Journal of Computational Science xxx (2018) xxx–xxx

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures☆

Ahmad Abdelfattah[a,*], Azzam Haidar[a], Stanimire Tomov[a], Jack Dongarra[a,b,c]

[a] Innovative Computing Laboratory, University of Tennessee, Knoxville, USA
[b] Oak Ridge National Laboratory, Oak Ridge, USA
[c] University of Manchester, UK

ARTICLE INFO

Article history:
Received 14 October 2017
Received in revised form
30 November 2017
Accepted 28 January 2018
Available online xxx

Keywords:
GPU computing
Matrix factorization
Batch computation

ABSTRACT

The use of batched matrix computations recently gained a lot of interest for applications, where the same operation is applied to many small independent matrices. The batched computational pattern is frequently encountered in applications of data analytics, direct/iterative solvers and preconditioners, computer vision, astrophysics, and more, and often requires specific designs for vectorization and extreme parallelism to map well on today's high-end many-core architectures. This has led to the development of optimized software for batch computations, and to an ongoing community effort to develop standard interfaces for batched linear algebra software. Furthering these developments, we present GPU design and optimization techniques for high-performance batched one-sided factorizations of millions of *tiny* matrices (of size 32 and less). We quantify the effects and relevance of different techniques in order to select the best-performing LU, QR, and Cholesky factorization designs. While we adapt common optimization techniques, such as optimal memory traffic, register blocking, and concurrency control, we also show that a different mindset and techniques are needed when matrices are tiny, and in particular, sub-vector/warp in size. The proposed routines are part of the MAGMA library and deliver significant speedups compared to their counterparts in currently available vendor-optimized libraries. Notably, we tune the developments for the newest V100 GPU from NVIDIA to show speedups of up to 11.8×.

1. Introduction

Batch computations apply the same numerical algorithm to a fairly large number of relatively small problems. The batched computing workload is quite different than the common workload for just one, typically large, matrix. The latter is served well by many software packages, including LAPACK [2], ScaLAPACK [3], PLASMA [4], and MAGMA [5]. The former, however, is relatively recent, and gained a lot of attention in many scientific communities, e.g., quantum chemistry [6], sparse direct solvers [7], astrophysics [8], and signal processing [9]. Software libraries such as Intel's MKL [10], NVIDIA's cuBLAS [11], and MAGMA recently started to provide highly optimized batched routines for many of the BLAS and LAPACK operations.

Existing numerical linear algebra software packages rarely achieve good performance on matrices of small sizes, because most of the optimization techniques that they carry out pay off only on large matrices. For example, the hybrid *lookahead* technique in MAGMA [12] is used to overlap the panel factorization (on the CPU) with the trailing matrix update (on the GPU). This design strategy is not efficient for small sizes, since the updates are no longer compute intensive, and therefore fail to overlap the panel factorization and the CPU-GPU communication. This is why new developments with different design strategies are needed.

While there have been new developments for GPU accelerated batched computations, for example the work done by Haidar et al. [13] and Abdelfattah et al. [14], there is still room for significant improvements when the matrix sizes are *tiny*. For these extremely small problems, the LAPACK-style blocking cannot achieve high performance, even if it is carried out on the GPU solely. Since the computation becomes memory bound on such small sizes, the cost of writing the factorized panel and then reading it back to perform the update becomes significant. Furthermore, the parallelization (to achieve sufficiently high occupancy) and the vectorization (for efficient warp use) become more challenging when sizes are less

☆ This is an extended version of our conference paper [1] that was invited to the JoCS special issue (https://doi.org/10.1016/j.procs.2017.05.250).

* Corresponding author.
E-mail addresses: ahmad@icl.utk.edu (A. Abdelfattah), haidar@icl.utk.edu (A. Haidar), tomov@icl.utk.edu (S. Tomov), dongarra@icl.utk.edu (J. Dongarra).

https://doi.org/10.1016/j.jocs.2018.01.005
1877-7503/

than 32, and must, for example, be done across matrices. Therefore, other design, parallelization, and vectorization strategies must be discovered in order to resolve the aforementioned issues regarding memory traffic and parallelization.

This paper presents highly optimized GPU kernels for batched one-sided factorizations. The paper extends the previous work by the authors for batched LU factorization and matrix inversion [1], and applies the same design principles to the QR, and Cholesky factorizations. In terms of the workload size, we consider one million matrices of tiny sizes, typically from 1 up to 32. In addition to the applications already mentioned, these factorizations are of particular importance to sparse direct solvers, such as the multifrontal solvers that can be found in SuiteSparse [15]. We adopt a step-by-step methodology, where incremental improvements in the kernel design lead to incremental performance gains. Such a methodology automatically justifies all of our design choices. While all the kernels share the same optimization techniques, our design for the LU factorization adopts a unique *lazy swap* strategy, which eliminates the expensive intermediate row interchanges, thus leading to a much faster kernel, but that is still numerically equivalent to an LAPACK-style LU-factorization. The performance results show significant speedups against the vendor-supplied cuBLAS kernels on a Pascal P100 GPU, as well as on the new Volta V100 GPU.

2. Related Work

The design of high performance dense linear algebra (DLA) software for GPUs was originally motivated by the high performance GPUs can achieve in embarrassingly parallel, compute intensive tasks, most notably on the matrix-matrix multiplication (GEMM) [16–18]. The high performance GEMM enabled the development of high performance DLA in libraries like MAGMA [5], where many of the LAPACK numerical algorithms are designed in a hybrid style to take advantage of both CPUs and GPUs [12].

The growing demand for high performance dense linear algebra on large batches of small matrices has led to early developments for batch matrix multiplication [19,20], which were then followed by more optimized kernels being available in cuBLAS, starting with version 8.0. The development of the batched GEMM in MAGMA enabled the development of batched one-sided factorizations routines based on the LAPACK-style blocking [13], but on a smaller scale. For example, while non-batched routines use a large blocking size (e.g., 512 to 1024) to get asymptotically optimal performance, batched routines block by much smaller sizes (e.g., 8 to 32), and rely on batched GEMM that is specifically tuned for small sizes in order to extract performance [21][22]. The developments for extremely small matrices, however, are more challenging. An approach that relies on separate panel/update stages [13] leads to redundant memory traffic. This cost can be affordable for medium sizes (e.g., 64 up to 256), but becomes significant for smaller sizes.

This is why other research efforts followed a *one-kernel approach*, where all computations are fused into a single GPU kernel. For example, Wang et al. [23] introduced FPGA-based parallel LU factorization of large sparse matrices, where the algorithm is reduced to factorizing many small matrices concurrently. Villa et al. [24] developed a GPU-based batched LU factorization, which has been used in subsurface transport simulation, where many chemical and microbiological reactions in a flow path are simulated in parallel [25]. Kurzak et al. [26] developed batched Cholesky factorization in single precision for sizes up to 100×100, which was used in an Alternating Least Squares (ALS) solver. Masliah et al. developed batched GEMM for very small sizes for both CPUs and GPUs [27]. Batched matrix inversion has been also introduced in the context of generating block-Jacobi preconditioners [28]. Batched QR factorization is of particular importance to H-matrix computation, as highlighted by Akbudak et al. [29], and by Boukaram et al. [30]. Kim et al. also introduced batched GEMM, triangular solve, and LU (no pivoting) for CPUs and Intel's Xeon Phi architectures based on a compact interleaved data layout [31].

This paper follows the same one-kernel approach to improve the MAGMA performance on very small sizes. It complements the work by Haidar et al. [13], which outperforms cuBLAS for medium and large sizes, but trails it for the sizes we focus on (up to 32).

3. Contributions

Below is a list of contributions for this paper.

1. Highly optimized GPU kernels for one-sided factorization on batch workloads. The developed kernels significantly outperform the state of the art designs from the vendor provided software. We typically consider single-node workloads that involve millions of extremely small matrices.
2. A set of unified design techniques that are oblivious to the three algorithms considered (LU, QR, and Cholesky factorizations). The paper manages to find a common ground among the three algorithms to achieve high performance.
3. The paper presents a detailed study of the different choices for every aspect of the kernel design, including thread configuration, matrix storage, occupancy, and others. The paper justifies the final design choice by showing intermediate performance results for different choices. Such a detailed study can be considered as a guide for designing other algorithms on similar workloads.

4. Background

This section introduces the computational steps for the LU, QR, and Cholesky factorizations on square matrices of size N×N. The description follows the LAPACK notations in double precision arithmetic.

The LU factorization computes the L and U factors of a general matrix A, such that A = P×L×U, where P is a permutation matrix that reflects the row interchanges required for pivoting. The matrix L is unit lower triangular, while U is upper triangular. The permutation matrix P is stored in a compact format using a *pivot vector* (IPIV), such that for $i \in \{1, 2, \cdots, N\}$, row i has been swapped with row IPIV(i).

There are four main steps in performing the unblocked LU factorization. Namely, these are: (1) locate the maximum absolute value in the current column (IDAMAX); (2) swap current row with the row with maximum absolute value in the current column (DLASWP); (3) scale the current column (DSCAL); and (4) rank − 1 update (DGER). Algorithm 1 shows the factorization using the four steps. According to LAPACK working notes [32], the LU factorization of a square matrix performs $(\frac{2N^3}{3} - \frac{N^2}{2} + \frac{5N}{6})$ floating point operations (FLOPs).

Algorithm 1. Unblocked LU factorization.

```
for i=1 to N do
    IPIV[i] = max_id = IDAMAX( ABS( A[i:N,i] ) )
    if ABS(A[max_id,i]) = ZERO then
        //U is singular, report error.
    end
    DLASWP: Exchange rows A[i, 1:N] and A[max_id, 1:N]
    DSCAL: A[i+1:N,i] *= (1 / A[i,i])
    DGER: A[i+1:N,i+1:N] -= A[i+1:N,i]×A[i,i+1:N]
end
```

While the LU factorization is able to factorize symmetric positive definite (SPD) matrices, the Cholesky factorization, shown in Algorithm 2, introduces a much faster algorithm for such matrices. The algorithm factorizes an SPD matrix $A = LL^T$, where L is a

Please cite this article in press as: A. Abdelfattah, et al., Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures, J. Comput. Sci. (2018), https://doi.org/10.1016/j.jocs.2018.01.005