BIT Numer Math (2011) 51:1009–1030
DOI 10.1007/s10543-011-0333-5

BIT

A GPU-based hyperbolic SVD algorithm

Vedran Novaković · Sanja Singer

Received: 6 August 2010 / Accepted: 12 April 2011 / Published online: 7 June 2011

Abstract A one-sided Jacobi hyperbolic singular value decomposition (HSVD) algorithm, using a massively parallel graphics processing unit (GPU), is developed. The algorithm also serves as the final stage of solving a symmetric indefinite eigenvalue problem. Numerical testing demonstrates the gains in speed and accuracy over sequential and MPI-parallelized variants of similar Jacobi-type HSVD algorithms. Finally, possibilities of hybrid CPU–GPU parallelism are discussed.

Keywords One-sided Jacobi algorithm · Hyperbolic singular value decomposition · Symmetric indefinite eigenvalue problem · GPU parallel programming

Mathematics Subject Classification (2000) 65F15 · 65Y05 · 65Y10

1 Introduction

In this paper we develop a hyperbolic SVD algorithm (HSVD) for graphics processors (GPUs). To the best of our knowledge, this is the first one-sided Jacobi-type HSVD algorithm for full column rank matrices of arbitrary dimensions, that uses GPUs.

Presented at the BIT50 conference in Lund, Sweden 17–20 June 2010.

Communicated by Axel Ruhe.

This work was supported by grants 037–1193086–2771 and 120–1201703–1672 by the Ministry of Science, Education and Sports, Republic of Croatia.

V. Novaković (✉) · S. Singer
Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, Ivana Lučića 5, 10000 Zagreb, Croatia
e-mail: venovako@fsb.hr

S. Singer
e-mail: ssinger@fsb.hr

The first paper that we know of, which describes a Jacobi-type computation of the SVD on GPUs, was published by Zhang and Dou in [27]. According to English summary of the paper, they compute the SVD by using the one-sided Jacobi algorithm on a matrix of order 512. The order of orthogonalization of pairs of columns is chosen by the parallel pivot strategy described in [1]. Later, Lahabar and Narayanan [9] have computed SVD on GPUs by bidiagonalization followed by a bidiagonal QR algorithm (in single precision arithmetic). Finally, Sachdev, Vanjani and Hall in [15] presented an algorithm for the Takagi SVD (for complex symmetric matrices) by a symmetrized version of the two-sided Kogbetliantz algorithm (in single precision arithmetic).

Given a Hermitian indefinite matrix M, factorized as $M = GJG^*$, with G of full column rank and J a diagonal matrix holding the inertia of M, we aim at computing the HSVD of the factor G,

$$G = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^*, \tag{1.1}$$

where U is unitary, V is J-unitary (i.e., $V^*JV = J$), and Σ is a diagonal matrix with positive diagonal entries.

If G is not of full column rank, Σ in (1.1) is not diagonal [26, Remark 5], and the non-diagonal part reflects the loss of rank of GJG^* compared with the rank of G. In the very important special case, when $J = I$, the HSVD is the ordinary SVD.

If the HSVD of a factor G is known, then the eigendecomposition of M is readily available as

$$M = GJG^* = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^*JV \begin{bmatrix} \Sigma & 0 \end{bmatrix} U^* = U(\Sigma^2 J)U^*,$$

i.e., the matrix U of left singular vectors of G is also the eigenvector matrix of M, while the eigenvalues of M are diagonal elements of $\Sigma^2 J$.

In many applications, M is already given implicitly, by its rectangular factor G, and the signature matrix J. In this case, G should be shortened either by the QR factorization (if G is tall) or by the hyperbolic QR factorization (see [18]) of G^* (if G^* is tall), for the efficiency of the HSVD algorithm. In both cases, the HSVD of a square matrix will be computed, either of R (in case of the QR factorization), or of R^* (in case of the hyperbolic QR factorization). To obtain matrices of singular vectors, after the completion of the HSVD, either U (in case of the ordinary QR factorization), or V (in case of the hyperbolic QR factorization) should be premultiplied by Q.

On the other hand, if M is given explicitly, G and J are usually computed by the Hermitian indefinite factorization with postprocessing (see [22]), which always produces G of the full column rank. Note that G obtained by the Hermitian indefinite factorization can be rectangular with more rows than columns, and should be shortened by the QR factorization. Therefore, it is sufficient to efficiently compute HVSD of a square matrix.

The HSVD also serves as the second stage in solving a Hermitian indefinite eigenproblem. Our method of choice for such a computation is the one-sided hyperbolic

Jacobi algorithm [25], which was much studied in recent years, due to its high relative accuracy [23] and various possibilities of efficient blocking and parallelization in the scope of the conventional CPU computing [20, 21].

The efficient GPU solution (especially, one with minimal CPU intervention) for the first stage (i.e., Hermitian indefinite factorization) remains an open problem.

We will show that the Jacobi algorithm is elegantly parallelizable on modern GPUs, utilizing them as a primary target for the algorithm execution. The CPU acts only as the ancillary unit: for driving and synchronizing the GPU computation, and for data initialization on the GPU.

Numerical experiments demonstrate the benefits of such an approach, compared to the fastest existing sequential and multi-process parallel Jacobi implementations. Significant speedup vs. sequential, and moderate speedup vs. the CPU-parallel algorithm with 4 processes are obtained.

Finally, we discuss the true strength of our approach, through the possibilities of combining the GPU parallelism with the traditional one, for very large problems.

The rest of the paper is organized as follows. In Sect. 2 the essentials of the Jacobi HSVD and basic tools for its parallelization are presented. Section 3 deals with the properties and constraints of a class of GPU computing platforms our algorithm, named GPUJACH1, is designed for. The algorithm itself is detailed in Sect. 4, and it is compared, by numerical testing, with a sequential and a CPU-parallel implementation of the Jacobi HSVD in Sect. 5. We conclude with a discussion of possible applications of GPUJACH1 in the context of hybrid CPU–GPU parallelism, and with notes on future work, in Sect. 6. The Appendix contains a proof of convergence for the modulus pivot strategy.

2 The essentials of the Jacobi HSVD

The hyperbolic one-sided Jacobi algorithm implicitly diagonalizes a definite pair (A, J), where $A := G^*G$, by J-unitary congruences [23]. "Implicitly" in this context means that the columns of G are orthogonalized.

The idea of the one-sided algorithm is to multiply the columns of $G^{(0)} := G$ from the right-hand side by J-unitary matrices V_k^{-*}, $k = 1, 2, \ldots, s$, to obtain the matrix $G^{(s)}$, with numerically orthogonal (not orthonormal) columns. If $[V^{-*}]^{(0)}$ is defined as $[V^{-*}]^{(0)} = I$, then

$$G^{(k)} = G^{(k-1)} V_k^{-*}, \qquad [V^{-*}]^{(k)} = [V^{-*}]^{(k-1)} V_k^{-*}, \quad k = 1, 2, \ldots, s. \tag{2.1}$$

If $G^{(s)}$ has sufficiently orthogonal columns, $[V^{-*}]^{(s)}$ serves as a quite good approximation of V^{-*}. Now, the matrix V of right (hyperbolic) singular vectors is easily obtainable from V^{-*}, since V is a J-unitary matrix, i.e., $V = JV^{-*}J$.

The hyperbolic singular values σ_i are norms of the final $G^{(s)}$, while the approximate matrix U of left singular vectors is computed by column scaling of $G^{(s)}$ by diagonal matrix $\Sigma^{-1} = \mathrm{diag}(\sigma_1^{-1}, \ldots, \sigma_n^{-1})$.

If the algorithm is used only for the eigenvalue computation, V^{-*} need not be accumulated, since U keeps the eigenvectors of $M = GJG^*$, while the eigenvalues are squares of singular values multiplied by a correct sign from J, i.e., $\lambda_i = \sigma_i^2 j_{ii}$.

Just for simplicity, from now on, we restrict ourselves to the real case, i.e., to symmetric matrices, instead of Hermitian. To emphasize this, we will use symbol ^{T} (instead of *) for transposition.

The J-unitary matrices V_k^{-T} from (2.1) are usually very simple—they are plane rotations (trigonometric and hyperbolic). Such a rotation orthogonalizes a pair of columns of G. Their non-identity parts can be represented as

$$\begin{aligned} V_T^{-T} &:= \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} = \begin{bmatrix} 1 & \tan\varphi \\ -\tan\varphi & 1 \end{bmatrix} \begin{bmatrix} \cos\varphi & 0 \\ 0 & \cos\varphi \end{bmatrix}, \\ V_H^{-T} &:= \begin{bmatrix} \cosh\varphi & \sinh\varphi \\ \sinh\varphi & \cosh\varphi \end{bmatrix} = \begin{bmatrix} 1 & \tanh\varphi \\ \tanh\varphi & 1 \end{bmatrix} \begin{bmatrix} \cosh\varphi & 0 \\ 0 & \cosh\varphi \end{bmatrix}. \end{aligned} \tag{2.2}$$

Note that φ is not needed explicitly, the tangent would suffice to construct a rotation. In the Hermitian case, only one additional angle α is needed, for example see [17].

The column orthogonalization starts by forming a 2×2 pivot block A_p,

$$A_P = \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ij} & a_{jj} \end{bmatrix} := \begin{bmatrix} g_i & g_j \end{bmatrix}^T \begin{bmatrix} g_i & g_j \end{bmatrix},$$

where g_i and g_j denote the ith and jth column of G (we may assume $i < j$).

Optionally, further processing of the column pair may be avoided if the columns are relatively orthogonal (up to the machine precision), i.e.,

$$a_{ij} < \varepsilon \sqrt{a_{ii} a_{jj}}, \tag{2.3}$$

with ε is the smallest floating-point number such that $\mathrm{fl}(1+\varepsilon) = 1$.

First, the adequate rotation (trigonometric V_T^{-T} if ith and jth diagonal element of J agree, or hyperbolic V_H^{-T} otherwise) to annihilate the element a_{ij} is computed. Then, this transformation (2.2) is applied from the right to the columns $[g_i \; g_j]$, giving

$$g_i' := (g_i - \tan\varphi g_j)\cos\varphi, \qquad g_j' := (g_j + \tan\varphi g_i)\cos\varphi \tag{2.4}$$

in the trigonometric case, and

$$g_i' := (g_i + \tanh\varphi g_j)\cosh\varphi, \qquad g_j' := (g_j + \tanh\varphi g_i)\cosh\varphi \tag{2.5}$$

in the hyperbolic case. The formulas (2.4)–(2.5), written pointwise, represent a basic tool for updating the columns of G. The columns of V^{-T} are updated in the same fashion, with appropriate change of notation, see (2.1).

The parts of the formulas (2.4)–(2.5) written in parentheses can be performed by a single fused multiply-add (FMA) operation, where available, with only one rounding of the result, thus improving the speed, while exhibiting smaller roundoff errors.

All Jacobi-type algorithms iterate until convergence criteria are met. Usually, these iterations are organized in sweeps (sometimes called cycles). For a symmetric matrix A of order n, a sweep is a collection of $n(n-1)/2$ annihilations of different elements in the strict upper triangle of A. In other words, in a sweep, pairs of columns of G (pivot pairs) are orthogonalized as described previously. The order of pivot pairs is

chosen according to a pivot strategy. In sequential use, widespread pivot strategies are row and column cyclic, since they use cache memory well (depending on data layout of arrays in memory). For both strategies, convergence [25] and asymptotic quadratic convergence [3] have been proved. If the pivot pairs are orthogonalized in a cyclic manner more than once (but a prescribed number of times) in a 'sweep', this 'sweep' is usually called a quasi-sweep and such a pivot strategy is called quasi-cyclic.

A simple convergence criterion is that no rotations occurred in a (quasi-)sweep, due to condition (2.3), which guarantees relative accuracy [23]. The process could also be stopped if the computed tangents are all below a predefined threshold $T_\varepsilon := \sqrt{\varepsilon}/2$, i.e., when the quadratic convergence is detected. The second threshold test is a replacement for relative orthogonality criterion (2.3) from [23] since we are trying to avoid an extra (quasi-)sweep. If we define

$$hyp = \begin{cases} -1, & \text{in the trigonometric case,} \\ 1, & \text{in the hyperbolic case,} \end{cases}$$

then

$$\theta = \begin{cases} \tan 2\varphi & \text{if } hyp = -1, \\ \tanh 2\varphi & \text{if } hyp = 1, \end{cases} \qquad t = \begin{cases} \tan \varphi & \text{if } hyp = -1, \\ \tanh \varphi & \text{if } hyp = 1. \end{cases}$$

If the tangent is very small, i.e., $|t| \ll 1$, then $\theta \approx 2t$. The angle θ is computed from the requirement that $a'_{ij} = 0$,

$$\theta = \frac{2a_{ij}}{-a_{ii} + hyp \cdot a_{jj}}.$$

Therefore, for $|t| \leq T_\varepsilon$, we have $a_{ij} \approx t(-a_{ii} + hyp \cdot a_{jj})$, and since

$$a'_{ii} = a_{ii} + hyp \cdot t a_{ij}, \qquad a'_{jj} = a_{jj} + t a_{ij}, \tag{2.6}$$

when $|t| \leq T_\varepsilon$, the second term from the right-hand side in (2.6) satisfies

$$|hyp \cdot t a_{ij}| = |t a_{ij}| \lesssim \frac{\varepsilon}{4} \cdot (|-a_{ii} + hyp \cdot a_{jj}|) \leq \frac{\varepsilon}{2} \cdot \max\{a_{ii}, a_{jj}\}.$$

Therefore $a'_{ii} \approx a_{ii}$ and $a'_{jj} \approx a_{jj}$ in (2.6). When $\mathrm{diag}(JA)$ is sorted and the process is near the end, the off-diagonal norm of A, i.e., $\|A - \mathrm{diag}(A)\|_F$, quadratically diminishes [3], and so do the off-diagonal elements. The tangents in the following (quasi-)sweep will also be quadratically smaller, and the diagonal updates in (2.6) will be negligible.

To summarize, the orthogonality convergence check is a safe fallback, with guaranteed numerical orthogonality of the eigenvectors, but in our testing experience, the threshold criterion has always happened before and terminated the process.

The Jacobi HSVD algorithm, when computing the eigendecomposition, can be implemented by using just one 2-dimensional array G, initialized to the factor G. At the end of the process, G holds the computed $U\Sigma$. The column norms of $U\Sigma$ are the hyperbolic singular values, and normalized columns are the left singular vectors of

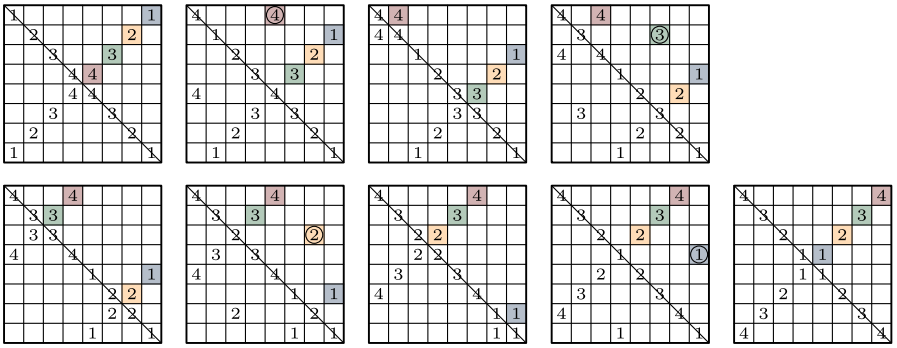

Fig. 1 Modified modulus strategy for $n = 8$ and $q = 4$. *Background colors* denote elements annihilated in one step, and *circled elements* are annihilated twice in a quasi-sweep

G, i.e., the eigenvectors of M. If the full HSVD is desired, the sequence of applied V_T^{-T} and V_H^{-T} transformations, i.e., the matrix V^{-T}, needs to be accumulated in another 2-dimensional array V, starting with the identity matrix I.

On a theoretical level, the main difference between a sequential and a parallel one-sided Jacobi algorithm is the choice of a pivot strategy. In any parallel algorithm, CPU or GPU based, a pivot strategy is chosen to simultaneously orthogonalize as many independent (non-overlapping) pivot pairs as possible. These pairs can either be two single columns, or two block-columns. Either way, this collection of independent pairs will be called a step.

Our choice of parallel pivot strategy, for CPU and GPU algorithms alike, is a slightly modified modulus strategy [11]. If the order n of a matrix is even, $n = 2q$, then a quasi-sweep has exactly n steps, and all the elements in the upper triangle of A are annihilated (some of them twice). For q independent tasks (e.g., CPU or GPU threads), exactly q pivot pairs are simultaneously orthogonalized in each step.

In Fig. 1 the modified modulus pivot strategy, for $n = 8$ ($q = 4$), is illustrated from a point of view of the (implicit) matrix A. Each subfigure shows a single step, with 4 pivot submatrices that can be independently processed by 4 tasks, either sequentially or simultaneously. After 8 steps a quasi-sweep is completed, and a new one will begin with reversed initial assignment of pivot columns to tasks, as shown in the ninth subfigure.

Our modified modulus pivot strategy differs from [11] in the choice of a start step (ours is on the antidiagonal), and in annihilating exactly twice the elements of the diagonal $(i, q + i)$, $i = 1, \ldots, q$, for even n. This is a quasi-cyclic strategy, designed to speedup the convergence. The proof of convergence for this strategy is a work in progress. If we avoid double annihilation, our strategy is shift equivalent to the modulus strategy, for which the proof of convergence is detailed in Appendix.

For more details of the modified modulus pivot strategy, including the efficient algorithm for step transitions, see Sect. 4 and [20].

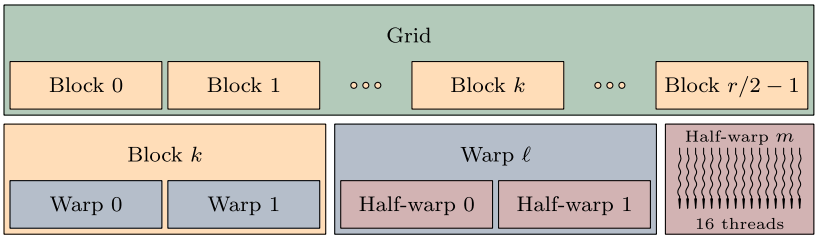

Fig. 2 The CUDA grid for GPUJACH1

3 The GPU computing platform

GPU computing has already evolved to a mainstream technology. Although vendor-neutral standards (like OpenCL [8]) have emerged, we implemented GPUJACH1 for the NVIDIA CUDA architecture [13], due to its maturity and close ties to the underlying hardware. The algorithm is portable, more or less efficiently, to any similar single-instruction multiple-threads platform (SIMT), provided IEEE 754-2008 [7] double-precision floating-point arithmetic is available and the basic computational concepts are found in (or could be mapped to) other architecture.

In short, the CUDA architecture provides a programmer with a thin layer of abstractions, programming tools and interfaces on top of the recent NVIDIA graphics processing hardware. The hardware itself is seen as a massively parallel device whose threads execute the same instruction sequence over (possibly) different data, stored on the device. The execution is initiated by the CPU (host), and a fast host-to-device and device-to-host data transfer is available (but not as fast as device memory access).

This SIMT paradigm entails a careful rethinking of even basic algorithms. Any branching causes a significant slowdown, since threads on divergent branches proceed sequentially. The Jacobi algorithm, however, fits this paradigm perfectly, because the same orthogonalization primitives are applied to different pairs of matrix columns concurrently.

A subroutine the device threads are executing is called a kernel. A kernel can be written in a high-level programming language (e.g., C), or by using the assembly-like, low-level, but general-purpose instruction set of the PTX [14] virtual machine.

The device threads are grouped into so-called warps, for memory access optimization, execution scheduling and finest-grain synchronization. A warp consists of 32 threads in the current CUDA hardware, with two half-warps of 16 threads each.

From a high-level perspective, blocks of threads are established. All blocks are of the same size, and may be seen as one, two or three dimensional arrays of threads. Threads are indexed and may be synchronized only within their blocks, i.e., different blocks are concurrent and mutually independent. For each launch of a kernel, the size of a block and the number of blocks are set, tailored to the nature of the algorithm and the data. Such a one or two dimensional array of blocks is called a grid.

The grid for GPUJACH1 kernels, working on a factor G with even number of columns r, is shown in Fig. 2.

Memory management is the major obstacle in GPU programming, as many distinct memory spaces of varying speed, size and accessibility are involved.

Input and output data are stored in the global memory, a large but slow portion of the GPU memory. It is accessible to the CPU and all GPU threads. Improper (uncoalesced) access tremendously degrades the performance [13], thus the carefully aligned addressing of successive locations by threads with successive indices is required.

The shared memory, on the other hand, is a small but fast device memory, allocated per block, and ideal for data exchange between threads of the same block.

The constant memory, small and cached, holds kernel parameters that are not changing between launches (e.g., device pointers to global memory arrays in GPUJACH1).

Arithmetic is done in per-thread registers. GPUJACH1 uses 64-bit floating-point and 32-bit integer arithmetic. FMA instruction (e.g., dot-products, column updates) and 24-bit integer multiplication (e.g., address calculations) are chosen, if possible.

GPUJACH1 targets NVIDIA GT200 graphics processor series, common at the time of writing this paper. The threads are executed by 30 8-core multiprocessors (SMs). Each SM has a 64 kB register file, one double-precision arithmetic unit and 16 kB of shared memory. More than one block can reside on an SM, as the resources (register usage per thread and shared memory allocation per block) and other constraints [13] allow. GPUJACH1 needs only 512 B of shared memory per block, but 48 32-bit registers per thread, thus at most 5 blocks can reside on an SM at a given time.

Two distinctive features of the GT200 series are that no cache is available for the global memory, and that single-precision arithmetic deviates in the guaranteed accuracy too much from the standard [7] to be useful for GPUJACH1. Both issues are addressed by the more advanced architectures (e.g., NVIDIA Fermi [13]).

4 The GPUJACH1 algorithm

The GPUJACH1 algorithm is an efficient parallel realization of the one-sided pointwise hyperbolic Jacobi SVD. GPUJACH1 provides all parts of the Jacobi process executed on the GPU as much as possible.

GPUJACH1 consists of, essentially, two parts:

– *host* routines (auxiliary), that are executed on the CPU, and
– *device* routines (computational), that are executed on the GPU.

Host routines serve mainly to call device routines in a synchronized manner and collect the resulting information, where needed.

A sketch of the main driver routine is given in Algorithm 4.1, and then its essential parts are described in details. In the following, by the subscript H host variables, and by D device variables are denoted. Arrays are assumed to be in Fortran (column-major) order, but indices are zero-based (as in C). Array slices are written in Matlab fashion.

GPUJACH1 holds the $n \times r$ factor G in arrays $G_{H,D}$ (we write $G_{H,D}$ as a shorthand for G_H and G_D, similarly for V). We may assume the factor to be square ($r \times r$), as left by preprocessing (see Sect. 1). The factor could also be preloaded on the GPU by a previous processing, and not needed on the CPU afterward, so G_H is optional. The factor must be of the full column rank, r needs to be even and not greater than n,

Algorithm 4.1: The host driver routine

Description: This is the main program, executed on the CPU.
Input: $n \times r$ factor G, and the number p of positive signs in J.
Output: the hyperbolic singular values Σ and the matrices of left (U) and hyperbolic right (V^T) singular vectors.

```
Host_Driver_Routine ;
begin
    G_D ← G_H; V_D ← I_r;    // optional
    Precompute_Data ;
    Sort_Diagonal ;
    for sweep = 0 to MaxSweep − 1 do
        Reset_Convergence ;
        for step = 0 to r − 1 do
            Jacobi_Step ;
        end for
        Check_Convergence ;
        Sort_Diagonal ;
    end for
    G_H ← G_D; V_H ← V_D; D_H ← D_D;    // optional
end
```

and n, for performance reasons, should be a multiple of the warp size. This is not a loss of generality, since the initial G could easily be bordered, as in (4.1), to satisfy these constraints

$$G'_{H,D} = \begin{matrix} n \\ \\ \\ \end{matrix} \left\{ \begin{bmatrix} G & \vdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \vdots & 1 \\ \cdots & \cdots & \cdots \\ 0 & \vdots & 0 \end{bmatrix} \right\} \ell \cdot WarpSize. \tag{4.1}$$

If the matrix V^{-T} is needed, it is accumulated in the array V_D, starting from the identity I_r, and is optionally transferred to V_H at the end. Arrays $V_{H,D}$, if used, must have the same number of columns as $G_{H,D}$, and the same restrictions (and appropriate bordering) apply.

In principle, the execution begins with all the data residing in the main (CPU) memory, and should be copied to the global memory of the GPU.

For efficiency, to access global memory data as few times as possible, and to reuse results of the computation while still in registers, the diagonal of the implicit pivot matrix A is kept and updated in each Jacobi step. While performance gains are here obvious, yet additional speedup can be obtained by a special diagonal sorting, described in [21] for block versions of the Jacobi algorithm. Keeping the diagonal, i.e., the eigenvalues, sorted ensures the quadratic convergence of the Jacobi algorithm [3, 19]. To facilitate the sorting, the diagonal entries d, the composition of the sort-

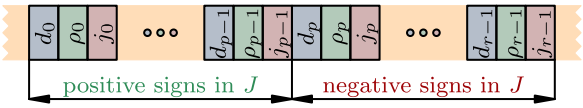

Fig. 3 $D_{H,D}$—packing of diagonal, permutation, and sign r-vectors in one

ing permutations ρ, and the signature J are packed into vectors $D_{H,D}$, as shown in Fig. 3.

The separation of $D_{H,D}$ to p entries with positive and $r-p$ entries with negative sign elements of J is always maintained. D_D is precomputed in Algorithm 4.2, and sorted in `Sort_Diagonal` routine. The sorting key of a diagonal package is d. The first part of the diagonal (the one with positive signs in J) is sorted decreasingly, and the last part (with negative signs in J) is sorted increasingly, with respect to that key. We chose a GPU implementation of the merge-sort, `thrust::sort`, from the Thrust library [6], to serve as `Sort_Diagonal`. Note that, at the end of the process, D_D holds the sorted and squared hyperbolic singular values d and the final permutation ρ.

Algorithm 4.2: Precompute_Data routine

Description: Routine computes
$D_D = (d = \mathrm{diag}(G_D^T G_D),\ \rho = \mathrm{id}_r,\ j = \mathrm{diag}(I_p, -I_{r-p}))$, and initializes the stepper vector S_D.

`Precompute_Data` (`Device_Code` (block k));
begin
  $(i, j) = (2k, 2k+1)$;
  $d_i = g_i^T g_i$; $d_j = g_j^T g_j$;
  $\rho_i = i$; $\rho_j = j$;
  if $i < p$ **then** $j_i = 1$ **else** $j_i = -1$; **if** $j < p$ **then** $j_j = 1$ **else** $j_j = -1$;
  `Initialize_Stepper` ;
end

Diagonal packing from Fig. 3 might seem unnecessary, when only diagonal entries and the permutation are strictly needed, and could be arranged into two simple, separate vectors of doubles and integers, respectively. Diagonal entries could also carry the respective signs from J, to simplify sorting. This approach is certainly possible, but we show in the following section it is slower, due to slightly more complex `Jacobi_Step` routine, than the presented solution. Moreover, packed j incurs no extra overhead over packing solely d and ρ together—the place j occupies in the package must exist because of data alignment constraints of the GPU [13].

The CUDA grid for GPUJACH1 kernels (Fig. 2) has exactly $b = r/2$ blocks. Each block is assigned its own pair of columns of G_D. In each Jacobi step these pivot pairs are mapped to blocks according to the modified modulus pivot strategy, implemented as an integer vector

$$S_D = (ip, jp, i\_blk, j\_blk)$$

of length $4 \times b$, and a device routine, called `Update_Stepper`, which updates S_D at the end of each step, independently by each block (see Algorithms 4.3 and 4.4).

Algorithm 4.3: Initialize_Stepper routine

`Initialize_Stepper` (`Device_Code` (block k))(r, $S_D = (ip, jp, i\_blk, j\_blk)$);

Description: Initialization of the modified modulus strategy stepper vector S_D to antidiagonal—$k \leftrightarrow (k, r-k-1)$. Variables ip and jp are auxiliary, while i_blk_k and j_blk_k contain the first and the second column index (non-permuted) of a pivot pair for block k.

begin
  $ip_k = k$; $i\_blk_k = ip_k$; $jp_k = r-k-1$; $j\_blk_k = jp_k$;
end

Algorithm 4.4: Update_Stepper routine

`Update_Stepper` (`Device_Code` (block k))(r, $S_D = (ip, jp, i\_blk, j\_blk)$);

Description: Block k determines the indices of the next pivot pair (i_blk_k, j_blk_k).

begin
  if $(ip_k + jp_k) > r-1$ **then**
    $ip_k = ip_k + 1$;
    if $ip_k = jp_k$ **then**
      $ip_k = ip_k - r/2$; $jp_k = ip_k$;
    end if
    $i\_blk_k = ip_k$;
  else
    $jp_k = jp_k + 1$; $j\_blk_k = jp_k$;
  end if
end

In other words, each block orthogonalizes its pivot pair $(\rho(i\_blk_k), \rho(j\_blk_k))$. In a block, each warp is dedicated to access a single column only. This design makes the device code almost completely uniform for all threads, thus avoiding branching as much as possible.

In the `Jacobi_Step` (Algorithm 4.5) columns of G_D and V_D are indexed by the current permutation ρ, and no physical swapping of columns ever takes place. Therefore, at the end of the process, G_D holds $U\Sigma$, and V_D holds V^{-T}, in the original column order. To match the computed (and permuted) singular values to the singular vectors, the elements of $D_{H,D}$ need to be permuted by the inverse of the final permutation ρ.

The rest of `Jacobi_Step` (Algorithm 4.5) is more or less standard. The final question and the crucial efficiency issue is how dot products and column updates (`daxpy`-like operations) are performed. We shall describe only the dot product computation, since the updating of columns is done in a similar fashion.

Algorithm 4.5: Jacobi_Step – the main computational routine

Description: This is the main GPU computational routine.

`Jacobi_Step` (`Device_Code` (block k));
begin
  $(i, j) = (S_D(k).i\_blk, S_D(k).j\_blk)$; // $i < j$
  $a_{ii} = d_i$; $a_{jj} = d_j$; // `already computed`
  $a_{ij} = g^T_{\rho(i)} g_{\rho(j)}$; // `dot-product`
    // `Indexing by permutation;`
    // `no physical swapping of` G_D `columns;`
  if $g_{\rho(i)}$ *and* $g_{\rho(j)}$ *relatively orthogonal (up to given* ε*)* **then goto** `End` ;
  if $j_i = j_j$ **then** set $hyp = -1$ **else if** $j_i = -j_j$ **then** set $hyp = 1$;
  if $hyp = -1$ **then** compute t from a_{ii}, a_{jj}, a_{ij} as $\tan\varphi$
  else if $hyp = 1$ **then** as $\tanh\varphi$;
  if $hyp = -1$ **then** compute c from t as $\cos\varphi$ **else if** $hyp = 1$ **then** as $\cosh\varphi$;
  $g'_{\rho(i)} = (g_{\rho(i)} + hyp \cdot t g_{\rho(j)})c$; $g'_{\rho(j)} = (t g_{\rho(i)} + g_{\rho(j)})c$; // `FMA, scal`
    // `While updating` G_D `columns, compute new` d'_i, d'_j;
  $d'_i = (g'_{\rho(i)})^T g'_{\rho(i)}$; $d'_j = (g'_{\rho(j)})^T g'_{\rho(j)}$;
  $v'_{\rho(i)} = (v_{\rho(i)} + hyp \cdot t v_{\rho(j)})c$; $v'_{\rho(j)} = (t v_{\rho(i)} + v_{\rho(j)})c$; // `FMA, scal`
  `Update_Convergence` ; // `as in (4.2)`
  `End: Update_Stepper` ;
end

For a dot product, each warp "grabs" $WarpSize = 32$ successive elements of its column vector, one element per each thread. The threads then exchange these values via shared memory and update (FMA) their local partial sums. Finally, the partial sums are reduced in the shared memory, as shown in Fig. 4. This reduction at warp level is free of synchronization [13], and needs to keep all the threads in a warp alive for further processing.

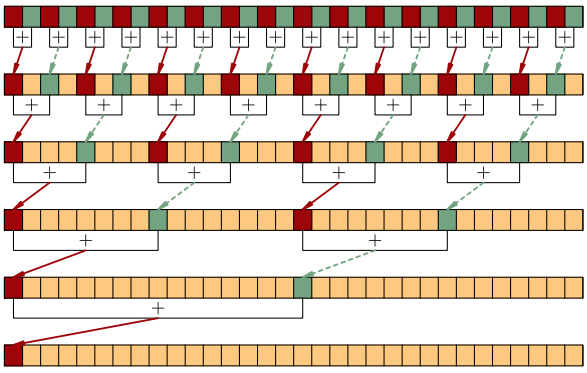

Fig. 4 Per-warp reduction (via shared memory) of the partial sums to the final dot-product. After each reduction stage "dashed" threads wait for the whole reduction to complete (but do not terminate)

The convergence statistics is held in an integer vector C_D of length b, which is updated independently for each block in a step. If no rotation was applied in block k, $C_D(k)$ remains unchanged, otherwise it is updated according to the following formula

$$C'_D(k) = \begin{cases} (11)_2 & \text{if } |t| > T_\varepsilon, \\ (01)_2 \text{ bit\_or } C_D(k) & \text{if } |t| \leq T_\varepsilon, \end{cases} \tag{4.2}$$

where t is the computed tangent in that block. Routine `Reset_Convergence` zeroes out C_D, and `Check_Convergence` returns bitwise-or reduction of C_D. For that reduction on the GPU we used `thrust::reduce` method of the Thrust library [6].

The reduction result tells whether there were no rotations in a quasi-sweep because the columns are orthogonal up to the machine precision (i.e., $(00)_2$), or the quadratic convergence was detected (i.e., $(01)_2$), or we must proceed further since no convergence criteria were fulfilled (i.e., $(11)_2$). Otherwise, we stop the process.

It is worth noting that the convergence status could be accumulated by atomic bitwise-or global memory instruction [14], but this approach is significantly slower.

5 Numerical testing

We found it intriguing to hand-code the main GPUJACH1 kernels (`Jacobi_Step` and `Precompute_Data`) in the PTX instruction set and wrap it up with the CUDA Driver API. The host part is written in C99 and C++.

GPUJACH1 was compared to the sequential (with row cyclic pivot strategy) [23] and MPI-parallel block-oriented [21] one-sided hyperbolic Jacobi algorithms.

The parallel block-oriented algorithm is a blocked CPU version of the pointwise algorithm described here. In a step, instead of elements, (modified) modulus strategy reaches blocks. In the first step, inside each block A_P (formed of block columns $G_P := [G_i\ G_j]$ as $A_P = G_P^T G_P$), each element in the strict upper triangle of A_P is annihilated only once. In all other steps, only the elements of the off-diagonal block $G_i^T G_j$ of A_P are annihilated. In both cases the inner pivoting strategy is row-cyclic. To achieve both, matrix A_p is factored by the specially pivoted Cholesky factorization $P^T A_P P = R^T R$, and then the HSVD of R is computed and the transformation matrix is applied from the right on G_P. This blocking serves twofold purpose. The first is speeding up the process by keeping data (blocks or their parts) in the cache and utilizing BLAS 3 instead of BLAS 1 operations. The second is efficiency of the parallelization, since the number of parallel tasks is usually much smaller than the matrix order.

This approach is the fastest known Jacobi-type parallel algorithm for computing the HSVD of moderately sized matrices. Note that the pointwise algorithms are less amenable for application on a cluster, since they require heavy communication between parallel tasks, when they reside on different machines.

Testing is performed on GTX280, GTX275, TeslaC1060 graphics cards, and on two quad-core machines: Intel Xeon X5470 (sequential and 4-task parallel testing), and Intel Core i7 950 (8-task parallel testing, with hyperthreading). For reference purposes, the speed benchmarks (measured in GFlops) are given in Table 1.

Table 1 Various speed measures of the testing machines

	Machine Rpeak	Linpack Benchmark Rmax	Matrices of order 1000 × 1000		
			Intel MKL `dgemm`		Parallel block-oriented Hyperbolic Jacobi
			Sequential	Parallel	
Xeon X5470	53.328	44.581	12.267	42.535	29.951
Core i7-950	48.960	45.972	12.315	39.939	35.343

Table 2 Dimension of matrices n, and parameter a used in generation

n	≤ 3168	≤ 6368	≤ 9568	≤ 10144
a	20	30	40	50

Table 3 Speed of HSVD on GTX275 for matrices of order 4096 (with accumulation of V^{-T}) and different number of positive signs in J

Number of positive signs	0	1	5	10	50	100	500	1000	2048
Time (with sorting)	273	274	276	285	312	321	354	356	357
Quasi-sweeps (with sorting)	13	13	13	14	15	15	16	16	16
Time (without sorting)	306	307	308	321	333	349	360	372	368
Quasi-sweeps (without sorting)	13	13	13	14	14	15	15	16	16

For testing purposes, symmetric indefinite matrices with random, uniformly distributed spectra in $[-a, -a \cdot 10^{-5}\rangle \cup \langle a \cdot 10^{-5}, a]$, were generated by a modified LAPACK `dlagsy` routine, rewritten, with all its support LAPACK and BLAS routines, in 80-bit `extended` precision (vs. 64-bit `double`). Then, matrices were factorized by the symmetric indefinite factorization with complete pivoting [2] (`xsybpc`), also in `extended` precision. The factor G is then downcasted to `double`. Graphically:

$$\xrightarrow{\texttt{dlarnd}} \Lambda_{64} \xrightarrow{\text{upcast}} \Lambda_{80} \xrightarrow{\texttt{xlagsy}} M_{80} \xrightarrow{\texttt{xsybpc}} G_{80} \xrightarrow{\text{downcast}} G_{64}.$$

The `extended` support came from GNU Fortran with `real(kind=10)` datatype. The parameter a varied depending on the matrix order n as follows in Table 2.

Besides these cases, with approximately the same number of positive and negative signs in J, we have also experimented on matrices with different number of positive (negative) signs. The experiments confirm that the former cases are the slowest ones, while the definite cases (with zero, or all, signs positive) are the fastest. An example of this behavior is given in Table 3.

This behavior is well-known [5]. The reason lies in the fact that trigonometric transformations keep the trace of the matrix constant, while the hyperbolic transformations decrease the trace. In exchange, the hyperbolic transformations enable the high relative accuracy of computed eigenvalues, and make the two-norm of the spectral projector small when the eigenvalues have big relative gaps (see [24]).

The spectrum Λ was saved in all cases, to be compared with the computed eigenvalues Λ'. The orders of matrices were $160 + 128k$, with $0 \leq k \leq 78$. Large values of k were tested only on TeslaC1060 (with 4 GB of memory), and skipped on GTX275

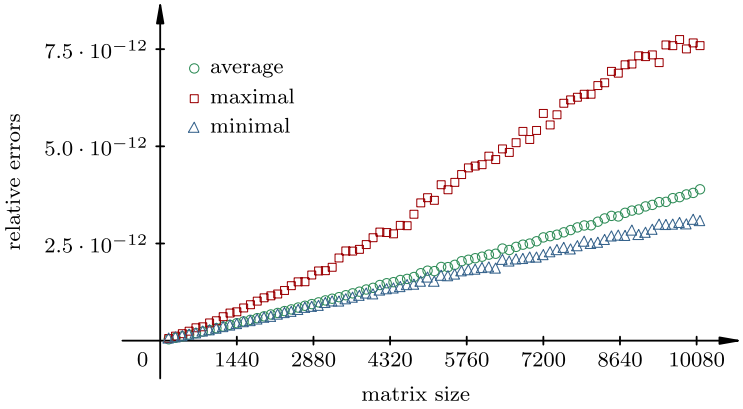

Fig. 5 Relative errors in the computed eigenvalues, $\frac{|\lambda_i - \lambda_i'|}{|\lambda_i|}$

and GTX280, due to the cards' insufficient memory sizes of 896 MB and 1 GB, respectively.

Relative errors in the computed eigenvalues (Fig. 5) are satisfactory in the context of high relative accuracy. Note that the average error is close to minimal.

Distance from orthonormality for the computed eigenvectors U is calculated as $d(U) = \|I - U^T U\|_F$ (see [12] for details). It grows linearly form 1.11×10^{-14} to 7.55×10^{-13} on full test spectrum. Row-distance $d(U^T)$ differs from $d(U)$, consistently with the bound $d(U) = d(U^T) + O(d^2(U))$, in the third digit, too little to be depicted.

GPUJACH1 timing started when all data were in place, and stopped when the algorithm converged, but before the final collection of data. The similar holds for the sequential and MPI-parallel timings. In all cases, and for all algorithms, accumulation of V^{-T} is included in the timings. If V^{-T} is not needed (e.g., for the eigenvalue problem), the speedups are a bit higher—more than $4\times$ in the 4-task case (for matrices of order $n \geq 4800$), and more than $1.25\times$ in the 8-task task case (for matrices of order $n \geq 8000$). The effect of diagonal sorting on the speed of the algorithm is also illustrated in Table 3. The similar speedup occurs on the other problem sizes.

Timing results on common test cases (Fig. 6) are perfectly consistent between GTX275 and GTX280. TeslaC1060 is a bit slower because of the lower clock frequencies, but it is the only choice in its generation for the large problems.

The speedup of GPUJACH1 vs. the sequential algorithm is shown in Fig. 7. The inevitable overhead of convergence checking (discussed in Sect. 4 and shown in Fig. 10) makes GPUJACH1 inappropriate for small matrices, but the speedup quickly stabilizes up to $17\times$ (at about $n = 4800$) in favor of GPUJACH1.

The speedup of GPUJACH1 vs. the parallel block-oriented algorithm (Figs. 8 and 9) is negligible (if none, i.e., CPU algorithm is faster), for matrices of order about 2000. One of the reasons is efficient caching the blocking algorithms were designed for, and GPUJACH1 had no caching opportunities whatsoever. When the CPU caches get too small for keeping the whole blocks without being frequently evicted, GPUJACH1 attains its peak speedup. The $3\times$ speedup over the 4-task case, com-

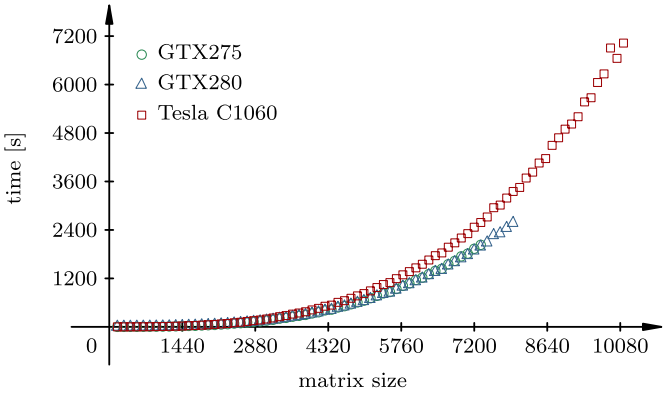

Fig. 6 Timing results of GPUJACH1 on square, full-rank matrices

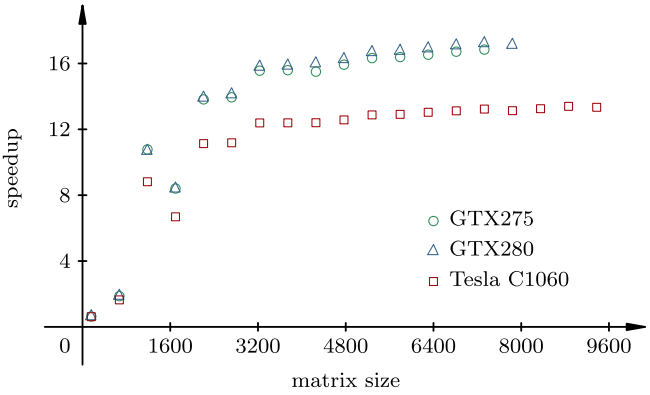

Fig. 7 Speedup of GPUJACH1 vs. the sequential algorithm (with accumulation of V^{-T})

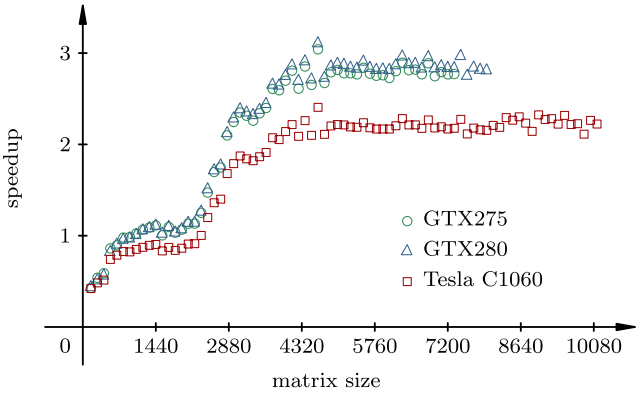

Fig. 8 Speedup of GPUJACH1 vs. the 4-task parallel block-oriented algorithm (with accum. of V^{-T})

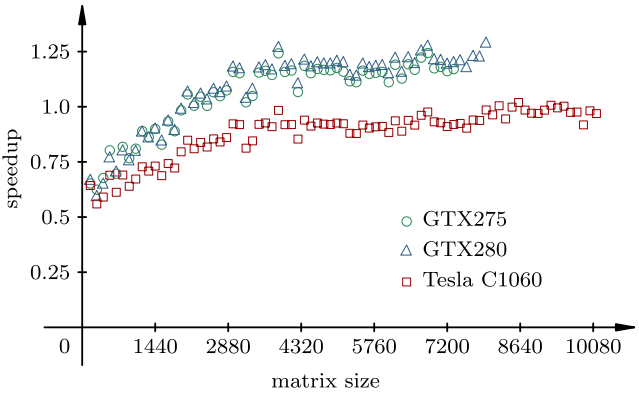

Fig. 9 Speedup of GPUJACH1 vs. the 8-task parallel block-oriented algorithm (with accum. of V^{-T})

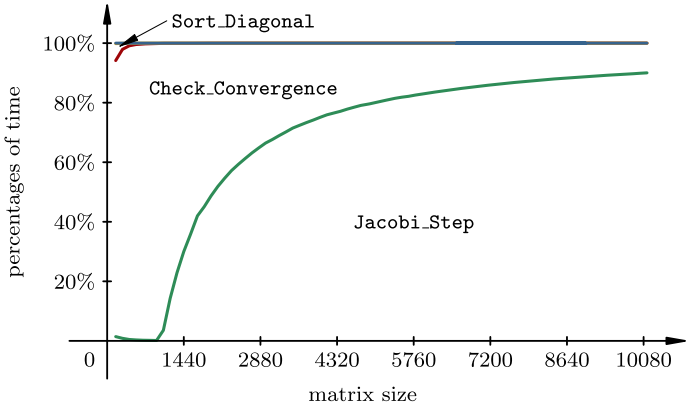

Fig. 10 Percentages of time

pared to 17× in the sequential case (if accumulation of V^{-T} is included) shows that the caching advantage of the block algorithm has dwindled.

GPUJACH1 algorithm pays off when `Jacobi_Step` routine takes a predominant part in the overall computation time. The convergence check is the main issue here, while the GPU sorting time is almost negligible, except for the very small problems. All the other routines take well below 0.1% of the time. See Fig. 10 for details.

The profiling results of our algorithm are shown in Table 4. The occupancy is not very high (5 blocks residing on a SM, instead of theoretical maximum of 8, because of the register file starvation), which indicates that the future effort should include simplifying the main `Jacobi_Step` kernel, maybe by breaking it down into a few lightweight kernels. However, the global memory throughput is satisfactory (71.91% of the theoretical limit of 102 GB/s on TeslaC1060).

We also adapted our algorithm to be able to run more (but still even) number of warps per block and to avoid the diagonal packing of (d, ρ, j) (see Fig. 3 and discussion in Sect. 4). However, the device code became more complex (i.e., uses 53

Table 4 Throughput and occupancy of HSVD on TeslaC1060 for matrices of order 4096 with 2048 positive/negative signs in J (with accumulation of V^{-T})

Computational kernel	Global memory throughput [GB/s]			Instruction throughput	Achieved occupancy	Register ratio
	Read	Write	Overall			
`Jacobi_Step`	50.0280	23.3173	73.3453	0.285392	0.3125	0.9375
`Precompute_Data`	84.3262	0.3294	84.6556	0.219198	0.5000	0.5000

Table 5 Speed of modified version of HSVD (without diagonal packing) on GTX275 for matrices of order 4096 with 2048 positive signs in J (with accum. of V^{-T}) and different number of warps per block

Number of warps	2	4	6	8
Time	394	429	589	544

registers and a couple of instructions more than GPUJACH1), and the timing results were disappointing (see Table 5).

A plausible explanation would be that the scheduling algorithm of NVIDIA GPUs works reasonably well, and that it puts as many blocks on a SM in a given time as possible. Also, the blocks complete execution when its slowest pair of warps does. If you have a pair of warps performing a trigonometric transformation, and another pair performing a hyperbolic one in the same block, the hyperbolic one will slow down the entire block. As a part of a block cannot be replaced on SM by a part of another block (only entire blocks are changeable), it seems reasonable to have only one pair of warps in a block, which turned out to be true. Therefore, having more pairs of warps per block could be viable if the computation of transformations, which causes imbalance of execution time, is isolated into a separate, lightweight kernel.

6 Conclusions and future work

GPUJACH1 is fast and usable in its own right. However, it can be incorporated into the hybrid CPU–GPU parallel Jacobi framework with ease. In [21], besides the block-oriented, the parallel full-block one-sided hyperbolic Jacobi algorithm is developed. The full-block algorithm diagonalizes its pivot block, while the block-oriented one only annihilates the off-diagonal elements of a block once. Instead of sequential diagonalization of a block in each MPI process, GPUJACH1 can be plugged in as a direct replacement. The speedup should be the same as the speedup of GPUJACH1 vs. the sequential algorithm, if each MPI process has access to its own GPU unit and block sizes are large enough.

On GT200 chips there is no cache for the global memory, and the shared memory is small. Thus, it is difficult to avoid the slow BLAS 1 operations, and no easy blocking is possible. On the other hand, Fermi chips have multiple levels of cache, and a larger shared memory. That gives opportunity to employ BLAS 3 operations for factorization in GPU and the shared-memory block diagonalization (the full-block way).

Our future work also includes the symmetric indefinite factorization for GPU, which is the missing key part to have a complete GPU-based symmetric indefinite Jacobi-type eigensolver.

Acknowledgements We would like to thank Prof. Većeslav Čorić from Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, for helping us acquiring a part of the testing equipment, and Prof. Vjeran Hari from the Department of Mathematics, University of Zagreb for helping us with the proof of the convergence of the modulus pivot strategy. We also thank the anonymous referees for their helpful suggestions, which substantially improved the paper.

Appendix

There are many parallel strategies worth studying. A bunch of them are described in [10, 11, 16]. An obvious choice of strategy is to take as many as possible, i.e., $\lfloor n/2 \rfloor$ independent pivot pairs for annihilation in each step, organized in $n-1$ steps in a sweep for even n and n steps for odd n. For many strategies which satisfy this property, the proof of the convergence is not known. Therefore, we loosen the requirement on the maximal number of independent pivot pairs, but still firmly require the convergence of the algorithm.

The modulus strategy (in two different shifted forms) was described in [16] and [4]. The name modulus strategy appears for the first time in [11], where on small examples, its equivalence to the row cyclic strategy is illustrated.

Here we prove that our form of the modulus strategy is weakly equivalent to the row cyclic strategy, and thus convergent. The convergence of the row cyclic strategy can be found in [25, Theorem 2.3].

Let us introduce a notation, taken from [5]. In a pivot strategy, let $I(i, j)$ be the index at which the pair (i, j) occurs. Adjacent pivot pairs (i, j) and (p, q) can swap their positions if $\{i, j\} \cap \{p, q\} = \emptyset$. This transposition of pairs is called an *admissible transposition*. Two pivot strategies O and O' are *equivalent* if one can be transformed to the other by a finite number of admissible transpositions.

Two pivot strategies O and O' are *shift equivalent*, if one is obtained from the other by a cyclic shift of the pivot pairs, i.e., if the position of each pair in strategy O is $I(i, j)$, its new position in strategy O' is

$$I'(i, j) := (I(i, j) + c) \mod n_p,$$

where $n_p = n(n-1)/2$ and $c \in \mathbb{Z}$.

Two pivot strategies O and O' are *weakly equivalent*, if there exist strategies O_i, $1 \le i \le r$, for some $r \ge 1$, such that in the sequence of strategies $O, O_1, \ldots, O_r, O'$ each two adjacent terms are either equivalent or shift equivalent. Obviously, a proof of convergence for one strategy in the sequence is sufficient to prove that all strategies in the sequence are convergent.

The convergence of an algorithm is usually proved for the two-sided algorithm. Since the one-sided transformations on a factor of a matrix are in theory (need not be numerically) the same as the two-sided algorithm, we immediately have the proof of convergence for the one-sided algorithm.

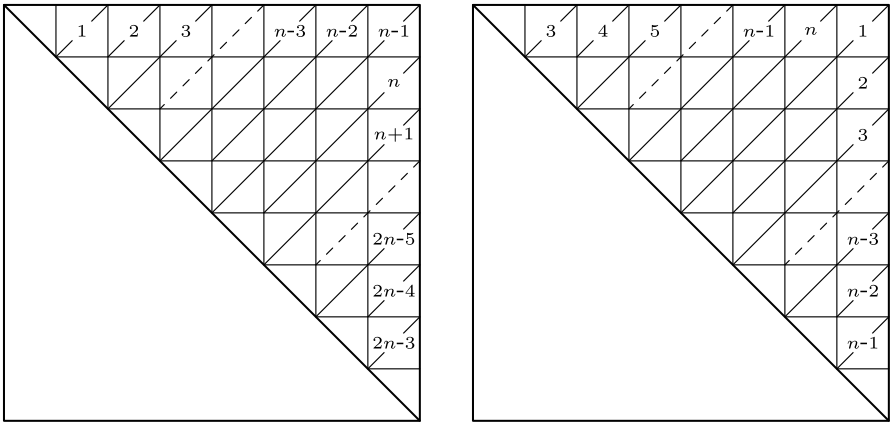

Fig. 11 The antidiagonal strategy (*left*), and the modulus strategy (*right*). The steps are labeled on each antidiagonal

Table 6 The order of annihilation in the antidiagonal strategy

Step	Simultaneously annihilated pivot pairs						
1	$(1,2)$						
2	$(1,3)$						
3	$(1,4)$	$(2,3)$					
4	$(1,5)$	$(2,4)$					
5	$(1,6)$	$(2,5)$	$(3,4)$				
$\vdots$	$\vdots$		$\vdots$				
$n-1$	$(1,n)$	$(2,n-1)$	$(3,n-2)$	$\cdots$	(k,ℓ)		
n		$(2,n)$	$(3,n-1)$	$\cdots$	$\begin{cases}(k+1,\ell), & n \text{ odd}\\ (k,\ell+1), & n \text{ even}\end{cases}$		
$\vdots$			$\ddots$		$\ddots$		
$2n-2$						$(n-2,n-1)$	$(n-2,n)$
$2n-3$							$(n-1,n)$

First we prove that the antidiagonal strategy is equivalent to the row cyclic strategy, and second that the modulus strategy is weakly equivalent to the antidiagonal strategy.

The antidiagonal strategy consists of the sequence of steps shown in Fig. 11 (left). The steps of the modulus strategy are shown in Fig. 11 (right).

Table 6 shows the order of annihilation in the antidiagonal strategy.

There is a minor difference between odd and even n in the step $n-1$ in Table 6, i.e.,

$$k = \begin{cases} \frac{n-1}{2}, & n \text{ odd},\\ \frac{n}{2}, & n \text{ even}, \end{cases} \qquad \ell = \begin{cases} k+2, & n \text{ odd},\\ k+1, & n \text{ even}. \end{cases}$$

By admissible transpositions, pivot pairs from the first column (pairs $(1, j)$) in Table 6 can be written before all pairs $(2, \cdot), (3, \cdot), \ldots, (k-1, \cdot)$, since pivot indices $(1, j)$, $j = 5, \ldots, n$ are disjoint with indices of these columns (the first index in $(1, j)$ is always smaller than the first index in other columns, and the second is always greater than the second index in other columns). The rest of the proof, for indices in the second, third, and other columns follows by induction over the column index. This completes the proof of equivalence of the antidiagonal and the row cyclic strategy.

The second step in the proof is to show that the antidiagonal and the modulus strategy are weakly equivalent.

According to Fig. 11 the modulus strategy consists of either one or two steps of the antidiagonal strategy. Note that the antidiagonal strategy, which consists of steps

$$O = (1, 2, 3, \ldots, n-2, n-1, n, n+1, n+2, \ldots, 2n-4, 2n-3),$$

is shift equivalent to the strategy which consists of steps

$$O_1 = (n-1, n, n+1, n+2, \ldots, 2n-4, 2n-3, 1, 2, 3, \ldots, n-2).$$

The pivot indices in step 1 (pair $(1, 2)$) are disjoint with the indices contained in steps $n+2, \ldots, 2n-3$ since the first index is at least 3, and the second is at least $\ell + 1 > 2$. Thus, the cyclic strategy O_1 is equivalent to

$$O_2 = (n-1, n, n+1, 1, n+2, \ldots, 2n-4, 2n-3, 2, 3, \ldots, n-2).$$

The similar reasoning holds for step 2, with indices disjoint with indices in steps $n+3, \ldots, 2n-3$, indices from step 3 are disjoint with indices in steps $n+4, \ldots, 2n-3$, and so on.

The final sequence of steps is

$$O' = (n-1, n, n+1, 1, n+2, 2, \ldots, 2n-3, n-2),$$

which is in fact the modulus strategy, since step $n-1$ of the antidiagonal strategy is the first step of the modulus strategy, step n is the second step of the modulus strategy, the third step of the modulus strategy consists of the steps $n+1, 1$ from the antidiagonal strategy, and so on.

This proves that the modulus strategy is weakly equivalent to row cyclic strategy.

References

1. Brent, R.P., Luk, F.T.: The solution of singular–value and symmetric eigenvalue problems on multiprocessor arrays. SIAM J. Sci. Stat. Comput. **6**(1), 69–84 (1985)
2. Bunch, J.R., Parlett, B.N.: Direct methods for solving symmetric indefinite systems of linear equations. SIAM J. Numer. Anal. **8**(4), 639–655 (1971)
3. Drmač, Z., Hari, V.: On quadratic convergence bounds for the J-symmetric Jacobi method. Numer. Math. **64**(1), 147–180 (1993)
4. Hari, V., Veselić, K.: On Jacobi methods for singular value decompositions. SIAM J. Sci. Stat. Comput. **8**(5), 741–754 (1987)

5. Hari, V., Singer, S., Singer, S.: Block-oriented J-Jacobi methods for Hermitian matrices. Linear Algebra Appl. **433**(8–10), 1491–1512 (2010)
6. Hoberock, J., Bell, N.: Thrust: a parallel template library. Version 1.3.0 (2010). URL http://www.meganewtons.com
7. IEEE: IEEE standard for floating–point arithmetic (2008). doi:10.1109/IEEESTD.2008.4610935
8. The Khronos Group Inc.: The OpenCL specification 1.1 (2010)
9. Lahabar, S., Narayanan, P.J.: Singular value decomposition on GPU using CUDA. In: Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009), Rome, Italy, May 23–29, 2009
10. Luk, F.T., Park, H.: On parallel Jacobi orderings. SIAM J. Sci. Stat. Comput. **10**(1), 18–26 (1987)
11. Luk, F.T., Park, H.: A proof of convergence for two parallel Jacobi SVD algorithms. IEEE Trans. Comput. **C–38**(6), 806–811 (1989)
12. Mathias, R.: Analysis of algorithms for orthogonalizing products of unitary matrices. Numer. Linear Algebra Appl. **3**(2), 125–145 (1996)
13. NVIDIA Corp.: NVIDIA CUDA C programming guide 3.1.1 (2010)
14. NVIDIA Corp.: PTX: parallel thread execution ISA version 2.1 (2010)
15. Sachdev, G.S., Vanjani, V., Hall, M.W.: Takagi factorization on GPU using CUDA. In: 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10), Knoxville, Tennessee, July 13–15, 2010. URL http://saahpc.ncsa.illinois.edu/10/papers/paper_19.pdf
16. Sameh, A.H.: On Jacobi and Jacobi-like algorithms for a parallel computer. Math. Comput. **25**(118), 579–590 (1971)
17. Singer, S.: Computing the eigendecomposition of a positive definite matrix. Master's thesis, Dept. of Mathematics, University of Zagreb (1993) (in Croatian)
18. Singer, S.: Indefinite QR factorization. BIT **46**(1), 141–161 (2006)
19. Singer, S., Singer, S., Hari, V., Bokulić, K., Davidović, D., Jurešić, M., Uščumlić, A.: Advances in speedup of the indefinite one-sided block Jacobi method. In: Simos, T.E., Psihoyios, G., Tsitouras, C. (eds.) AIP Conf. Proc. Numerical Analysis and Applied Mathematics, vol. 936, pp. 519–522. AIP, New York (2007)
20. Singer, S., Singer, S., Novaković, V., Davidović, D., Bokulić, K., Uščumlić, A.: Three-level parallel J-Jacobi algorithms for Hermitian matrices. Technical report, University of Zagreb (2010). URL arXiv:1008.4166 [cs.NA]
21. Singer, S., Singer, S., Novaković, V., Uščumlić, A., Dunjko, V.: Novel modifications of parallel Jacobi algorithms. Numer. Algorithms (2011). doi:10.1007/s11075-011-9473-6
22. Slapničar, I.: Componentwise analysis of direct factorization of real symmetric and Hermitian matrices. Linear Algebra Appl. **272**, 227–275 (1998)
23. Slapničar, I.: Highly accurate symmetric eigenvalue decomposition and hyperbolic SVD. Linear Algebra Appl. **358**, 387–424 (2003)
24. Slapničar, I., Veselić, K.: Perturbations of the eigenprojections of a factorized Hermitian matrix. Linear Algebra Appl. **218**, 273–280 (1995)
25. Veselić, K.: A Jacobi eigenreduction algorithm for definite matrix pairs. Numer. Math. **64**(1), 241–269 (1993)
26. Zha, H.: A note on the existence of the hyperbolic singular value decomposition. Linear Algebra Appl. **240**, 199–205 (1996)
27. Zhang, S., Dou, H.: Matrix singular value decomposition based on computing unified device architecture. J. Appl. Res. Comput. **24**(6), 4 (2007) (in Chinese)