Variable-size batched condition number calculation on GPUs

Hartwig Anzt
Karlsruhe Institute of Technology, Germany
University of Tennessee, Knoxville, USA
hartwig.anzt@kit.edu

Jack Dongarra
University of Tennessee, Knoxville, USA
Oak Ridge National Laboratory, USA
University of Manchester, Manchester, UK
dongarra@icl.utk.edu

Goran Flegar
Universidad Jaume I, Castellon, Spain
flegar@uji.es

Thomas Grützmacher
Karlsruhe Institute of Technology, Germany
thogru.kit@gmx.de

ABSTRACT

We present a kernel that is designed to quickly compute the condition number of a large collection of tiny matrices on a graphics processing unit (GPU). The matrices can differ in size and the process integrates the use of pivoting to ensure a numerically-stable matrix inversion. The performance assessment reveals that, in double precision arithmetic, the new GPU kernel achieves up to 550 GFLOPs (billions of floating-point operations per second) and 800 GFLOPs on NVIDIA's P100 and V100 GPUs, respectively. The results also demonstrate a considerable speed-up with respect to a workflow that computes the condition number via launching a set of four batched kernels. In addition, we present a variable-size batched kernel for the computation of the matrix infinity norm. We show that this memory-bound kernel achieves up to 90% of the sustainable peak bandwidth.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**;

KEYWORDS

Batched routines, GPU, condition number, numerical linear algebra

ACM Reference Format:
Hartwig Anzt, Jack Dongarra, Goran Flegar, and Thomas Grützmacher. . Variable-size batched condition number calculation on GPUs. In *Proceedings of (ICPP'18).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

The condition number of a matrix A quantifies how sensitive the linear system $Ax = b$ is with respect to changes in the right-hand side vector b [14, 15]. If the condition number is large, tiny changes in b can cause significant changes in the solution vector x. While this assumes exact arithmetic, the condition number is even more relevant when working with limited precision, which is the de-facto standard in scientific computing. As the limited precision format implies the rounding of values (e.g., those in the right-hand side vector), the condition number then determines the attainable accuracy of the solution. The same effect has to be taken into account when computing the inverse of a matrix. Scientific computations using a single precision format throughout the complete algorithm often ignore the relevance of the condition number. This is motivated by the fact that rounding effects impact all numerical operations in a similar way, and the only way to improve the accuracy is to transform the complete work-flow to operate in a higher precision format. In contrast, taking the condition number into account becomes essential in mixed precision algorithms, which handle part of the computations in a less accurate format than working precision. In that scenario, special care has to be paid to the numerical effects and rounding error propagation: For example, already casting the matrix A to a lower precision format can potentially turn a regular matrix into a singular one. Unfortunately, computing the condition number of a (large) matrix A is computationally expensive, and even though strategies for cheaply approximating the condition number in an iterative fashion have been developed [10], many mixed precision algorithms refrain from employing an explicit analysis and instead leave it to the application scientist to analyze the numerical effects [13].

A recently proposed mixed precision strategy to the preconditioned iterative solution of linear systems does not convert the system matrix to lower precision, but instead forms a block-Jacobi preconditioner where, if appropriate, the diagonal blocks are stored in reduced precision [6]. The elegance of this approach is that not all diagonal blocks need to use the same precision, but the precision format can be chosen locally, with each block adapted to its condition number. Hence, this approach does not require assessing the condition number of the system matrix A, but instead needs to inspect the condition number of each diagonal block in the Jacobi preconditioner. The preconditioner typically consists of a significant number of small blocks, which motivates the variabesize batched condition number routine presented in this paper. We provide the necessary background on condition number computation and batched routines in Section 2. We then develop the kernel computing the condition number for a large set of matrices on a graphics processing unit (GPU) in Section 3. In Section 4 we use runtime experiments on the latest server-line GPU architectures from NVIDIA to asses the routine performance and relate it to the cost of the block-Jacobi preconditioner generation. We also include a performance analysis on a variant computing the condition number by launching a set of four batched routines, including a

ICPP'18, ,

variable-size batched matrix infinity norm kernel. We conclude in Section 5 with an outlook on future research opportunities.

2 BACKGROUND AND RELATED WORK

2.1 Matrix conditioning

The condition number cond(A) of a matrix A reflects how sensitive the corresponding linear system $Ax = b$ is to small changes in the right-hand side vector b [14]. More precisely, for the error e to a solution x, the condition number is defined as the maximum ratio of the relative error in the solution x to the relative variation in the right-hand side vector b:

$$\begin{aligned}
\text{cond}(A) &= \max_{e,b\neq 0}\left(\frac{\frac{\|A^{-1}e\|}{\|A^{-1}b\|}}{\frac{\|e\|}{\|b\|}}\right) \\
&= \max_{e,b\neq 0}\left(\left(\frac{\left\|A^{-1}e\right\|}{\|e\|}\right)\cdot\left(\frac{\|b\|}{\|A^{-1}b\|}\right)\right) \\
&= \left(\max_{e\neq 0}\left(\frac{\left\|A^{-1}e\right\|}{\|e\|}\right)\right)\cdot\left(\max_{b\neq 0}\left(\frac{\|b\|}{\|A^{-1}b\|}\right)\right) \\
&= \left(\max_{e\neq 0}\left(\frac{\left\|A^{-1}e\right\|}{\|e\|}\right)\right)\cdot\left(\max_{x\neq 0}\left(\frac{\|Ax\|}{\|x\|}\right)\right) \\
&= \left\|A^{-1}\right\|\cdot\|A\|\,. \qquad (1)
\end{aligned}$$

This equation holds for any matrix norm and induced vector norm, in particular for the maximum norm $\|A\|_\infty$. The maximum norm can for both, the matrix and the vector norm, cheaply be computed as maximum of the absolute sums of the distinct matrix rows in A [12]:

$$\|A\|_\infty = \max_{1\leq i\leq m}\sum_{j=1}^{n}|a_{ij}|. \qquad (2)$$

We note that the row sums are unaffected by neither column nor row exchanges, and hence the infinity norm being unaffected by the application of the standard partial (i.e., row) pivoting. The algorithm we design in Section 3 for computing the condition number for a set of matrices is based on computing the infinity norm for each of the matrices and their corresponding inverses.

2.2 Batched routines

The qualifier "batched" identifies a procedure that applies the same operation to a large collection of data entities. In general, the subproblems (i.e., the data entities) are all small and independent, turning the overall problem into an embarrassingly-parallel operation [11]. An efficient batched algorithm employs a parallel formulation that simultaneously performs the operation on several/all subproblems to yield a more fruitful exploitation of the computational resources. This abstraction is particularly important on highly parallel architectures like GPUs, where scheduling one data entity after another may waste a large fraction of the computational resources. Batched routines also reduce the kernel invocation overhead as they replace a sequence of routine calls with a single kernel. In addition, if the data for the subproblems is conveniently stored in the GPU memory, a batched routine can orchestrate a more efficient (coalesced) memory access.

In recent years, the development of batched routines for linear algebra operations has received considerable interest because of their application in machine learning, astrophysics, quantum chemistry, hydrodynamics, and hyperspectral image processing, among others. Examples of batched kernels for the dense BLAS appear in [1, 3, 8, 9], and there exists a strong community effort on designing an interface standard for these routines [11]. Batched routines have also been developed for sparse linear algebra functionality, including batched sparse matrix-vector multiplication [5] and routines for generating sparse approximate inverses for incomplete triangular factors [4].

Batched routines are typically classified into two subsets: those where all data entities have the same size, and those where the data entities can differ in size (within a range). The latter type of batched routines, usually referred to as "variable-size," are more complicated in design, but offer higher flexibility in terms of target applications [2].

3 BATCHED CONDITION NUMBER ROUTINE

In [8] we designed a batched routine for the in-place inversion of variable-sized matrices (up to dimension 32×32) on GPUs. In this section, we review the key ideas in [8], and add the calculation of the condition number with little overhead to this scheme.

The calculation can be decomposed into four steps, see Figure 1: (1.1, 1.2) Calculate the infinity norm of A; (2) invert A; (3.1, 3.2) compute the infinity norm of A^{-1}; and (4) derive the condition number as $\text{cond}(A) = \|A\|_\infty \cdot \left\|A^{-1}\right\|_\infty$, according to (1). It is possible to realize these steps in separate stand-alone CUDA kernels or, as proposed in this work, using a single kernel. Combining the four components into a single kernel radically reduces the main memory access, which is typically crucial for the performance of batched routines. The complete procedure is realized entirely in registers. Each thread stores and is responsible for operations on a single row of the matrix, and data from other threads is communicated via warp shuffles. The entire matrix is read from main memory at the beginning of the routine, and written back when the routine finishes. Thus, there is no additional data movement necessary during the procedure.

Matrix inversion

Once the matrix is present in registers, the matrix inversion is realized by applying the variable-size batched Gauss-Jordan elimination ("GJE") described in [8]. The complete inversion process is handled in registers, and communication is realized via warp shuffles [17]. This limits the matrix size to problems of dimension less or equal than 32×32. In-place inversion avoids the need of additional memory.

To ensure numerical stability, we use the implicit pivoting strategy presented in [7]. Instead of swapping rows and assigning threads to a fixed row, we move the workload to the thread owning the data and keep track of a row swap history. If the inverse matrix is required, all the row swaps are applied at once when writing the inverse matrix back to main memory.

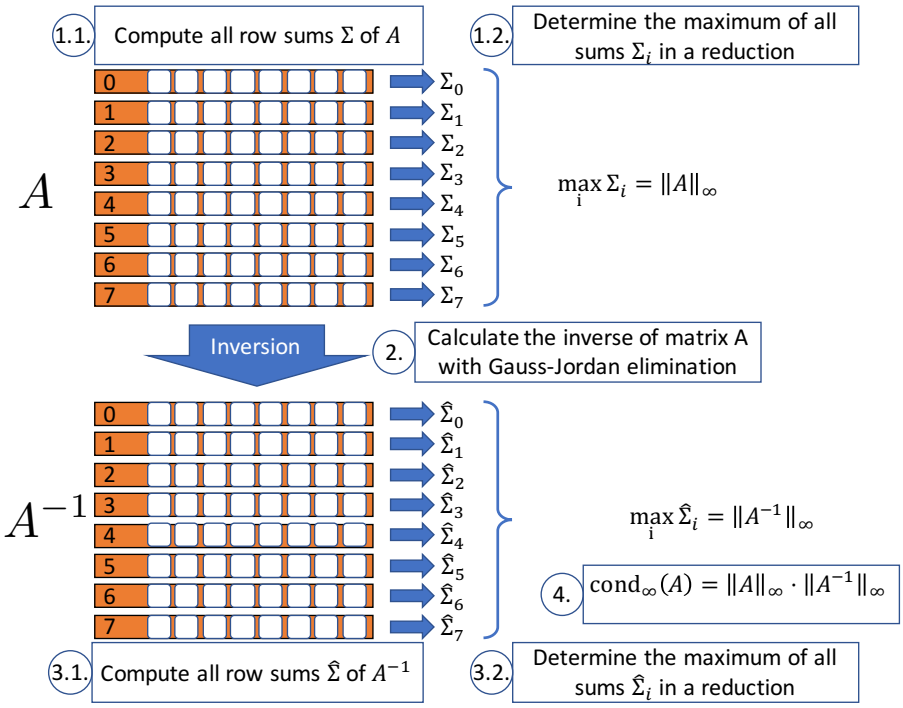


Figure 1: Calculation of the condition number with $A \in \mathbb{R}^{8\times8}$. The orange blocks denote threads and the inscription their thread-ID, each thread keeps its matrix row in registers.

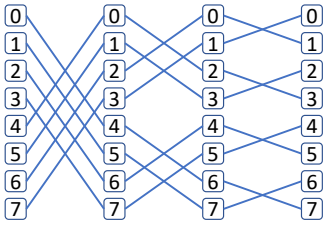


Figure 2: Reduction example with eight threads and the CUDA XOR-shuffle. The numbers represent the thread-ID. The connections between threads show an exchange of their current reduction value. At the end, all threads have the same result, which is why this reduction is also called "all-reduction".

Calculating matrix norms in registers

With each thread keeping a complete matrix row in registers, the absolute sum of each row can be computed in a data-parallel fashion. No communication is required for this step (see Figure 1, steps 1.1. and 3.1.). Finding the maximum of the absolute sums requires a global reduction over all threads assigned to the matrix (see Figure 1, steps 1.2. and 3.2.). The necessary communication can be realized efficiently using CUDA's XOR-shuffle (see Figure 2).

Multiple problems per warp

Using one warp for a single matrix is inefficient if the batch contains only matrices significantly smaller than the warp size. To tackle this, we follow the strategy proposed in [8] by taking advantage of the sub-warp support in CUDA shuffles. The sub-warp size has to be defined globally prior to the kernel invocation, and has to be a power of two [8]. Let k_m denote the size of the largest block in the matrix batch and assume p_m is the smallest power of 2 such that $p_m \geq k_m$. The size of a sub-warp is now set to p_m, which means that each warp can process $32/p_m$ matrices. The performance penalty of this approach is that, in every sub-warp, there are at least $p_m - k_m$ threads that remain idle and do not contribute to the result. The advantage, on the other hand, is that this yields a generic function that needs no preprocessing and avoids the expensive calculations that are necessary to determine the mapping between threads and rows.

4 EXPERIMENTAL ANALYSIS

In this section, we analyze the performance of the batched condition number routine experimentally.

We run the performance analysis on the latest two architectures in NVIDIA's server line for high performance scientific computing: the NVIDIA P100 GPU (Pascal generation) and the NVIDIA V100 GPU (Volta generation). Both architectures adhere to the traditional SIMT execution model. The NVIDIA V100 is part of the Volta generation where each thread has its own program counter. This allows threads to be scheduled independently; however, this feature comes at the price of each thread using two 32-bit registers for its program counter [19]. In Table 1 we list some of the key characteristics of the GPU architectures [18, 19]. In addition to the theoretical peak bandwidth listed by NVIDIA, we also report in

	P100	V100
Architecture	Pascal	Volta
DP Performance	5.3 TFLOPs	7 TFLOPs
SP Performance	10.6 TFLOPs	14 TFLOPs
HP Performance	21.2 TFLOPs	112 TFLOPs
SMs	56	80
Operating Freq.	1.15 GHz	1.53 GHz
Memory Capacity	16 GB	16 GB
Memory Bandwidth	732 GB/s	900 GB/s
Sustained BW	560 GB/s	846 GB/s
L2 Cache Size	4 MB	6 MB
L1 Cache Size	64 KB	128 KB

Table 1: Key characteristics of the high-end NVIDIA GPUs. The Half (HP) Performance of the V100 is for the Tensor cores. The sustained memory bandwidth is measured using the AXPY function from the CUDA cuBLAS library.

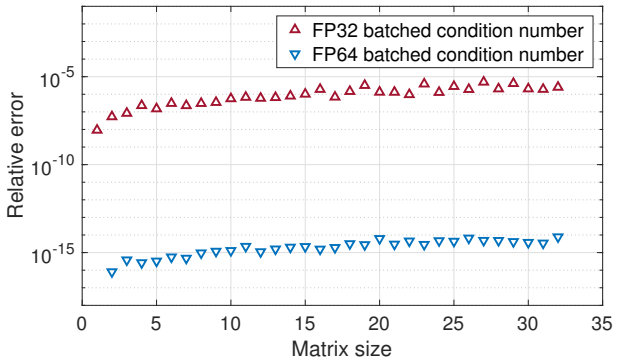


Figure 3: Relative error $\frac{\text{cond}(A)-cond_{MAT}(A)}{cond_{MAT}(A)}$ of the batched condition number calculation. The reference point $cond_{MAT}(A)$ is computed using MATLAB's COND function. The data is averaged over 100 matrices.

Table 1 the "sustained memory bandwidth" that we could attain using the CUBLASDAXPY function of NVIDIA's cuBLAS library.

Although all computations are executed on the GPU, we mention that the host system is powered by two Intel Xeon E5-2650 v3 (codename "Haswell") processors running at 2.30 GHz. We implemented the batched condition number routine in the MAGMA open source software framework [16] with the matrix inversion based on GJE with partial pivoting [8]. The kernel is implemented in the CUDA programming model, with CUDA version 9.0 used to compile and run the kernels. By default, we use a thread block size of 128. The batch of test matrices is generated with random entries.

In a first experiment we check the correctness of the developed batched condition number kernel by taking MATLAB's COND function as reference (MATLAB version 2017b [20]). In Figure 3 we visualize the relative error of the single and double precision versions (SP and DP, respectively) for increasing matrix sizes. The data there reflects the average relative error over 100 random matrices. The relative error stays within the approximation accuracy of the respective floating-point format for all matrix sizes.

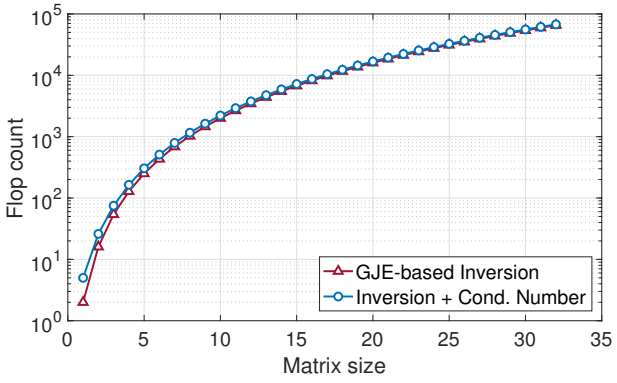


Figure 4: Floating-point operation count for inverting a single matrix via GJE vs. the variant computing also the condition number of the matrix.

	P100	V100
Registers		
Available / Block	65536	65536
Used / Thread	89	100
Used / Block	12288	13312
Derived maximum of active Blocks / SM	5	4

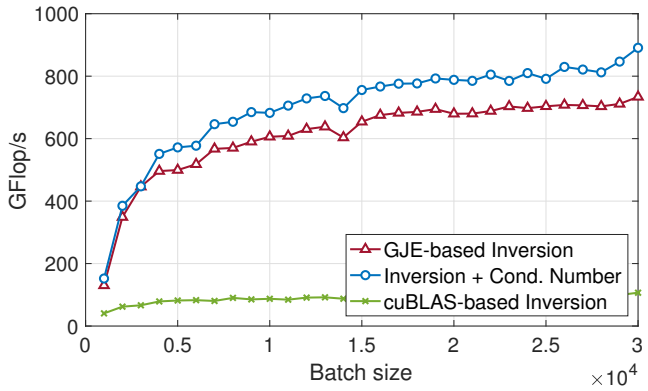
Table 2: Statistics on compiling and executing the DP variable-size batched condition number kernel on a batch of 50,000 matrices of size 32×32.

Next, we compare the performance of the variable-size batched condition number routine with the variable-size batched matrix inversion kernel based on GJE [8]. The motivation is that the batched condition number kernel presented in Section 3 has the GJE-based matrix inversion as a central component and computes the condition number by adding norm calculations prior to and after the matrix inversion. In the performance comparison we only count the operations of the matrix inversion, while the norm calculation and the condition number calculation are viewed as "overhead" to the matrix inversion. This approximation simplifies the analysis of the performance penalty introduced by adding the condition number calculation to the matrix inversion. Furthermore, it is reasonable as the additional cost (in terms of floating-point operations) of the condition number calculation quickly becomes negligible for increasing matrix sizes; see Figure 4.

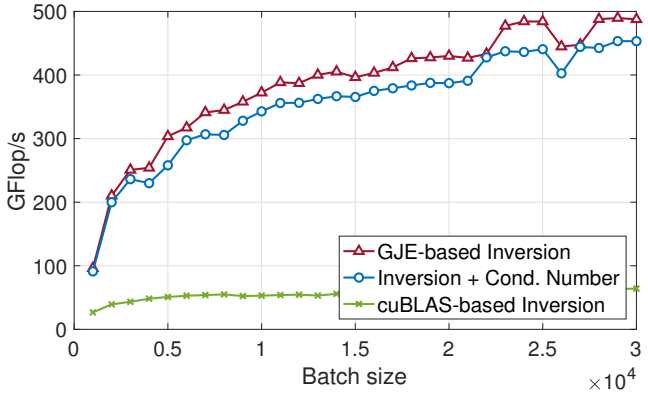
For reference, in the comparison we include the performance we achieve with the batched inversion routine available in NVIDIA's cuBLAS library.

In Figure 5 we focus on the P100 GPU and we consider a homogeneous batch containing square matrices of orders 16 (top row) and 32 (bottom row). The left-hand side figures are for IEEE SP, the right-hand side figures are for IEEE DP.

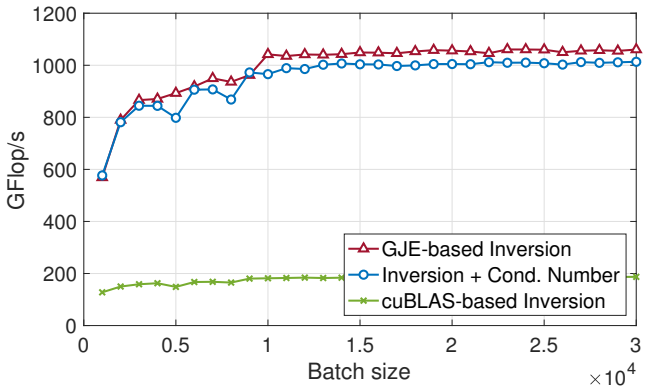
The analysis reveals that the performance of the batched condition number routine grows with the batch size following a similar trend to that observed for the GJE-based matrix inversion. The overhead of the condition number assessment ranges between 5%



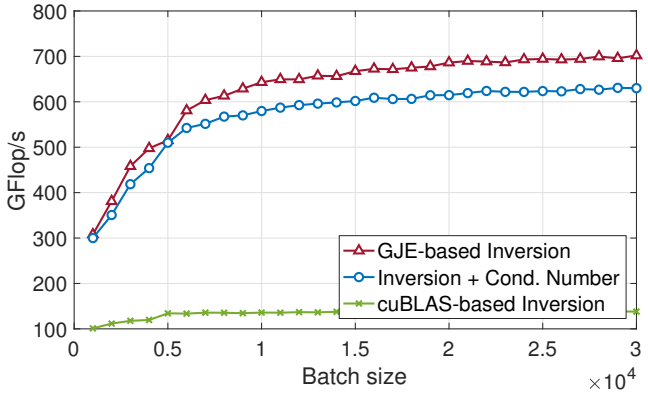
(a) Batch containing matrices of size 16×16, single precision.



(b) Batch containing matrices of size 16×16, double precision.

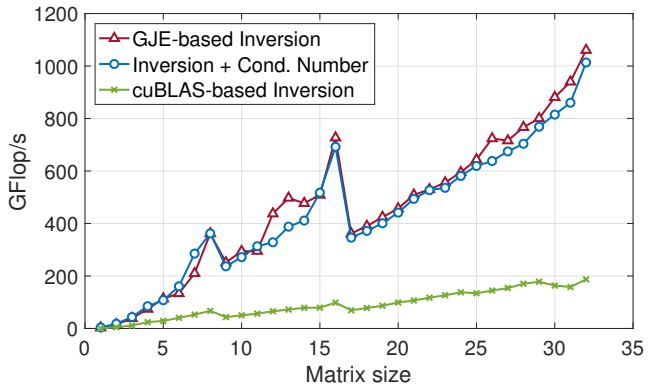


(c) Batch containing matrices of size 32×32, single precision.

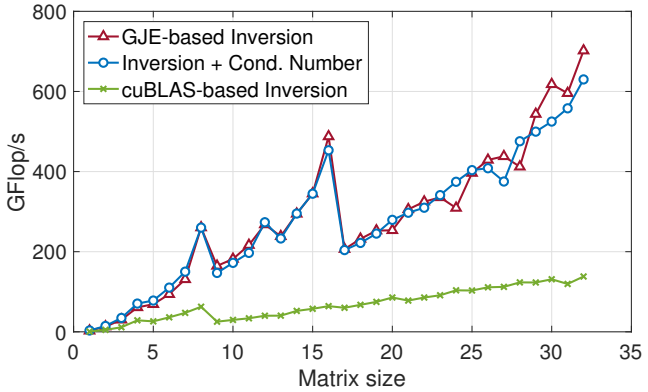


(d) Batch containing matrices of size 32×32, double precision.

Figure 5: Performance analysis of the variable-size batched condition number routine on P100 GPU in comparison to batched matrix inversion routines.

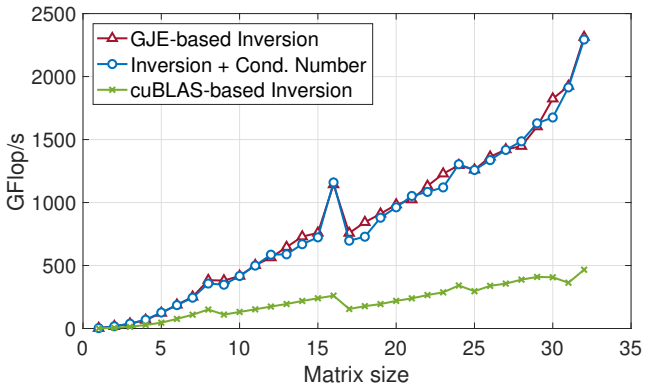


(a) Batches containing 30,000 matrices, single precision.

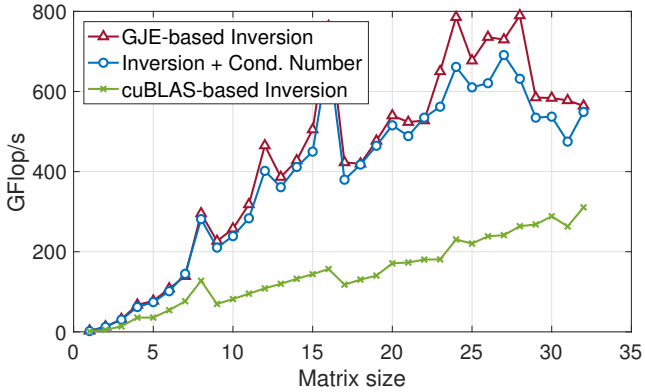


(b) Batches containing 30,000 matrices, double precision.

Figure 6: Performance of the variable-size batched condition number routine on P100 GPU for increasing matrix size and a fixed batch size of 30,000 matrices.



(a) Batches containing 30,000 matrices, single precision.



(b) Batches containing 30,000 matrices, double precision.

Figure 7: Performance of the variable-size batched condition number routine on V100 GPU for increasing matrix size and a fixed batch size of 30,000 matrices.

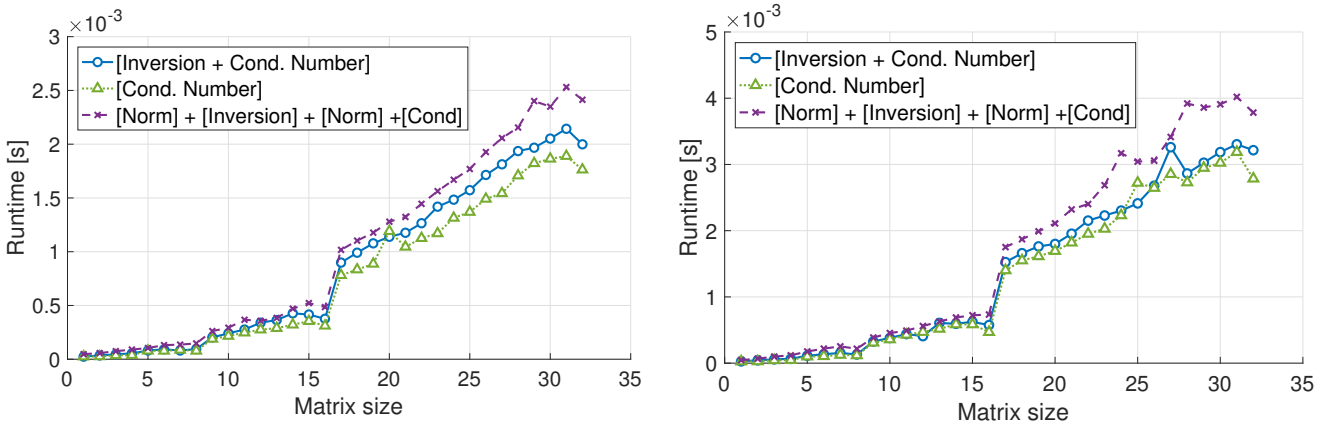
and 15%, depending on the floating-point format and the matrix size.

To further investigate the performance variation as a function of matrix size, in Figure 6 we fix the batch size to 30,000 matrices and then vary the size of the matrices in the batch. In the SP regime (left-hand side figure), the performance of the kernel inverting the matrices and computing the condition number increases consistently with the matrix size, with some local peaks marking sweet spots from the point of view of the strategy for multiple-problems-per-warp (for size 8 and 16). The peak performance for a batch of 32×32 matrices exceeds 1 TFLOPs (i.e., 10^{12} floating-point operations per second, or FLOPs). This is more than a 6× speedup over NVIDIA's batched inversion routine, which is neither capable of handling batches containing matrices of different size, nor does compute the condition number. Ignoring minor performance fluctuations, the same trend can be observed for the DP case. There, the performance peak is about 550 GFLOPs, which is half the SP performance.

We now run the same performance test on the newer V100 GPU, with the results visualized in Figure 7. The same local peaks can be observed for sizes 8 and 16. In the SP regime, the V100 behaves similarly to the P100, reaching a peak performance of 2.3 TFLOPs; see Figure 7b. However, the performance pattern is different in the DP case; see Figure 7b. Unlike in the P100 case, the performance does not grow for sizes beyond 24. Instead, the performance drops significantly for sizes larger than 23. To investigate this behavior in Table 2 we compare how the variable-size batched condition number kernel is compiled and executed in the 32×32 DP case. We notice that, on the V100, the kernel is compiled using additional registers. This, in the end, limits the number of blocks that can be executed simultaneously on a multiprocessor.

The design of the variable-size batched condition number routine is motivated by the idea of storing the inverted blocks of a block-Jacobi preconditioner in less than working precision [6]. This requires both the inversion of the diagonal blocks and the calculation of the condition number of each the blocks to ensure regularity. The kernel we developed combines both steps. At the same time, if only the condition number is of interest, writing the inverse matrices back to main memory is not required. We now turn back to the P100 architecture and assess the benefits obtained from dropping the writes to main memory from the batched condition number routine; see Figure 8. In this analysis we also include a variant that computes the condition number by launching four separate kernels for the building blocks outlined in Figure 1: A batched norm calculation; the GJE-based batched matrix inversion; a second batched norm calculation for the inverse matrix; and a kernel computing the condition number as the ratio between the norm of the matrix and the norm of its inverse. A pattern we notice for all runtime data in Figure 8 reflects the sweet-spots of the multiple-problems-per-warp strategy: For batches containing only matrices of dimension smaller than 16×16, we can handle multiple matrices with each warp. Once the problem size becomes larger than 16×16, the runtime increases drastically. Comparing the different realizations of the batched condition number calculation, we notice a 25%–30% higher execution time for the variant composed of four batched routines. This was expected, as launching four separate kernels significantly increases the data access volume. Completing the condition number calculation in registers without writing the inverse matrix to main memory is, in the SP case, about 10% faster than the combination of inversion and condition number calculation; see Figure 8a. In the DP case, the runtime benefits of avoiding the main memory writes are smaller; see Figure 8b.

For completeness, we include a performance analysis of the variable-size batched matrix infinity norm calculation as this routine may also be used as a stand-alone function. As this operation has an arithmetic intensity of $O(1)$ (concretely, $O(n^2)$ memory reads versus $O(n^2)$ floating-point operations for a matrix of size n) we assess the efficiency of the developed kernel by analyzing the achieved memory bandwidth. In Figure 9 we consider uniform batches containing matrices of order 16×16 (dashed lines) and 32×32 (solid lines), and increase the batch size from 1,000 to 50,000. The data reveals that the DP kernel reaches about 500 GB/s, which



(a) Runtime for batches containing 30,000 matrices, single precision. **(b) Runtime for batches containing 30,000 matrices, double precision.**

Figure 8: Runtime comparison (P100 GPU) of different kernel variants computing the condition number: [Inversion + Cond. Number] enhances the GJE-based matrix inversion with the condition number calculation; [Cond. Number] only computes the condition number using the same strategy, but does not write the inverse matrix back to memory; [Norm] + [Inversion] + [Norm] +[Cond] invokes four separate kernels to compute the condition number and the inverse.

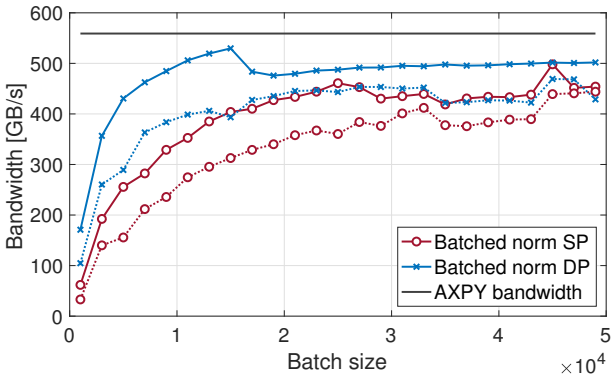


Figure 9: Bandwidth achieved for the variable-size batched infinity matrix norm. Solid lines are for matrices of size 32×32**; dashed lines for matrices of size** 16×16**.**

is 10% below the sustained bandwidth we attained with the cuBLAS-Daxpy routine from NVIDIA's cuBLAS library. For matrices where each row takes 128 bytes of memory (sizes 16×16 in DP or 32×32 in SP) we observe about 450 GB/s. In the SP case with matrices of dimension 16×16 the variable-size matrix infinity norm achieves about 75-80% of the measured peak bandwidth.

5 SUMMARY AND FUTURE WORK

We presented a variable-size batched condition number kernel for GPUs. The routine combines the matrix infinity norm calculation with matrix inversion via Gauss-Jordan elimination enhanced with implicit pivoting. The kernel keeps the data in registers only, and all communication is handled via warp shuffles. Following this strategy, we achieve performance rates of up to 1 SP TFLOPs and 500 DP GFLOPs when running the kernel on NVIDIA's P100 GPU architecture. In addition to the composed batched condition number routine, we present a variable-size batched matrix infinity norm. In a memory efficiency analysis, we observe that this memory-bound kernel achieves up to 90% of the sustained memory bandwidth.

All functionalities are designed to fit into the MAGMA-sparse open source software package. As part of future work, we plan to investigate how to efficiently integrate the batched condition number routine into the adaptive precision block-Jacobi preconditioning framework.

ACKNOWLEDGMENTS

This work was partly funded by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0010042. H. Anzt was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by projects TIN2014-53495-R of the Spanish *Ministerio de Economía y Competitividad* and the EU H2020 project 732631 OPRECOMP.

The authors want to acknowledge the access to the PizDaint supercomputer at the Swiss National Supercomputing Centre granted under the project #d65.

REFERENCES

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 21–38.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2017. Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3079079.3079103

[3] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2018. Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures. *Journal of Computational Science* (2018). https://doi.org/10.1016/j.jocs.2018.01.005

[4] Hartwig Anzt, Edmond Chow, Thomas Huckle, and J. Dongarra. 2016. Batched Generation of Incomplete Sparse Approximate Inverses on GPUs. In *Proceedings*

of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '16). 49–56.

[5] Hartwig Anzt, Gary Collins, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017. Flexible Batched Sparse Matrix-vector Product on GPUs. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '17)*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/3148226.3148230

[6] Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J. Higham, and EnriqueâĂŁS. Quintana-Ortí. 2018. Adaptive Precision in Block-Jacobi Preconditioning for Iterative Sparse Linear System Solvers. *Concurrency and Computation: Practice and Experience* (2018). https://doi.org/10.1002/cpe.4460

[7] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017. Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'17)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3026937.3026940

[8] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2018. Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors. *Parallel Comput.* (2018). https://doi.org/10.1016/j.parco.2017.12.006

[9] Ali Charara, David E. Keyes, and Hatem Ltaief. 2017. A framework for dense triangular matrix kernels on various manycore architectures. *Concurrency and Computation: Practice and Experience* 29, 15 (2017). https://doi.org/10.1002/cpe.4187

[10] A. K. Cline, C. B. Moler, G. W. Stewart, and J. H. Wilkinson. 1979. An Estimate for the Condition Number of a Matrix. *SIAM J. Numer. Anal.* 16, 2 (1979), 368–375. https://doi.org/10.1137/0716029 arXiv:https://doi.org/10.1137/0716029

[11] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, and Mawussi Zounon. 2017. Optimized Batched Linear Algebra for Modern Architectures. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 511–522.

[12] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.

[13] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack J. Dongarra. 2017. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2017, Denver, CO, USA, November 13, 2017*. 10:1–10:8. https://doi.org/10.1145/3148226.3148237

[14] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.).

[15] Nicholas J. Higham. 2014. Numerical Conditioning. In *Walter Gautschi. Selected Works with Commentaries*, Claude Brezinski and Ahmed Sameh (Eds.). Vol. 1. Birkhäuser, New York, 37–40. https://doi.org/10.1007/978-1-4614-7034-2_5

[16] Innovative Computing Lab. 2018. Software distribution of MAGMA version 2.3. http://icl.cs.utk.edu/magma/. (2018).

[17] NVIDIA Corp. [n. d.]. *CUDA C Programming Guide*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[18] NVIDIA Corp. 2016. Whitepaper: NVIDIA Tesla P100. (2016).

[19] NVIDIA Corp. 2017. Whitepaper: NVIDIA TESLA V100 GPU ARCHITECTURE. (2017).

[20] The MathWorks, Natick, MA, USA 2017b. *MATLAB Optimization Toolbox*. The MathWorks, Natick, MA, USA.