shown for single and double precisions. We show the performance of the host CPU using OpenBLAS, which is called inside an OpenMP for loop using 64 threads.

Figure 8 shows the performance on the A100 GPU for square matrices. MAGMA has a clear asymptotic advantage thanks to the careful utilization of the batch GEMM kernel (from both cuBLAS and MAGMA's own kernel). As mentioned before, the performance graph of MAGMA is the marriage of three different factorization strategies. The first is the fully fused factorization for sizes ≤32, the second is performing the factorization using the fused panel/update kernel only, and the third is the LAPACK style strategy utilizing batch GEMM. For single/double precision, MAGMA is up to 2.3×/3.3× faster than KBLAS, up to 16.2×/25.4× faster than cuBLAS, and up to 21.9×/14.8× against OpenBLAS+OpenMP.

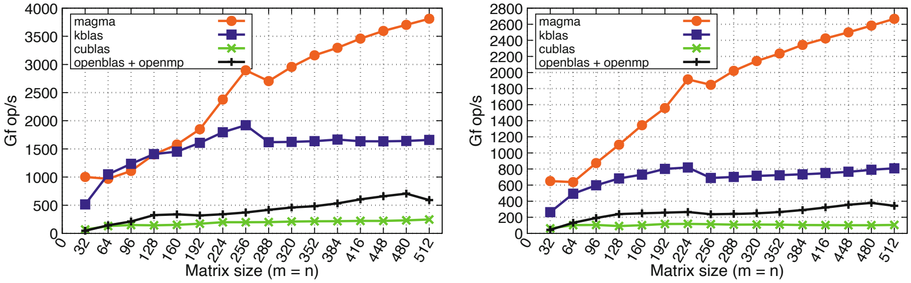


Fig. 8. Final performance of the batch QR factorization in single/double precision (left/right). Results are shown for square matrices using a Tesla A100 GPU. Batch size = 1k.

Figure 9 shows the final performance for tall-skinny matrices with exactly 16 columns. This test case represents problems that require the solution of least square systems. This test case corresponds to one factorization strategy in MAGMA, which is the fused panel and update kernels. But recall that MAGMA has two different kernels for each of the panel and update steps, and invokes the faster of the two depending on the matrix size. Similar to square sizes, both cuBLAS and the OpenBLAS with OpenMP are underperforming. Both MAGMA and KBLAS have the staircase-like behavior, which means that they both try to take advantage of the fast memory levels on the GPU, but face gradual degradation due to increased occupancy. However, MAGMA has an asymptotic advantage for single precision, and an overall advantage for double precision. This means that our solution has a better use of the available resources on the GPU. For single/double precision, MAGMA is up to 1.6×/1.7× faster than KBLAS, up to 5.8×/7.4× faster than cuBLAS, and up to 36.3×/65.9× against OpenBLAS+OpenMP.

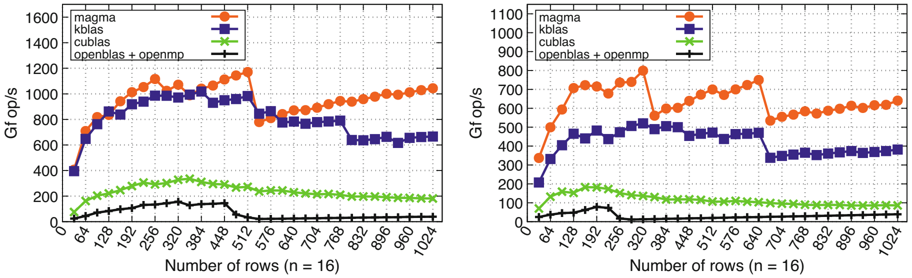


Fig. 9. Final performance of the batch QR factorization in single/double precision (left/right). Results are shown for tall-skinny matrices (n = 16) using a Tesla A100 GPU. Batch size = 1k.

Figures 10 and 11 show the corresponding results on the AMD MI100 GPU, where we compare the MAGMA performance against hipBLAS as well as OpenBLAS + OpenMP. To the best of our knowledge, KBLAS does not support AMD GPUs. We observe that the performance is lower than the A100 performance numbers. This is due to multiple reasons. **First**, the batch GEMM kernel on the A100 is better tuned for the use cases we need than on the MI100 GPU. **Second**, we notice that the fused kernels for performing the panel and the updates are also slower than on the A100. Our experience with porting our solution to AMD GPUs indicates that performing computations in the Local Data Share (LDS) memory is slower than the shared memory on NVIDIA GPUs. This is crucial to both the panel and the update kernels, since we perform many tree reduction in shared memory. MAGMA still outperforms hipBLAS for square sizes. The speedups range between 2.6× and 11.5× for single precision, and between 3.8× and 10.1× for double precision.

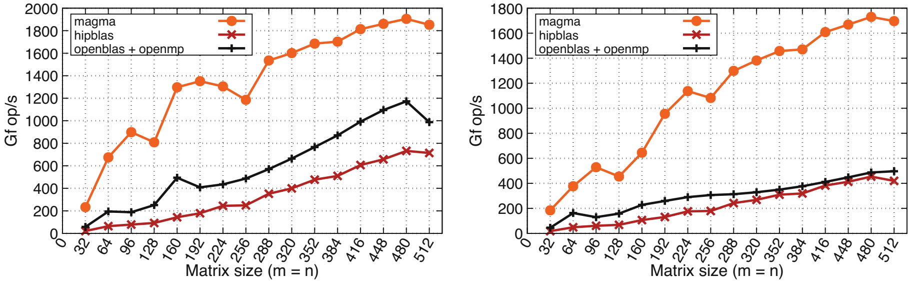


Fig. 10. Final performance of the batch QR factorization in single/double precision (left/right). Results are shown for square matrices using an AMD MI100 GPU. Batch size = 1k.

Another bottleneck on the MI100 GPU is the amount of the LDS memory available for one thread-block, which has a maximum of 64KB. This is nearly half the amount that we can allocate dynamically on the A100 GPU. This limits the ability of MAGMA to cache relatively large panels, and forces it to switch to either use thinner panels or to use the LAPACK-style factorization. Both situations hinder the performance due to the increased memory traffic. We also observe that the staircase shape in Fig. 11 are more frequent and more severe, which can also be explained by the relatively limited opportunities of data reuse. MAGMA still outperforms hipBLAS for tall-skinny matrices. The speedups range between 3.0× and 14.5× for single precision, and between 1.13× and 12.6× for double precision.

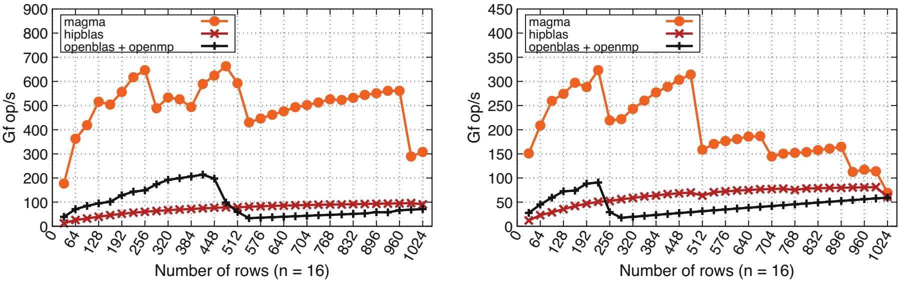


Fig. 11. Final performance of the batch QR factorization in single/double precision (left/right). Results are shown for tall-skinny matrices (n = 16) using an AMD MI100 GPU. Batch size = 1k.

In general, the asymptotic performance is not close to the GPU theoretical peak performances. This is mainly due to the focus on relatively small sizes, which limits the batch GEMM performance on both the A100 and the MI100 GPUs. The rank-updates use relatively small widths that are not enough to saturate the GPU compute power. Note that the batch DGEMM kernel from cuBLAS uses the Tensor Cores units, and the batch SGEMM kernel from hipBLAS uses the Matrix Core units. A possible performance improvement is to incorporate these accelerators in MAGMA's own batch GEMM kernel, and tune them specifically for these rank updates.

10 Conclusion and Future Work

This paper shows the underlying complexity of optimizing batch linear algebra operations on GPUs, taking the dense batch QR factorization as an example. We show that, depending on the problem size, there could be different strategies of performing the factorization. Since memory traffic is often critical to batch routines, fused kernels are used to efficiently utilize the memory bandwidth. However, kernel fusion increases the complexity of the overall solution, since it introduces new non-standard routines not found in BLAS or LAPACK. Our final solution for the batch QR factorization has three different strategies for execution, and within each strategy, there are multiple run-time decisions to select the best performing kernel. Future directions include investigating the performance regression on AMD GPUs, extension to variable-size batches, and considering more efficient algorithms for very tall and skinny matrices.

References

1. LAPACK - Linear Algebra PACKage. http://www.netlib.org/lapack/
2. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.: Performance, design, and autotuning of batched GEMM for GPUs. In: ISC High Performance 2016, Frankfurt, Germany, 19–23 June 2016, Proceedings, pp. 21–38 (2016)
3. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.: Factorization and inversion of a million matrices using GPUs: challenges and countermeasures. Procedia Comput. Sci. **108**, 606–615 (2017). ICCS 2017, Zurich, Switzerland
4. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.J.: Batched one-sided factorizations of tiny matrices using GPUs: challenges and countermeasures. J. Comput. Sci. **26**, 226–236 (2018)
5. hipBLAS. https://github.com/ROCmSoftwarePlatform/hipBLAS
6. Anderson, M., Sheffield, D., Keutzer, K.: A predictive model for solving small linear algebra problems in GPU registers. In: IEEE 26th International Parallel Distributed Processing Symposium (IPDPS) (2012)
7. Anzt, H., Dongarra, J., Flegar, G., Quintana-Ortí, E.S.: Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. PMAM 2017, pp. 1–10. ACM, New York (2017)
8. Auer, A.A., et al.: Automatic code generation for many-body electronic structure methods: the tensor contraction engine. Mol. Phys. **104**(2), 211–228 (2006)
9. Boukaram, W.H., Turkiyyah, G., Ltaief, H., Keyes, D.E.: Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. Parallel Comput. (2017). https://doi.org/10.1016/j.parco.2017.09.001
10. Haidar, A., Dong, T., Luszczek, P., Tomov, S., Dongarra, J.: Batched matrix computations on hardware accelerators based on GPUs. IJHPCA **29**(2), 193–208 (2015)
11. Haidar, A., Tomov, S., Luszczek, P., Dongarra, J.: Magma embedded: towards a dense linear algebra library for energy efficient extreme computing. In: 2015 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6, September 2015
12. MAGMA. http://icl.cs.utk.edu/magma/
13. PLASMA, October 2017. https://bitbucket.org/icl/plasma
14. Intel Math Kernel Library. http://software.intel.com/intel-mkl/
15. Kurzak, J., Anzt, H., Gates, M., Dongarra, J.: Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. IEEE Trans. Parallel Distrib. Syst. **27**, 2036–2048 (2015)
16. Messer, O., Harris, J., Parete-Koon, S., Chertkow, M.: Multicore and accelerator development for a leadership-class stellar astrophysics code. In: Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing" (2012)
17. NVIDIA CUBLAS. https://developer.nvidia.com/cublas
18. Tomás Dominguez, A.E., Quintana Orti, E.S.: Fast blocking of householder reflectors on graphics processors. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 385–393 (2018). https://doi.org/10.1109/PDP2018.2018.00068

19. Van Zee, F.G., van de Geijn, R.A.: BLIS: a framework for rapidly instantiating BLAS functionality. ACM TOMS **41**(3), 33 (2015). https://dl.acm.org/doi/10.1145/2764454
20. Walker, Homer F.: Implementation of the GMRES method using householder transformations. SIAM J. Sci. Stat. Comput. **9**(1), 152–163 (1988). https://doi.org/10.1137/0909010
21. Yeralan, S.N., Davis, T.A., Sid-Lakhdar, W.M., Ranka, S.: Algorithm 980: sparse QR factorization on the GPU. ACM TOMS **44**(2), 17:1–17:29 (2017)