# Structured Orthogonal Inversion of Block $p$-Cyclic Matrices on Multicore with GPU Accelerators ⋆

Sergiy Gogolenko[1], Zhaojun Bai[2], and Richard Scalettar[2]

[1] Donetsk National Technical University, Donetsk, 83001, Ukraine
sergiy.gogolenko@gmail.com
[2] University of California, Davis, CA 95616, USA
{bai@cs,scalettar@physics}.ucdavis.edu

**Abstract.** We present a block structured orthogonal factorization (BSOF) algorithm and its parallelization for computing the inversion of block $p$-cyclic matrices. We aim at the high performance on multicores with GPU accelerators. We provide a quantitative performance model for optimal host-device load balance, and validate the model through numerical tests. Benchmarking results show that the parallel BSOF based inversion algorithm attains up to 90% of `DGEMM` performance on hybrid CPU+GPU systems.

**Keywords:** $p$-cyclic matrix; matrix inversion; structured orthogonal factorization; performance modelling; GPU acceleration

## 1 Introduction

Since the pioneering works of Varga, Young, Romanovsky, and others in the 1950s, $p$-cyclic matrices have been found to be a very useful class of structured matrices with applications in numerical solutions of differential equations, Markov chain modeling and quantum Monte Carlo simulations. The concept of block $p$-cyclic matrices in its modern term is referred to matrices which can be transformed to the following *normalized block p-cyclic form* by row and/or column permutations:

$$H = \begin{bmatrix} A_1 & & & B_p \\ B_1 & A_2 & & \\ & B_2 & A_3 & \\ & & \ddots & \ddots \\ & & & B_{p-1} \; A_p \end{bmatrix}, \tag{1}$$

⋆ This work was supported by the National Science Foundation under grant NSF-PHY-1005503. SG would like to thank the Fulbright Program Office in Ukraine and the Institute of International Education for finantial support during this study. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

where $A_i$ and $B_i$ are non-zero blocks. For the sake of simplicity, in this paper, we are concerned entirely with the normalized block $p$-cyclic matrices, and furthermore, we assume that $A_i$ and $B_i$ are $n$-by-$n$ square blocks, although in some applications $A_i$ and $B_i$ are rectangular. The fact that we discuss only matrices with the square blocks $A_i$ and $B_i$ does not limit the generality of approaches presented in this paper.

The early studies of $p$-cyclic matrices were closely related to numerical solution of differential equations [3,9,10]. In these applications, the $p$-cyclic matrices are also referred to as bordered almost block diagonal (BABD) matrices. An incomplete list of BABD-based numerical algorithms includes multiple shooting and finite difference schemes for two-point boundary value problems (BVPs), orthogonal spline collocation methods for separable elliptic BVPs, method of lines and Keller's box scheme for various initial BVPs [3, 9]. The $p$-cyclic matrices also appear in Markov chains modeling, where the $p$-cyclic stochastic matrices represent infinitesimal generators of continuous-time Markov chains with periodic transition graphs for queuing networks and stochastic Petri nets [2]. In quantum Monte Carlo (QMC) simulations of Hubbard model for strongly correlated materials, the inverses of $p$-cyclic matrices, referred to as Green's functions, are required to be repeatedly computed *explicitly* for physical observables, see [1, 6] and references therein. Other sources of applications of $p$-cyclic matrices include some linear least-square problems and parameter estimation with non-linear DAE models.

In contrast to the subject of solving block $p$-cyclic linear systems, where we observe tremendous progress over the last six decades, the problem of computing $p$-cyclic matrix inversion explicitly remains in a state of infancy. The recent advances are mainly related to computing some particular blocks in the inverse of a $p$-cyclic matrix using well-known explicit expressions [1,6]. For instance, the paper [6] addresses stabilized algorithms for calculation of diagonal blocks of the inverse of block $p$-cyclic matrices. To the best of our knowledge, there are no previous work focused on numerical algorithms for the entire inversion of block $p$-cyclic matrices, which is required for time-dependent physical measurements in the quantum Monte Carlo simulation [1]. Filling this gap is the main purpose of our paper.

In this paper, we pay particular attention to algorithmic solutions designed specifically for high performance computing on GPU accelerated multicore systems. We should point out that numerical libraries for GPGPU computing, including widely used CUSPARSE, CULA, PARALUTION, and CUSP, do not support structure matrix inversion. Furthermore, solvers in dense linear algebra libraries for GPUs such as CUBLAS, MAGMA [7], and CULA, do not implement mechanisms for avoiding redundant computations with zero-blocks.

## 2 Previous work

Historically, the studies of $p$-cyclic matrices were primarily focused on iterative and direct methods for $p$-cyclic linear systems. The vast literature on iterative

methods cover in details successive overrelaxation, aggregation and disaggregation, Chebyshev semi-iterative, and Krylov subspace methods [2]. On the other hand, the attention to the direct solvers is also remarkable. Researchers explored numerous variations of Gaussian elimination and orthogonal factorization approaches for solving $p$-cyclic systems. There is a large volume of literature on Gaussian elimination dealing with a special case of $p$-cyclic systems, called almost block diagonal (ABD) systems [9]. Nevertheless, while handling the ABD systems successfully, Gaussian elimination processes could fail. In fact, in [10], it is shown that the Gaussian elimination with row partial pivoting produces exponential error growth for $p$-cyclic systems arising from multiple shooting for some linear BVPs with mixed two-point boundary conditions. There is a number of approaches that enlarge the class of linear systems for which numerical stability is ensured, such as certain forms of pre-scaling and replacing row-by-row pivoting with more accurate panel pivoting strategies. In the recent paper [5] Khabou et al. propose to use a panel rank-revealing pivoting strategy based on strong rank revealing QR, which significantly reduces the growth factor, and thus results in practical stability of Gaussian elimination in most cases.

Due to numerical stability issues of Gaussian elimination algorithms, Wright proposed to use a structured orthogonal factorization (SOF) [9]. He described a serial and two parallel block SOF algorithms. The first parallel algorithm uses a recursive factorization process similar to cyclic reduction, whereas the second one factorizes $p$-cyclic matrix in two steps, at first splitting the entire matrix on parts and factorizing these parts concurrently, and then performing factorization of the reduced $p$-cyclic matrix formed from border blocks of the parts factorized in the previous step. A proof of the stability of SOF is presented in [9].

## 3 Basic algorithms

This section gives a brief overview of an algorithm for structure-exploiting orthogonal inversion of block $p$-cyclic matrices. It is referred to as BSOFI. For more details of the BSOFI and its modifications such as blocking and batching, we refer readers to our technical report [4].

The algorithmic framework of BSOFI is composed of three phases. The first one is the block SOF of $H$: $H = QR$. Once factors $Q$ and $R$ are computed, the inverse is calculated by the identity $H^{-1} = R^{-1}Q^T$ in two phases, namely inversion of the factor $R$ and applying the transpose of the factor $Q$.

*Block structured orthogonal factorization (BSOF)* is a block structured QR factorization algorithm introduced in [9], and has an identical complexity to the best known block structured Gaussian elimination based algorithm. The essence of this algorithm is in transformation of the matrix $H$ through a sequence of $p-1$ block row updates (Fig. 1).

This reduction process results in the factorization $H = QR$, where $Q$ is a product of the orthogonal $2n$-by-$2n$ matrices $Q^{(k)}$ extended by identity blocks:

$$Q = \prod_{k=1}^{p-1} Q_k = \prod_{k=1}^{p-1} I_{n(k-1)} \oplus Q^{(k)} \oplus I_{n(p-k-1)}, \quad Q^{(k)} = \begin{bmatrix} Q_{11}^{(k)} & Q_{12}^{(k)} \\ Q_{21}^{(k)} & Q_{22}^{(k)} \end{bmatrix},$$

**Data**: $H$, $n$, $p$
**Result**: $R$, $\{Q^{(k)} | 1 \leq k < p-1\}$

1 $R \leftarrow O$; $\tilde{A}_1 \leftarrow A_1$ ; $\tilde{B}_1 \leftarrow B_p$;
2 **for** $k \in \{1, 2, ..., p-2\}$ **do**
3 Compute the QR: $Q^{(k)} \begin{bmatrix} R_{kk} \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{A}_k \\ B_k \end{bmatrix}$;
4 Update $\begin{bmatrix} R_{k,k+1} & R_{k,p} \\ \tilde{A}_{k+1} & \tilde{B}_{k+1} \end{bmatrix} \leftarrow \left(Q^{(k)}\right)^T \begin{bmatrix} 0 & \tilde{B}_k \\ A_{k+1} & 0 \end{bmatrix}$
5 Compute the QR: $Q^{(p-1)} \begin{bmatrix} R_{p-1,p-1} & R_{p-1,p} \\ 0 & R_{p,p} \end{bmatrix} = \begin{bmatrix} \tilde{A}_{p-1} & \tilde{B}_{p-1} \\ B_{p-1} & A_{p,p} \end{bmatrix}$

Fig. 1: `BSOF` – Wright's serial version of SOF algorithm.

and $R$ has block upper bidiagonal form with the last block column:

$$R = \begin{bmatrix} R_{11} & R_{12} & & & R_{1,p} \\ & R_{22} & R_{23} & & R_{2,p} \\ & & \ddots & \ddots & \vdots \\ & & & R_{p-1,p-1} & R_{p-1,p} \\ & & & & R_{p,p} \end{bmatrix}. \tag{2}$$

*Inversion of matrix $R$ via block back substitution* is the second phase. The inverse $X = R^{-1}$ is block upper triangular, and its non-zero blocks can be computed via block back substitution (BBS). We show a row version of the BBS in Fig. 2a by taking into account the zero-blocks of $R$, while solving the matrix equation $RX = I$ for $X$. Likewise, the column version of the BBS algorithm is based on solving $XR = I$.

Both versions of the BBS have their virtues and flaws. The columnwise BBS requires two times more floating point operations (flops) compared to its row version. On the other hand, we are able to perform SOF and the column version in parallel, which allows to overcome lack of parallelism in the factorization phase (see Fig. 1). In contrast, the latter is impossible in the row version.

*Applying the orthogonal factor $Q^T$ to $R^{-1}$* is the last phase. Due to the orthogonality of $Q^{(k)}$, the inverse of $Q$ is equal to

$$Q^{-1} = Q^T = \prod_{k=1}^{p-1} Q_{p-k}^T = \prod_{k=1}^{p-1} I_{n(p-k-1)} \oplus \left(Q^{(p-k)}\right)^T \oplus I_{n(k-1)}. \tag{3}$$

Thus, computing product $R^{-1}Q^T$ is equivalent to applying Householder reflectors of $\left(Q^{(k)}\right)^T$ to the pairs of column panels of $R^{-1}$ from right in a backward order, as shown in Fig. 3a. This is the gist of the last phase of BSOFI.

If matrices $Q^{(k)}$ are given in an explicit form, we benefit from the upper triangular structure of matrix $R^{-1}$ by means of replacing line 2 in Fig. 3a by the

[3] Batched denotes group of kernels that can be implemented in a single batched run.

**Data**: $R$, $n$, $p$
**Result**: $X$

1 $X \leftarrow O$;
2 $X_{p-2:p,p-2:p} \leftarrow R^{-1}_{p-2:p,p-2:p}$;
3 `Batched` $_{i=1:p-3}$ $\{X_{ii} \leftarrow R^{-1}_{ii}\}$ ;
4 $X_{1:p-3,p} \leftarrow R_{1:p-3,p} \cdot X_{p,p}$ ;
5 `Batched` $_{i=1:p-3}$
$\{X_{i,p} \leftarrow -X_{ii} \cdot R_{i,p}$,
$X_{i,i+1} \leftarrow -X_{ii} \cdot R_{i,i+1}$ $\}$ ;
6 **for** $i \in \{p-3, p-4, ..., 1\}$ **do**
7 $\quad X_{i,i+2:p} \leftarrow$
$\quad X_{i,i+2:p} + X_{i,i+1} \cdot X_{i+1,i+2:p}$ ;
8 $\quad X_{i,i+1} \leftarrow X_{i,i+1} \cdot X_{i+1,i+1}$;

(a) `BSTRI_RV` – Row Version of the BBS

**Data**: $R$, $n$, $p$
**Result**: $X$

1 $X \leftarrow O$;
2 `Batched` $_{j=3:p}$ $\{X_{jj} \leftarrow R^{-1}_{jj}\}$;
3 `Batched` $_{j=3:p-1}$
$\{X_{j-1,j} \leftarrow -R_{j-1,j} X_{jj}\}$;
4 $X_{1:2,1:2} \leftarrow R^{-1}_{1:2,1:2}$; $X_{1,p} \leftarrow X_{11} X_{1,p}$;
5 **for** $j \in \{3, ..., p-1\}$ **do**
6 $\quad$ `Batched` $\{X_{1:j-1,j} \leftarrow X_{1:j-1,j-1} X_{jj}$,
$\quad X_{1:j-1,p} \leftarrow$
$\quad X_{1:j-1,p} + X_{1:j-1,j-1} R_{j-1,p}\}$
7 $X_{1:p-1,p} \leftarrow X_{1:p-1,p} + X_{1:p-1,p-1} X_{p-1,p}$;
8 $X_{1:p-1,p} \leftarrow -R_{1:p-1,p} X_{p,p}$;

(b) `BSTRI_CV` – Column Version of the BBS

Fig. 2: Inversion of matrix $R$ via block back substitution.[3]

**Data**: $X$, $\{Q^{(k)} | 1 \le k < p-1\}$, $n$, $p$
**Result**: $X$

1 **for** $k \in \{p-1, p-2, ..., 1\}$ **do**
2 $\quad X_{1:p,k:k+1} \leftarrow X_{1:p,k:k+1} Q^{(k)T}$

(a) `BSOI` – Update $X$ via applying $Q^T$

**Data**: $X_{1:p,k:k+1}$, $Q^{(k)}$, $n$, $p$
**Result**: $X_{1:p,k:k+1}$

1 $W_{1:k+1,k:k+1} \leftarrow X_{1:k+1,k:k+1} Q^{(k)T}$;
2 $W_{k+2:p,k:k+1} \leftarrow X_{k+2:p,k:k+1} Q^{(k)T}_{1:2,2}$;
3 $X_{1:p,k:k+1} \leftarrow W$;

(b) `BSOI_Qk` – Applying $Q^T_k$

Fig. 3: Applying the orthogonal factors (Householder reflectors) to $R^{-1}$.

algorithm `BSOI_Qk` from Fig. 3b. This simple modification allows to reduce the number of flops in the algorithm shown in Fig. 3a from $8n^3p(p-1)$ to $2n^3(3p^2 - p - 4)$. Note that complete reconstruction of matrices $Q^{(k)}$ from Householder reflectors requires $O(n^3p)$ extra flops.

Computational complexity of the BSOFI algorithms is shown in Table 1. If `BSTRI_RV` is used, the total flops is $\Theta(7nN^2)$, where $N = n \times p$. This is roughly just two times more than the minimum flops count $\Theta(\frac{7}{2}nN^2)$ for the unstable Gaussian elimination based inversion without pivoting.

## 4 Parallel implementation on multicore with GPU accelerators

[4] The lower order terms are omitted for the sake of simplicity. More accurate formulae are presented in [4].

Table 1: Operation counts for the three phases of BSOFI algorithm.[4]

| Phase | Routine | Additions | Multiplications | Total Flops |
|---|---|---|---|---|
| I | `BSOF` | $\frac{1}{6}n^2\,(46np - 60n + 15p)$ | $\frac{1}{6}n^2\,(46np - 60n + 39p)$ | $\frac{1}{3}n^2\,(46np - 60n + 27p)$ |
| II | `BSTRI_RV` | $\frac{1}{6}n^3\,(3p^2 + 7p - 21)$ | $\frac{1}{6}n^3\,(3p^2 + 7p - 21)$ | $\frac{1}{3}n^3\,(3p^2 + 7p - 21)$ |
| | `BSTRI_CV` | $\frac{1}{6}n^3\,(6p^2 - 11p + 12)$ | $\frac{1}{6}n^3\,(6p^2 - 11p + 12)$ | $\frac{1}{3}n^3\,(6p^2 - 11p + 12)$ |
| III | `BSOI` | $n^2p\,(3np - 2n + p)$ | $n^2p\,(3np - 2n + p)$ | $2n^2p\,(3np - 2n + p)$ |

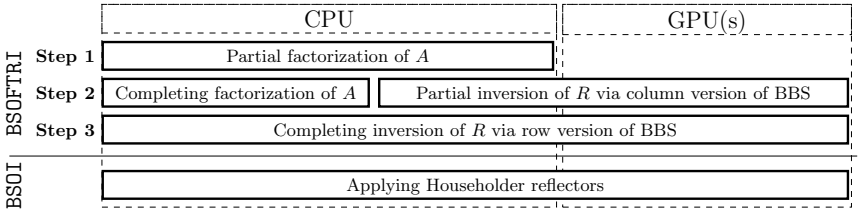



Fig. 4: Framework of the BSOFI adopted to execution on hybrid CPU+GPU platforms.

*Parallel "host-device" BSOFI algorithm.* We design our parallel "host-device" algorithm in a way to maximally benefit through extensive use of well optimized vendor-specific linear algebra kernels. The latter implies paying attention to the limited choice of batched linear algebra kernels for GPUs and the diversity in the kernel's performance on throughput and latency oriented processors.

Specifically, our design is inspired by the following well-known observation. The performance efficiency highly varies for different numerical kernels, and the matrix-matrix multiplication routine `DGEMM` tends to be the most efficient among other BLAS/LAPACK kernels. Furthermore, performance gaps between `DGEMM` and other kernels are usually much lower for latency oriented processors compared to the throughput oriented ones. At the same time, in both cases, the gaps become smaller as the size of the problem grows. Hence, to attain better performance of hybrid CPU+GPU algorithm, it is preferable to exploit the throughput oriented GPU accelerators only for `DGEMM` and, conversely, to use the latency oriented CPUs for the whole variety of required kernels. In addition, such work distribution strategy lets to avoid those LAPACK kernels for GPU platforms, which require CPU resources, and thus may interfere with pure CPU kernels executed in parallel. Specifically, following the recipes given in [8], QR factorization routines from the state-of-the-art LAPack API implementations for GPUs, such as Magma [7] and CuLA, usually use an approach, where column panels are factorized on CPU and afterwards sent to GPU for trailing matrix update.

Since a vast part of computations in the BSOFI is spent on `DGEMM`, this algorithm has a great potential to be reorganized in accordance with the work distribution strategy discussed above. The necessary modifications are sketched in Fig. 4. To overcome the lack of parallelism and `DGEMM` operations in the fac-

torization phase, we modify the basic BSOFI algorithm by merging the first two phases – factorization of $H$ and inversion of factor $R$ – in a single factorization/inversion algorithm `BSOFTRI`. Hence, in the `BSOFTRI`, factorization is a part of computation process which utilizes both host and devices in parallel.

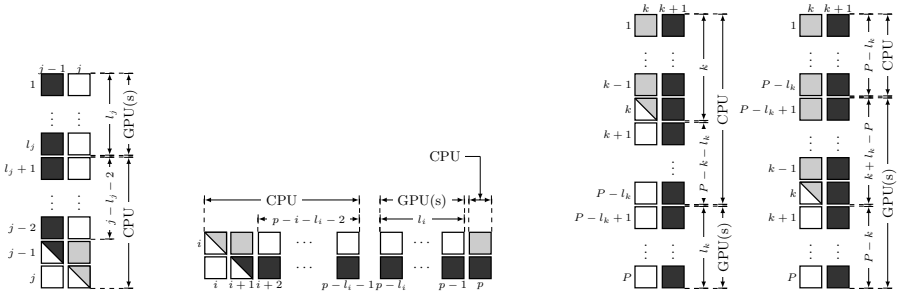
The merged factorization/inversion phase consists of three steps. At the first step, we perform partial factorization of input $p$-cyclic matrix $H$ on the host, where we run $l_F$ loop iterations of `BSOF` algorithm (see Fig. 1). This step is aimed at preparing columns of $R$ for further inversion and avoiding idles related to synchronizing concurrent threads in the next step. Since the optimal number of iterations $l_F$ is usually relatively small, this step does not influence the overall performance of the algorithm much. At the second step, we fork the computational process on two asynchronous threads. The first thread completes factorization on CPU, whereas the second one computes upper left corner of matrix $R^{-1}$ performing iterations of the column-wise inversion algorithm `BSTRI_CV` (see Fig. 2b). Once the SOF is completed, we join both threads and proceed to the third step, where we continue computing $R^{-1}$ via row version of the BBS algorithm `BSTRI_RV` (see Fig. 2a) omitting treatment of the $j_F$ already inverted blocks columns of $R^{-1}$. Switching from column-wise to row-wise inversion algorithm reduces computational complexity of `BSOFTRI` compared to algorithm which uses only column version of BBS.

Depending on the ratio between multicores and device performance and the value of $p$, we make a decision on the need for processing the last column panel in `BSOFTRI` inversion thread. If device performance is insufficient to invert more than first $p-2$ column panels of $R$ while $H$ is factorizing, we postpone processing the last column panel in order to avoid doubling of computational costs introduced by the original column version of BBS shown in Fig. 2b (see table 1).

In order to minimize data transfers from host to device in `BSOFTRI` algorithm, we employ devices in computing only those blocks of $R^{-1}$, which correspond to the zero blocks of $R$. The workload distribution between CPU and GPU(s) is controlled by parameters $l_j$ and $l_i$ respectively as illustrated in Fig. 5a and 5b.

In the phase of applying Householder reflectors to $R^{-1}$, we update block column pairs by means of explicit reconstruction of matrices $Q^{(k)}$ and the scheme similar to the algorithm shown in Fig. 3b. In this way, we replace `DORMQR` calls for applying reflectors to column panels in favor of more efficient `DGEMM` calls under the small computational overhead. The upper parts of block columns are updated on the host, whereas devices are used to update lower parts. Since the lower part of $R^{-1}$ has more zero blocks, this approach requires less data transfers from CPU to GPUs. In order to avoid physical data transfers inside devices, we store block columns of the input matrix $W$ in the reversed order with respect to the natural order of columns in matrix $X$. Namely, the $(k+1)$th block column of $X$ corresponds to $W_{:,1}$ and the $k$th block column of $X$ corresponds to $W_{:,2}$.

[5] White – zero blocks, light gray – input non-zero blocks, dark gray – blocks processed (partially or fully) in the preceding iteration. Irrelevant zero blocks are omitted.



(a) `BSTRI_CV` (b) `BSTRI_RV` (c) `BSOI_Qk`, if $l_k + k \leq p$ (d) `BSOI_Qk`, if $l_k + k > p$

Fig. 5: Workload distribution between host and device(s) while processing $j$th block column and $i$th block row of $R^{-1}$ (a,b) and applying $Q_k^T$ to $R^{-1}$ (c,d).[5]

Herewith we use the following equality to update column pairs on device

$$X_{p-l_k:p,k:k+1} Q^{(k)T} = \left[X_{p-l_k:p,k+1}\ X_{p-l_k:p,k}\right] \left[Q^{(k)}_{1:2,2}\ Q^{(k)}_{1:2,1}\right]^T \tag{4}$$

The workload distribution between host and device is controlled by $l_k$. If $l_k + k \leq p$ (Fig. 5c), then $X_{p-l_k:p,k} = 0$, and hence device does not require submatrix $Q^{(k)}_{1:2,1}$ to compute update according to (4). In contrast, if $l_k + k > p$ (Fig. 5c), whole matrix $Q^{(k)}$ and non-zero blocks of $k$th column panel of $X$ should be sent from host to device for further processing. For more details on the algorithm presented in this paragraph, see [4].

*Performance modelling and load balancing.* The parameters $l_i$, $l_j$, $l_k$ and $l_F$ introduced in the previous paragraph, control workload distribution between host and devices, and play a crucial role in performance tuning. The following text is an excerpt of results related to choosing above-mentioned parameters and performance modelling. These results are based on the following assumptions: (i) the elapsed time for multiplying $mn$-by-$n$ and $kn$-by-$n$ matrices is nearly $mk$ times more than the wall time for computing the product of two $n$-by-$n$ matrices; (ii) the performance of numerical kernel is roughly proportional to the number of CPU cores utilized in its computing. These assumptions are consistent with benchmarking results for moderate and large values of $n$ [4]. For more details and derivation of formulae, we refer to [4].

The formulae presented below use the following notation for time measures. $T^{\mathrm{cr}}_{\mathtt{R(P)}}$ denotes the wall time for BLAS and LAPACK routines `R` with a tuple of parameters `P` on the computational resource CPU or GPU, respectively. E.g., $T^{\mathrm{cpu}}_{\mathtt{DGEMM(m,n,k)}}$ is an elapsed time for computing a product of $m$-by-$k$ and $k$-by-$n$ matrices by means of the routine `DGEMM` on the CPU. We alias the routines for copying rectangular matrices to and from GPU with `SET` and `GET` respectively. We use braces if these data exchange operations can be executed algorithmically in parallel with some numerical kernel(s). I.e., $\{T^{\mathrm{gpu}}_{\mathtt{DGEMM(n,n,n)}}, T^{\mathrm{gpu}}_{\mathtt{GET(n,n)}}\}$ is

equal to $\max\{T^{\text{gpu}}_{\texttt{DGEMM(n,n,n)}}, T^{\text{gpu}}_{\texttt{GET(n,n)}}\}$ if copying is asynchronous and implemented via `cublasGetMatrixAsync`, and $T^{\text{gpu}}_{\texttt{DGEMM(n,n,n)}} + T^{\text{gpu}}_{\texttt{GET(n,n)}}$ if synchronous routine `cublasGetMatrix` is used.

The optimal values of the parameters $l_j$, $l_i$, and $l_k$ correspond to the situation if the elapsed time of processing assigned kernels on both host and devices are roughly the same in each iteration. These conditions result in the following approximations

$$l_j \approx \frac{1}{1+\kappa_C}\left(j + c_j\right), \quad l_i \approx \frac{1}{1+\kappa_R}\left(p - \min\{i, j_F\} + 1 + c_i\right), \quad (5)$$

$$l_k \approx \begin{cases} \dfrac{1}{1+\kappa_Q}\left(p + k + 2 + c'_k - c''_k\right), & \text{if } k \le \dfrac{\kappa_Q p - 2 - c'_k}{\kappa_Q + 2}, \\ \dfrac{1}{1+\kappa_Q}\left(p + \dfrac{\kappa_Q}{2}(p-k) + 1 + c'_k/2 - c''_k\right), & \text{if } k > \dfrac{\kappa_Q p - 2 - c'_k}{\kappa_Q + 2}, \end{cases} \quad (6)$$

where $\eta$ is a ratio between the number of cores involved in factorization of $H$ and the number of cores involved in inversion of $R$ in the second step of `BSOFTRI` (e.g., the typical values of $\eta$ for single hexa-core are $3:3$, $4:2$, or $5:1$),

$$\kappa_C = \frac{\{T^{\text{gpu}}_{\texttt{DGEMM(n,n,n)}}, T^{\text{gpu}}_{\texttt{GET(n,n)}}\}}{T^{\text{cpu}}_{\texttt{DGEMM(n,n,n)}}/(1+\eta)},\ c_j = 2\frac{T^{\text{cpu}}_{\texttt{DTRTRI(n)}} + 2T^{\text{cpu}}_{\texttt{DTRMM(n,n)}}}{2T^{\text{cpu}}_{\texttt{DGEMM(n,n,n)}}} - 1 > \frac{1}{6},$$

$$\kappa_R = \frac{\{T^{\text{gpu}}_{\texttt{DGEMM(n,n,n)}}, T^{\text{gpu}}_{\texttt{GET(n,n)}}\}}{T^{\text{cpu}}_{\texttt{DGEMM(n,n,n)}}},\ c_i = 2\frac{T^{\text{cpu}}_{\texttt{DTRTRI(n)}} + 3T^{\text{cpu}}_{\texttt{DTRMM(n,n)}}}{2T^{\text{cpu}}_{\texttt{DGEMM(n,n,n)}}} - 2 > -\frac{1}{3},$$

$$\kappa_Q = \frac{\{T^{\text{gpu}}_{\texttt{DGEMM(2*n,n,n)}}, T^{\text{gpu}}_{\texttt{GET(n,n)}}\}}{T^{\text{cpu}}_{\texttt{DGEMM(2*n,n,n)}}},\ c'_k = \frac{T^{\text{cpu}}_{\texttt{DORGQR(2*n,2*n,n)}}}{T^{\text{cpu}}_{\texttt{DGEMM(2*n,n,n)}}} - 2 > \frac{1}{3},\ c''_k = \frac{T^{\text{gpu}}_{\texttt{SET(n,n)}}}{T^{\text{cpu}}_{\texttt{DGEMM(2*n,n,n)}}}.$$

In order to reduce idle time related to the synchronization of parallel threads in the merged factorization/inversion phase, the total elapsed time for inversion of the first $j_F$ columns should be less than the elapsed time for performing $j_F - l_F$ factorization steps. This condition leads to the following lower bound for $l_F$

$$l_F(\delta) \ge \frac{1}{2} - c_j + \frac{\delta\eta}{10\beta_F}\left(c_j^2 + c_j - \frac{1}{4}\right) + \frac{5\beta_F}{2\delta\eta}, \quad (7)$$

where

$$\beta_F = \frac{T^{\text{cpu}}_{\texttt{DGEQRF(2*n,n)}} + T^{\text{cpu}}_{\texttt{DORMQR('R','N',2*n,n,n)}}}{5T^{\text{cpu}}_{\texttt{DGEMM(n,n,n)}}},$$

$\delta = 1$ if the last column panel inversion is postponed in the second step of `BSOFTRI`, and $\delta = 2$ otherwise. The latter inequality usually holds true for some $2 \le l_F \le 6$. If $\delta = 1$, $l_F$ and $j_F$ are linked by expression

$$\hat{l}_F(j_F) = p - \frac{1}{5}\left(\frac{\eta}{\beta_F}\left(\frac{1}{2}\frac{\kappa_C}{1+\kappa_C}\left(j_F^2 + (1+2c_j)j_F - 4c_j - 6\right) + 1\right) - 3p + 10\right).$$

Hence, postponed processing of the last column panel in the second step of `BSOFTRI` makes sense only if $l_F(1) > \hat{l}_F(p-2)$ for minimal $l_F$ which satisfies (7).

Results of the theoretical performance study are summarized in the table 2. For the sake of simplicity, the lower order terms are neglected. $\theta$ is a decreasing function of $l_F$. Its upper bound is $2\sqrt{2}$. The lower bound on $\theta$ is $\sqrt{3}$ if $\delta = 1$.

Table 2: Performance model of parallel "host-device" BSOFI, where Metric 1 is the number of flops on GPU, Metric 2 is the number of words CPU⇌GPU and Metric 3 is the number of messages CPU⇌GPU.

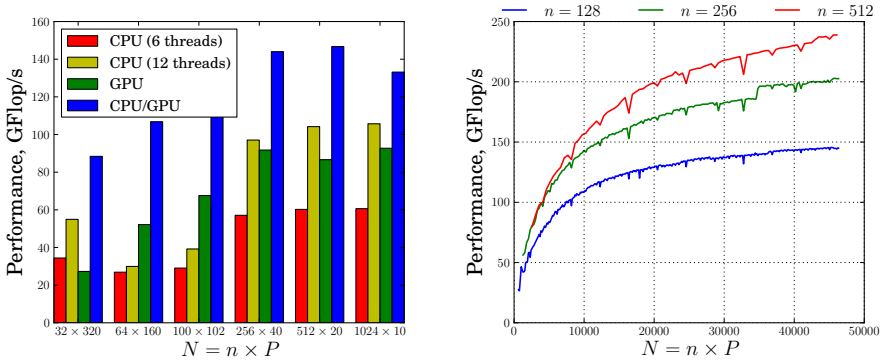
| Metric | `BSOFTRI` | `BSOI` |
|---|---|---|
| 1 | $\frac{n^3 p}{1+\kappa_R}\left(\theta^2\beta_F\left(1+\frac{\delta-1}{\delta\eta}\left(1+\frac{1}{\kappa_C}\right)\right)+p+1+2c_i\right)$ | $\frac{2n^3 p}{1+\kappa_Q}\left(3p+1+2c'_k-4c''_k\right)$ |
| 2 | $\frac{n^2 p}{1+\kappa_R}\frac{1}{2}\left(\frac{\theta^2\beta_F}{\delta}+p+5+2\kappa_R+2c_i\right)$ | $\frac{n^2 p^2}{1+\kappa_Q}\frac{1}{2}\frac{3\kappa_Q+4}{\kappa_Q+2}$ |
| 3 | $2p+2(2-\delta)\theta\sqrt{\frac{\beta_F}{\delta\eta}\left(1+\frac{1}{\kappa_C}\right)(p-2)}+2\delta-12$ | $2p-2$ |

## 5 Experimental results and analysis

*Experimental setup.* In order to examine our algorithmic solutions, we developed codes for stand-alone CPU and GPU processing, as well as hybrid CPU+GPU implementation. Our solvers receive $p$-cyclic matrix $H$ in an unpacked form as input, and replace it with its inverse $H^{-1}$ by performing in-place inversion. The POSIX threads are used for threading in the second step of `BSOFTRI`. For performance data presented below, the codes were compiled with ICC and linked with CuBLAS, Magma, and Intel's MKL library. Our codes are publicly available from `https://github.com/SGo-Go/BSOFI`. The performance data were collected on a 2-socket Intel Xeon X5670 coupled with NVIDIA GeForce GTX480 GPU. Intel Xeon X5670 is a 6-core processor with 2.9GHz clock rate. GTX480 is a CUDA-enabled, which implements Fermi architecture, and has 15 streaming multiprocessors with 32 CUDA cores in each. For multi-GPU studies, we used a multi-GPU Fermi node on the Dirac cluster, housed at NERSC of Lawrence Berkeley National Laboratory. This node contains 2 Intel 5530 2.4GHz Quad core Nehalem processors, and 4 C1060 NVIDIA Tesla GPUs.

*Performance tuning.* In order to make our hybrid CPU+GPU codes architecture-aware, we perform benchmarking of basic kernels used in the modified SOI algorithm, evaluate parameters for (5)–(7), and embed their approximate models into the code. Our experiments have shown that parameters $\kappa_R$, $\kappa_C$, and $\kappa_Q$ depend dramatically on the block size $n$ if $n$ is small, and this dependence can be sufficiently well approximated by the first order rational functions. We obtain parameters of these rational functions by Gauss-Markov estimator. The correction parameters $c_i$, $c_j$, $c'_k$, and $c''_k$ are approximated by descending step-wise functions of $n$. At first, we filter curves for these parameters received after benchmarking, and afterwards round the filtered curves to the closest integers. The same approach is used to build a step-wise approximation for parameter $l_F$.

*Benchmarking results.* To investigate the quality of exploiting structure by BSOFI algorithm, we compare performance of our CPU implementation with naïve BLAS3 LU inversion and inversion by multifrontal sparse LU solvers from UMFPack. Benchmarking shows significant speed-up of BSOFI with respect to




(a) Performance of the CPU, GPU and hybrid BSOFI codes

(b) Performance of the hybrid BSOFI codes

Fig. 6: Performance of BSOFI on a 2-socket Intel Xeon X5670 coupled with NVIDIA GeForce GTX480 GPU.

LU inversion. We observe up to 22× speed-up if $n \times p < 10^4$ and $n \geq 32$. The general tendency is an increase of speed-up with a decrease of $n$.

Fig. 6a shows the performance of BSOFI codes on different platforms for $N = 10120$. Hybrid implementation is up to 1.7× faster over the best of CPU and GPU codes. Its peak performance is higher than peak performance of `DGEMM` on CPU and is only 1.1× lower than the peak of `DGEMM` on GPU. Moreover, the difference in performance in the interval $32 \leq n \leq 1024$ does not exceed 1.5× for our hybrid CPU+GPU implementation. Fig. 6b compares the performance of CPU+GPU codes for different sizes of $p$-cyclic matrices if $n$ is fixed. If $n$ is large, performance of subroutine `BSOI` on the single GPU node is two times more in the case of CPU+GPU implementation than in the case of pure CPU implementation. At the same time, performance improvements for subroutine `BSOFTRI` are less significant than for `BSOI`. If $n \gtrapprox 512$, performance curves are close to each other. It is a consequence of reaching maximum performance for `DGEMM` on both CPU and GPU.

More benchmarking results on both single and multi-GPU platforms can be found in [4].

## 6 Conclusions and further directions

We presented serial and parallel algorithms for structured orthogonal inversion of block $p$-cyclic matrices. We provided a performance model and discussed host-device load balance. We developed and explored CPU, GPU and hybrid CPU+GPU codes Benchmarking has shown that our codes for multicores with GPU accelerators maintain sustainable performance for different values of problem size $n$, and attain up to 90% of realistic peak performance in terms of the operation of the matrix-matrix multiplication.

There are numerous ways to extend results presented in this paper. Since GPUs have a lot in common with Intel MIC architecture, it seems natural to verify the approaches on multicores with MIC accelerators. Another promising direction is in coupling of SOI with inversion based on explicit formulae. We conclude by mentioning that the solutions proposed here can be extended to the problems with other block structured matrices such as block upper Hessenberg matrices. This leads us to believe that the BSOFI could be a vital substitute to conventional Gaussian elimination based inversion for broader classes of block structured matrices.

## References


1. Bai, Z., Chen, W., Scalettar, R., Yamazaki, I.: Numerical methods for Quantum Monte Carlo simulations of the Hubbard model. In: Hou, T.Y., Liu, C., Liu, J.G. (eds.) Multi-Scale Phenomena in Complex Fluids, Series in Contemporary Applied Mathematics, vol. 12, chap. 1, pp. 1–100. World Scientific (2009)
2. Ernst, O.G.: Equivalent iterative methods for $p$-cyclic matrices. Numerical Algorithms 25(1-4), 161–180 (2000)
3. Fairweather, G., Gladwell, I.: Algorithms for almost block diagonal linear systems. SIAM Review 46(1), 49–58 (2004)
4. Gogolenko, S., Bai, Z.: A structured orthogonal inversion of block $p$-cyclic matrices on multicores with GPU accelerators. Tech. Rep. CSE-2013-78, CS Dept., UC Davis (2013), `www.cs.ucdavis.edu/research/tech-reports/2012/CSE-2013-78.pdf`
5. Khabou, A., Demmel, J., Grigori, L., Gu, M.: LU factorization with panel rank revealing pivoting and its communication avoiding version. SIAM J. Matrix Analysis Applications 34(3), 1401–1429 (2013)
6. Tomas, A., Chang, C.C., Scalettar, R., Bai, Z.: Advancing large scale many-body QMC simulations on GPU accelerated multicore systems. In: Proceedings of IPDPSW'12. pp. 308–319. IEEE, Washington, DC, USA (2012)
7. Tomov, S., Nath, R., Ltaief, H., Dongarra, J.: Dense linear algebra solvers for multicore with GPU accelerators. In: Proceedings of IPDPSW'10. pp. 1–8. IEEE, Atlanta, GA, USA (2010)
8. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. Rep. UCB/EECS-2008-49, EECS Dept., UC Berkeley (2008), `www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html`
9. Wright, S.J.: Stable parallel algorithms for two-point boundary value problems. SIAM J. Sci. Stat. Comput. 13(3), 742–764 (May 1992)
10. Wright, S.J.: A collection of problems for which Gaussian elimination with partial pivoting is unstable. SIAM J. Sci. Comput. 14(1), 231–238 (Jan 1993)