Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 1783–1792

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Variable-Size Batched Gauss-Huard for Block-Jacobi Preconditioning

Hartwig Anzt[1], Jack Dongarra[1,2,3], Goran Flegar[4], Enrique S. Quintana-Ortí[4], and Andrés E. Tomás[4]

[1] Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee, US
hanzt@icl.utk.edu, dongarra@icl.utk.edu
[2] Oak Ridge National Laboratory, USA
[3] School of Computer Science, University of Manchester, United Kingdom
[4] Depto. Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), Castellón, Spain
flegar@uji.es, quintana@uji.es, tomasan@uji.es

Abstract

In this work we present new kernels for the generation and application of block-Jacobi preconditioners that accelerate the iterative solution of sparse linear systems on graphics processing units (GPUs). Our approach departs from the conventional LU factorization and decomposes the diagonal blocks of the matrix using the Gauss-Huard method. When enhanced with column pivoting, this method is as stable as LU with partial/row pivoting. Due to extensive use of GPU registers and integration of implicit pivoting, our variable size batched Gauss-Huard implementation outperforms the batched version of LU factorization. In addition, the application kernel combines the conventional two-stage triangular solve procedure, consisting of a backward solve followed by a forward solve, into a single stage that performs both operations simultaneously.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: Sparse linear systems, iterative methods, block-Jacobi preconditioner, linear systems, Gauss-Huard factorization, Gauss-Jordan elimination, graphics processing units (GPUs)

1 Introduction

Iterative methods for the solution of sparse linear systems can strongly benefit from the integration of a preconditioner that is inexpensive to calculate and apply, and improves the convergence rate of the iterative scheme [14]. On a parallel system, the efficiency of the preconditioner also depends on how well these two building blocks, preconditioner calculation and application, can be formulated in terms of parallel algorithms.

Block-Jacobi preconditioners are more complex to handle than their (scalar) Jacobi counterparts, as they base the scaling on the inverse of the block diagonal. Nevertheless, the additional effort can be justified, as block-Jacobi preconditioners often provide faster convergence for problems that inherently carry a block structure.

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.186

The higher cost of block-Jacobi schemes comes from extracting the diagonal blocks from the coefficient matrix of the linear system, which is typically stored using a specialized data structure for sparse matrices, and the scaling with the inverse of this collection of small independent blocks. The latter can be realized by either explicit inversion in the preconditioner setup phase, or by generating LU factors that are then used in triangular solves during the preconditioner application (solve phase). Here, we note that parallelism in block-Jacobi preconditioners comes from the existence of multiple independent problems of small size.

In [3] we introduced a batched routine for GPUs that explicitly generates the block-inverse of a block-Jacobi preconditioner. Our implementation, based on Gauss-Jordan elimination (BGJE), 1) integrates an efficient scheme to extract the required matrix entries from the sparse data structures, 2) applies an implicit pivoting strategy during the inversion, and 3) computes the inverses using the GPU registers. As a result, our BGJE kernel clearly outperforms the batched LU-based factorization kernel in MAGMA [12].

In this paper we revisit the computation of block-Jacobi preconditioners on GPUs via variable size batched routines, making the following contributions:

- Our block-Jacobi preconditioner generation computes a triangular decomposition of the diagonal blocks, avoiding the computationally expensive and numerically dubious explicit generation of inverses.
- Instead of utilizing the conventional LU factorization for the decomposition, we rely on the Gauss-Huard (GH) algorithm [11]. The cost of this algorithm is analogous to that of the LU-based method, and much lower than the cost of an explicit inversion (such as GJE). Furthermore, when combined with *column* pivoting, GH offers a numerical stability that is comparable with that of the LU-based method with partial pivoting [9].
- In contrast to the batched LU factorization kernels in MAGMA, and the GPU parallel version of the GH algorithm in [6], we design our CUDA kernel to address the small problems of variable size arising in the computation of the block-Jacobi preconditioner by heavily exploiting the GPU registers. Furthermore, we reduce data movements by performing the column permutations (required for pivoting) implicitly.

2 Background and Related Work

Block-Jacobi preconditioning. The block-Jacobi method arises as a blocked variation of its (scalar) Jacobi counterpart that extends the idea of diagonal scaling to block-diagonal inversion with the diagonal blocks of A gathered into $D = \text{diag}(D_1, D_2, \ldots, D_N)$, $D_i \in \mathbb{R}^{m \times m}$, $i = 1, 2, \ldots, N$.

An important question in this context is how to choose the diagonal blocks. In the optimal case, these blocks should reflect the properties of the coefficient matrix A. Fortunately, many linear systems exhibit some inherent block structure. For example, if A comes from a finite element discretization of a partial differential equation (PDE), each element typically has multiple variables associated with it [7]. These variables are typically tightly coupled, leading to dense diagonal blocks. As all variables belong to the same element, they share the same column sparsity pattern. Such sets of variables are often referred to as *supervariables*. A popular strategy to determine the blocks for block-Jacobi is *supervariable blocking* [7], which is based on identifying variables sharing the same column-nonzero-pattern. Depending on the predefined upper bound for the size of the blocks, multiple supervariables adjacent in the coefficient matrix can be agglomerated within the same diagonal block. To help ensure that supervariables accumulated

into the same Jacobi block are coupled, the matrix should be ordered so that nearby variables are also close in the PDE mesh. This is satisfied for locality-preserving ordering techniques such as reverse Cuthill-McKee or natural orderings [7].

There exist different strategies of employing a block-Jacobi preconditioner in an iterative solver setting. One option is to explicitly compute the block-inverse before the iterative solution phase, and apply the preconditioner in terms of a matrix-vector multiplication. A second approach is to extract the diagonal blocks in the preconditioner setup phase, and solve one linear system per block within the preconditioner application. In-between these strategies falls the idea (explored in this paper) of factorizing the diagonal blocks in the setup, and performing small triangular solves in the preconditioner application. These three strategies differ in the workload size, and how this work is distributed between the preconditioner setup phase and the preconditioner application phase.

Solution of linear systems. The conventional procedure to solve a linear system with an $m \times m$ coefficient matrix D_i commences with the computation of the LU factorization (with partial pivoting) $P_i D_i = L_i U_i$, where L_i is unit lower triangular, U_i is upper triangular, and P_i is a permutation [10]. This is followed by the application of P_i to the right-hand side vector; and the solution of two triangular systems with L_i and U_i. Assuming a single right-hand side vector, this four-stage procedure requires $2m^3/3$ floating-point operations (flops) for a linear system of order m.

Gauss-Jordan elimination (GJE) is an efficient procedure for matrix inversion. In terms of theoretical cost and practical performance, GJE is competitive with the standard approach based on the LU factorization [13, 5]. However, if the goal is not the matrix inversion but retrieving the solution of a linear system, GJE incurs significant overhead, requiring $2m^3$ flops. The *Gauss-Huard* (GH) algorithm is a variant of GJE for the solution of linear systems with a computational cost consistent with that of the LU-based approach. Furthermore, the GH-based solver can be combined with a column-version of the standard partial (row) pivoting to offer a numerical stability that is comparable with that of the LU-based solver enhanced with partial pivoting [9]. Figure 1 illustrates a Matlab implementation of the GH solver for a system $D_i x_i = b_i$. Column-pivoting is applied to the coefficient matrix during the factorization, and undone in the final step of the algorithm.

In the GH algorithm we distinguish a "decomposition phase", operating exclusively on the matrix entries; and an "application phase", which transforms the right-hand side vector of the linear system into the sought-after solution. Although we realize that a GH decomposition does not provide a factorization in the classical LU interpretation, we use this term to distinguish the operations on the system matrix (in the preconditioner setup phase) from those on the right-hand side vector (in the preconditioner application).

GH with implicit pivoting. Pivoting can be a costly operation on parallel architectures, as this process involves two different memory access patterns. For example, column pivoting requires the selection of the largest element in a single row, and this is followed by an exchange of two columns at each iteration of the algorithm. If the matrix rows are distributed among the processors column-wise, the selection can be performed using a parallel reduction, while the column swap exposes little parallelism since all but the two processors involved in the swap remain idle. If the matrix is distributed row-wise the situation is reversed.

To tackle this problem for GJE, in [3] we introduced implicit (row) pivoting, eliminating the sequential part of the process by postponing the exchange step, and applying the swaps from all iterations at the end of the algorithm in parallel. In GJE, implicit (row) pivoting is enabled by noticing that its iterations are row-oblivious (i.e., the operations performed do not depend on the actual position of the row in the matrix, but only on the currently selected pivot row).

```
1  % Input  : m x m nonsingular matrix block Di, right-hand side bi
2  % Output : Di overwritten by the GH factorization, solution xi
3  p  = [1:m];
4  for k = 1 : m
5     % Row elimination. Matrix-vector product (GEMV)
6     Di(k,k:m)              = Di(k,k:m) - Di(k,1:k-1) * Di(1:k-1,k:m);
7     bi(k)                  = bi(k)     - Di(k,1:k-1) * bi(1:k-1);
8     % Column pivoting (explicit)
9     [abs_ipiv, ipiv]       = max(abs(Di(k,k:m)));
10    ipiv                   = ipiv+k-1;
11    [Di(:,ipiv), Di(:,k)] = swap(Di(:,ipiv), Di(:,k));
12    [p(ipiv),    p(k)]     = swap(p(ipiv),    p(k));
13    % Diagonalization. Vector scaling (SCAL)
14    Di(k,k+1:m)            =  Di(k,k+1:m) / Di(k,k);
15    bi(k)                  =  bi(k)       / Di(k,k);
16    % Column elimination. Outer product (GER)
17    Di(1:k-1,k+1:m)        = Di(1:k-1,k+1:m) - Di(1:k-1,k) * Di(k,k+1:m);
18    bi(1:k-1)              = bi(1:k-1)       - Di(1:k-1,k) * bi(k);
19 end
20 xi(p) = bi;
```

Figure 1: Simplified loop-body of the basic GH implementation in Matlab notation.

This is different for GH, as the operations performed on the columns do depend on the column position in the matrix. Precisely, iteration i updates only the columns with index greater than i. By noticing that all the columns with smaller index were already chosen as pivots in an earlier iteration, this requirement can be reformulated in a column-oblivious manner: iteration i updates only the columns that have not yet been chosen as pivot. This binary predicate still does not provide enough information to compute the GEMV on line 6 of Figure 1, as the order of elements in the input vector depends on the order of columns in the matrix: j-th value in the vector is the i-th element of the j-th column (i.e., the i-th element of the column chosen as pivot in the j-th iteration). Without exchanging the columns, this information can be propagated by maintaining a list of previous pivots.

This implicit pivoting strategy incurs some instruction and memory overhead, however insignificant to the savings coming from omitting explicit column swapping. We note that introducing implicit pivoting in GH preserves the numerical stability of the original algorithm.

Related work on batched routines. The term "batched" refers to a setting where a certain computational kernel is applied to a large set of (independent) data items [1]. The motivation for having a special design (and interface) of those routines is that applying them to a single data item does not fully utilize the hardware resources, so handling the distinct problems serially leaves them unused. The independence of the data items allows the application of the operations to multiple data items in parallel. In particular, for architectures providing a vast amount of hardware concurrency, such as GPUs, replacing standard algorithms with data-parallel implementations can result in significant performance gains [2].

For the previously mentioned GJE, we demonstrated in [3] how the inversion of a set of small linear systems can be realized efficiently on NVIDIA GPUs. There we also described how to combine the batched routine with data extraction and insertion steps to efficiently generate a block-inverse matrix for block-Jacobi preconditioning.

3 Design of CUDA Kernels

Utilizing GH for a block-Jacobi preconditioner requires the initial extraction of the diagonal blocks from the matrix A stored in the compressed sparse row (CSR) format (as required by MAGMA [12]). After this, the sequence of blocks is fed into a batched version of the GH algorithm (BGH), and the resulting decomposition, along with the pivoting information, is inserted back into the preconditioner matrix and the pivot vector, respectively. The preconditioner application step uses this information to apply BGH to the right-hand side vector, ultimately turning each vector block into the solution of the corresponding small linear system. Details of the distinct steps and their efficient realization on GPUs are described in this section.

Variable Size Batched Gauss-Huard decomposition. The BGH kernel applies the GH algorithm to factorize a set of small independent blocks. Following the kernel design in [3], the BGH implementation takes advantage of the large register count and warp shuffle instructions in recent GPU architectures. The GH decomposition commences by reading the diagonal block, with each thread storing a single column in registers. The actual computation is realized entirely in the registers, using warp shuffles for inter-thread communication. This approach eliminates the latency of memory and caches, as well as the load and store instructions, decreasing the complexity of the kernel. Column permutations required to perform the pivoting can be avoided by using implicit pivoting as described in Section 2. The application of the pivoting is delayed until the preconditioner matrix is inserted into the sparse data structure. An additional register array is used to store the pivoting information, and this array is replicated in each thread for quick access during the GEMV step.

The use of warp shuffles and registers limits the scope of the kernel to blocks of size up to 32, as the number of threads cannot exceed the warp size. Nevertheless, this covers the typical application scenario for block-Jacobi preconditioning [3].

Batched Gauss-Huard application. The solution of the linear system lacks any reuse of matrix elements. Hence, the kernel has to be designed with focus on optimizing memory access. Since the solution vector is needed in each step of the GH algorithm, it is read into registers in an interleaved pattern, with each thread storing one component of the vector.

Each outer loop iteration k of the GH application algorithm updates the k-th element of the solution vector with the dot product between the first $k-1$ elements of the k-th matrix row and the solution vector (Figure 1, line 7). This can be implemented as a parallel reduction using warp shuffles. After that, a parallel `AXPY` updates the first $k-1$ elements of the solution vector using the first $k-1$ elements of k-th column (Figure 1, line 17). To attain coalescent memory access for both operations, the matrix $\widetilde{D}_i$ can be decomposed into lower and upper parts: $\widetilde{D}_i = L_i + U_i$, with the diagonal belonging to the former. Matrix L_i is stored in row-major order to enable coalescent access to its rows and U_i in column-major order to provide fast access to the columns. Alternatively, matrix U_i can be transposed with respect to the anti-diagonal to convert its columns into rows while preserving its triangular structure. The resulting matrix $\hat{D}_i = L_i + U_i^{AT}$ is then stored in row-major order. In this manner, both, the rows of L_i and the columns of U_i (i.e. the rows of U_i^{AT}) can be accessed in coalescent manner.

The application process is completed by writing the solution vector back to memory, taking into account the permutation generated in the decomposition step.

Batched data extraction and insertion. The extraction of a diagonal block from the sparse coefficient matrix is similar to the "shared extraction" strategy in [3]. A notable difference, though, is that the block has to be distributed among the threads in column-wise fashion. As a result, the strategy can be implemented using only a fraction of the amount of shared memory.

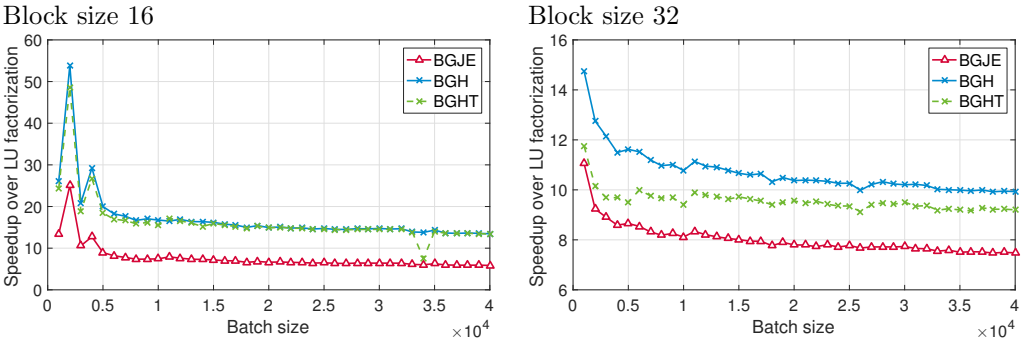

Figure 2: Speedup of BGJE and BGH / BGHT over BLU taken from the MAGMA software package.

Once the extraction of a single row to shared memory is completed, the elements of this row are immediately copied into the registers (one element per thread), making the shared memory locations available for the extraction of the next row. Thus, conversely to the shared memory extraction in [3], the shared memory has to hold only a single row of the diagonal block.

The insertion step writes the decomposed blocks $\widetilde{D}_i$ (or $\hat{D}_i$) back into the sparse matrix structure. Storing $\widetilde{D}_i$ is faster due to coalescent memory writes. At the same time, storing $\widetilde{D}_i$ results in noncoalescent memory access to the matrix U_i during the preconditioner application. Conversely, storing $\hat{D}_i$ moves the cost of noncoalescent memory access from preconditioner application to preconditioner setup. The noncoalescent memory accesses when writing $\hat{D}_i$ can be avoided by first preparing the matrix in shared memory, and then writing it to the main memory in coalescent fashion. However, this significantly increases shared memory consumption, decreasing the number of warps that can be scheduled per SM, and ultimately achieves lower performance. Consequently, we refrain from presenting performance results for this approach.

4 Numerical Experiments

We evaluate the block-Jacobi preconditioner based on GH by comparing the performance of the BGH implementation against kernels offering similar functionality. Furthermore, we assess the effectiveness of the arising preconditioner within an iterative solver setting. We emphasize that the implementations are part of the same software stack, and that the kernel implementations are similar in design, and received the same level of tuning. This ensures a fair comparison and conclusions with fair credibility.

Hardware and software framework. We use an NVIDIA Tesla P100 GPU with full double precision support. We employ NVIDIA's GPU compilers that are shipped with CUDA toolkit 8.0. All kernels are implemented using the CUDA programming model and are designed to integrate into the MAGMA-sparse library [12]. MAGMA-sparse is also leveraged to provide a testing environment, the block-pattern generation, and the sparse solvers. All computations use double precision arithmetic, as this is the standard in scientific computations. Since the complete algorithm is executed on the GPU, the details of the CPU are not relevant for the following experimentation.

Performance of BGH. Figure 2 compares the performance of our BGH implementation

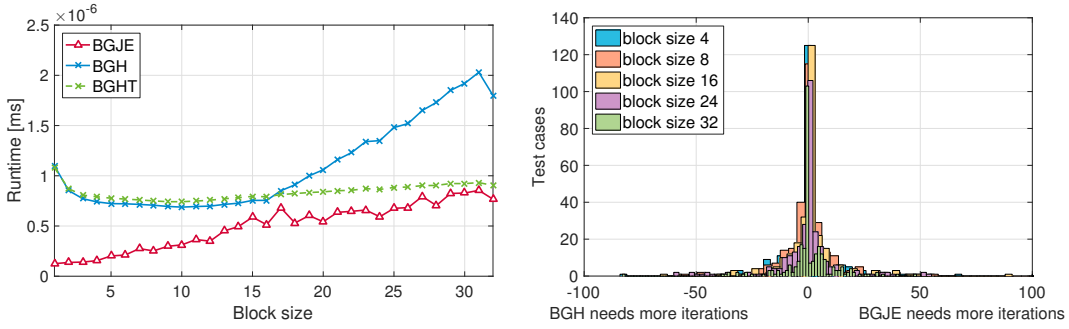

Figure 3: Left: Runtime of block-Jacobi application for distinct block sizes and a problem of row size 1,000,000. Right: BiCGSTAB convergence variations when using block-Jacobi based on BGJE or BGH.

with alternative approaches providing similar functionality. The baseline implementation is the batched LU factorization (BLU) kernel provided in the MAGMA library (version 2.0 [12]), designed for the LU factorization of a large set of small problems. We note that, conversely to the BGJE and BGH kernels, the scope of the BLU kernel is limited to settings where all small systems are of the same same size. The results in the figure are expressed in terms of speedup over this routine. BGJE is the implementation proposed in [3] which explicitly generates a block-inverse for Jacobi preconditioning. For GH, we also include data for the variant storing the upper triangular part transposed (BGHT), allowing for faster access during the preconditioner application. As BLU currently only supports batches of equal-size problems, while BGJE and BGH only work for problems of order up to 32, we limit the analysis to block sizes 16 and 32.

For block size 16 (see left plot in Figure 2) and large batch counts, the BGJE kernel is about 6× faster than BLU; and we observe even larger speedups for smaller batch sizes. Adding the transposed storage to the BGH kernel has a minor impact: for relevant batch sizes, both BGH and BGHT are 12–20× faster than BLU. This is different for block size 32 (see right-hand side plot in Figure 2): Adding the transposed storage to the BGH kernel reduces the speedup of BGHT over BLU to values around 8.5×, but BGH remains more than 10× faster than BLU. Even though BGJE computes the explicit inverse, and hence executes more operations than BLU and GH, this kernel is more than 7.5× faster than BLU. For completeness, we mention that for block size 32, BGJE delivers about 600 GFLOPS (billions of flops/second) on this architecture (see [3]).

Performance of block-Jacobi application. Figure 3 (left) shows the runtime of three preconditioner application strategies: sparse matrix-vector multiplication if the preconditioner was generated using GJE (BGJE), versus Gauss-Huard application using diagonal blocks $\widetilde{D}_i$ (BGH) or $\hat{D}_i$ (BGHT). We fix the problem size to 1 million rows and consider different diagonal block sizes. As could be expected, the approach based on the sparse matrix-vector product is always faster than both GH-based application strategies, which can be attributed to the lower number of flops and better workload balance.

Even though half of the memory accesses in BGH are noncoalescent, we observe only minor performance deviations from the BGHT kernel for block sizes smaller than 16. An explanation is that smaller blocks fit into less cache lines, so they can be read only once, and kept in cache for the duration of the kernel. If this happens, the noncoalescent reads are as fast as the coalescent

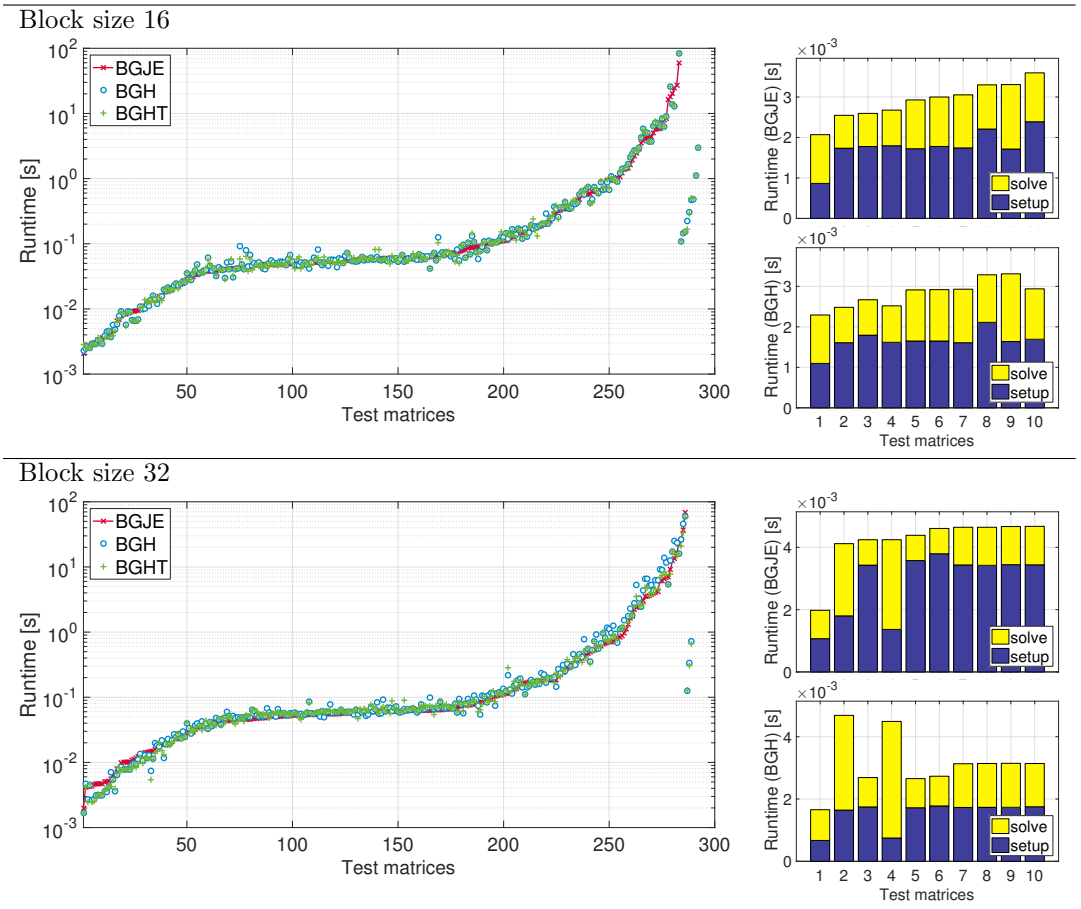

Figure 4: Left: Total execution time (setup+solve) for BiCGSTAB enhanced with block-Jacobi preconditioning based on either BGJE, BGH or BGHT. Top-level results are for a block-size bound 16; bottom-level results are for a block-size bound 32. Right: Decomposition of the total execution time into preconditioner setup time and iterative solver runtime for the left-most test cases.

ones, and both strategies result in the same performance. Once the blocks become too large for cache, some cache lines need to be evicted during the GH application, and noncoalescent reads in BGH start to impact the performance of this approach.

Iterative solver analysis. We next assess the efficiency of the distinct strategies for block-Jacobi preconditioning in an iterative solver framework. For this purpose we integrate the block-Jacobi preconditioner(s) into the BiCGSTAB iterative solver provided in MAGMA-sparse [4], and test the preconditioned solver setting for a variety of linear systems. The test matrices are chosen from the SuiteSparse matrix collection [8], combined with a right-hand side vector with all entries equal to one. We start the iterative solver with an initial guess of zero, and stop once the relative residual norm is decreased by six orders of magnitude. We allow for up to 10,000 iterations.

First, we evaluate whether the difference between GJE and GH in terms of numerical stability has any impact on the preconditioner efficiency. At this point, we recognize that rounding can have significant effect on a preconditioner's efficiency, and a more accurate preconditioner does not inevitably result in faster convergence of the iterative solver. Figure 3 (right) displays the convergence difference of BiCGSTAB depending on whether the block-Jacobi preconditioner is based on GH or GJE. The x-axis of the histogram reflects the iteration overhead; the y-axis shows the number of test cases for which GJE provided a "better" preconditioner (bars left of center) or GH did (bars right of center). For all block sizes, the majority of the problems is located in the center, reflecting the cases where both methods resulted in the same iteration count. Furthermore, the histogram exposes a high level of symmetry, suggesting that the numerical stability of the method based on explicit inversion plays a minor role.

While being similar from the convergence rate point of view, the pending question is whether there exist any performance differences making GJE or GH superior. The plot in the left-hand side of Figure 4 arranges the test systems according to increasing execution time of the BiCGSTAB solver preconditioned with block-Jacobi based on BGJE. The block structure was generated via the supervariable blocking routine provided by MAGMA-sparse with a maximum block size of 16 (top) and 32 (bottom). The execution times comprise both the preconditioner setup and the iterative solver times. In addition to the block-Jacobi using BGJE, we also include the total solver runtime for the variants using BGH and BGHT.

In most cases, the BGH and BGHT execution times are close, or even match those of BGJE. In particular for block size 32, the results may suggest that BGJE is slightly better if the execution time is large (right-most data). Conversely, BGH and BGHT are faster if the solution time is small (left-most data). On the right of Figure 4 we decompose the total solution time into its preconditioner setup and iterative solver components for the first 10 test problems. For these instances, the preconditioner setup time accounts for a significant portion of the total solver execution time, and the higher cost of explicit block-inversion is not compensated for by the slightly faster preconditioner application.

5 Concluding Remarks

We have designed data-parallel GPU kernels for the efficient generation and application of block-Jacobi preconditioners, based on the Gauss-Huard method, which can be embedded into any Krylov-based iterative solver for sparse linear systems. Our kernels exploit the intrinsic parallelism in the two algorithmic steps, preconditioner generation and preconditioner application. In contrast, our kernel implementation is specifically designed for small block sizes, exploiting the GPU registers to outperform their MAGMA counterparts by a large margin. Furthermore, our variable size batched Gauss-Huard kernel integrates an implicit version of column pivoting to eliminate costly data movements due to column permutations, while delivering the same numerical stability. Compared with block-Jacobi based on our register-tuned batched Gauss-Jordan elimination, which is also designed for small block sizes, the variable size batched Gauss-Huard kernels offer faster preconditioner generation and higher numerical stability, but slower preconditioner application. Our experimental results, using a state-of-the-art NVIDIA's P100 GPU, are consistent with this analysis. Implementing the block-Jacobi preconditioner on top of the batched Gauss-Huard provides better performance if the iterative solver converges within a small number of iterations. In these cases the cost of the preconditioner generation is considerable compared with the whole iterative solve (including the preconditioner application).

Acknowledgments

This material is supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award #DE-SC-0010042. The researchers from UJI were supported by project TIN2014-53495-R of MINECO and FEDER.

References

[1] A. Abdelfattah, H. Anzt, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan. Linear algebra software for large-scale accelerated multicore computing. *Acta Numerica*, 25:1–160, 5 2016.

[2] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance tuning and optimization techniques of fixed and variable size batched Cholesky factorization on GPUs. *Procedia Computer Science*, 80:119–130, 2016. ICCS 2016.

[3] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *8th Int. Workshop Programming Models & Appl. for Multicores & Manycores*, PMAM, pages 1–10, 2017.

[4] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Koehler. Efficiency of General Krylov Methods on GPUs – An Experimental Study. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 683–691, 2016.

[5] P. Benner, P. Ezzatti, E. Quintana-Ortí, and A. Remón. Matrix inversion on CPU-GPU platforms with applications in control theory. *Concurrency and Computation: Practice and Experience*, 25(8):1170–1182, 2013.

[6] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón. Revisiting the Gauss-Huard algorithm for the solution of linear systems on graphics accelerators. In *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015*, pages 505–514, 2016.

[7] E. Chow and J. Scott. On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning. Technical Report Technical Report RAL-P-2016-006, Rutherford Appleton Laboratory, 2016.

[8] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. on Mathematical Software*, 38(1):1–25, 2011.

[9] T. J. Dekker, W. Hoffmann, and K. Potma. Stability of the Gauss-Huard algorithm with partial pivoting. *Computing*, 58:225–244, 1997.

[10] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[11] P. Huard. La méthode simplex sans inverse explicite. *EDB Bull, Direction Études Rech. Sér. C Math. Inform. 2*, pages 79–98, 1979.

[12] Innovative Computing Lab. Software distribution of MAGMA version 2.0. `http://icl.cs.utk.edu/magma/`, 2016.

[13] E. S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn. A note on parallel matrix inversion. *SIAM J. Scientific Computing*, 22(5):1762–1771, 2001.

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.