Applying OOC Techniques in the Reduction to Condensed Form for Very Large Symmetric Eigenproblems on GPUs

Davor Davidović
Rudjer Bošković Institute
Centre for Informatics and Computing
Bijenička street 54, 10000–Zagreb, Croatia
davor.davidovic@irb.hr

Enrique S. Quintana-Ortí
Universitat Jaume I
Depto. de Ingeniería y Ciencia de Computadores
Av. Vincent Sos Baynat, 12.071–Castellón, Spain
quintana@icc.uji.es

Abstract

In this paper we address the reduction of a dense matrix to tridiagonal form for the solution of symmetric eigenvalue problems on a graphics processor (GPU) when the data is too large to fit into the accelerator memory. We apply out-of-core techniques to a three-stage algorithm, carefully redesigning the first stage to reduce the number of data transfers between the CPU and GPU memory spaces, maintain the memory requirements on the GPU within limits, and ensure high performance by featuring a high ratio between computation and communication.

1 Introduction

A few applications arising, e.g., in molecular dynamics, computational quantum chemistry, finite element modeling and multivariate statistics, require the solution of large-scale dense eigenproblems of the form:

$$AX = \Lambda X, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is symmetric, $\Lambda \in \mathbb{R}^{n \times n}$ is a diagonal matrix with the sought-after eigenvalues, and $X \in \mathbb{R}^{n \times n}$ contains the associated eigenvectors [5]. When a significant fraction of the eigenvalues are desired, efficient and numerically stable solvers first reduce A to tridiagonal form,

$$Q^T AQ \to T, \tag{2}$$

where $T \in \mathbb{R}^{n \times n}$ is tridiagonal and $Q \in \mathbb{R}^{n \times n}$ stands for the applied orthogonal transforms [5], to then obtain the eigenvalues/eigenvectors of T using, e.g. the $\texttt{MR}^3$ algorithm [4]. In case the eigenvectors of the original matrix are also desired, a back-transform is necessary to recover them from those of T.

In this paper we focus on the reduction to tridiagonal form via orthogonal transforms. There exist basically two alternatives to perform this reduction. The *one-stage approach* computes a sequence of $n-2$ Householder (orthogonal) transforms [5] which annihilate the entries below the first sub-diagonal. Overall the algorithm requires $4n^3/3$ flops but, unfortunately, half of them are cast in terms of the symmetric matrix-vector product, a memory-bounded operation which renders the global algorithm quite inefficient on current general-purpose architectures. The memory bottleneck is partially alleviated in [7] where the authors leverage the higher memory bandwidth of the GPU to report an important increase in the performance of this algorithm.

The *multi-stage approaches* [2] diminish the number of level 2 BLAS flops in the reduction to tridiagonal form in exchange for an increment in the computational cost. In particular, these algorithms first compute

$$Q_1^T AQ_1 \to B_1, \tag{3}$$

where $B_1 \in \mathbb{R}^{n \times n}$ is a matrix of bandwidth w_1 and $Q_1 \in \mathbb{R}^{n \times n}$ collects the corresponding orthogonal transforms [5]. In the two-stage variant, this band matrix is then reduced to tridiagonal form

$$Q_2^T B_1 Q_2 \to T, \tag{4}$$

so that $Q = Q_1 Q_2 \in \mathbb{R}^{n \times n}$ yields the reduction/orthogonal transform in (2). Alternatively, a truly multi-stage algorithm can be employed to successively transform A into a series of matrices of narrower band, $w_1 > w_2 > w_3, \ldots, w_{m-1}$. In [1] the authors show the potential of the two-stage algorithm to leverage the data-parallel architecture of a GPU. In [6] the same approach is show to outperform the one-stage procedure on multi-core processors as well.

All previous work for the reduction of a symmetric matrix to tridiagonal form on a hybrid CPU-GPU platform assume that the matrix fits into the memory of the GPU.

However, the storage capacity of as-of-today GPUs is relatively small, typically between 1.5 and 3 Gbytes, so that the dimension of the matrices that they can hold is limited (roughly $n \approx$ 13,500-19,000 for real double-precision data). For very large problems, as those appearing in molecular dynamics or computational quantum chemistry, this is clearly insufficient since, in these applications, n can be as large as 300,000. The main contribution of this paper is a reformulation of the multi-stage tridiagonalization algorithm that is not constrained by the dimension of the memory of the GPU. In particular, we maintain the whole matrix A in the main memory of the system, and carefully orchestrate the data transfers with the device (GPU) to hide the latency of the (PCI-Express) interconnect between the two memory spaces. As a result, the limitation on the dimension of the problems that can be solved using hardware accelerators is overcome while delivering high performance. Our solution can be viewed as the result of applying Out-of-Core (OOC) techniques to deliver sustainable (scalable) performance in the reduction to tridiagonal form, where the (in-)core memory corresponds to the device storage while the secondary (out-of-core) storage is the main memory. In this article we refer to the scalability as the ability to handle growing matrix sizes on the fixed architecture.

In our approach, we choose the bandwidth w_1 so that the whole matrix B_1 in (4), when saved in packed storage using a $w_1 \times n$ array, fits into the GPU memory. In the practical applications that we target, n ranges between 100,000 and 300,000 so that w_1 can still be chosen large enough to hide the interconnect latency. Thus, the application of OOC techniques is only necessary in the first stage.

The paper is structured as follows. In the next section we review the algorithm in the SBR toolbox [2] for the orthogonal reduction of a dense symmetric matrix to band form which is the basis for the software that we have developed. In Section 3 we introduce hybrid CPU-GPU algorithms for the computation of the QR factorization of a rectangular panel and the update of a symmetric matrix involved in the reduction to band form. The hybrid reduction of the band matrix to narrower band form is addressed next, in Section 4. These codes are evaluated on a system equipped with an Intel-based multi-core processor and an NVIDIA "Fermi" GPU in Section 5. A few remarks close the paper in Section 6.

2 The SBR Toolbox

SBR (Successive Band Reduction) is a software package for the reduction of dense symmetric matrices to band form (SYRDB) and the reduction of band matrices to narrower band (SBRDB) or tridiagonal form (SBRDT). Accumulation of the orthogonal transformations and repacking routines for storage rearrangement are also provided in the toolbox. We next describe routine SYRDB for the reduction of a dense matrix to band form that performs the initial transformation in (3).

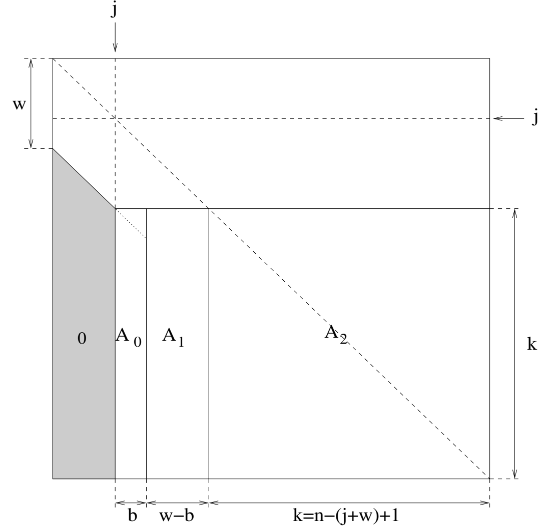

Figure 1. Partitioning of the matrix during one iteration of routine SYRDB.

Consider that the first $j-1$ columns of the matrix A have been already reduced to band form with bandwidth w. Let b denote the algorithmic block size, and assume for simplicity that $j+w+b-1 \leq n$, and n, w are integer multiples of w, b, respectively; see Figure 1. Then, during the current iteration of routine SYRDB, b new columns of the band matrix are computed as follows:

1. Compute the QR factorization of $A_0 \in \mathbb{R}^{k \times b}$, $k = n-(j+w)+1$:
$$A_0 = Q_0 R_0, \quad (5)$$
where $R_0 \in \mathbb{R}^{b \times b}$ is upper triangular and the orthogonal factor Q_0 is implicitly stored as a sequence of b Householder vectors. The cost of this first step is $2b^2(k-b/3)$ flops.

2. Construct the factors of the compact WY representation [5] of the orthogonal matrix $Q_0 = I_k + WY^T$, with $W,\ Y \in \mathbb{R}^{k \times b}$. The cost of this step is kb^2 flops.

3. Apply the orthogonal matrix to $A_1 \in \mathbb{R}^{k \times w-b}$:
$$A_1 := Q_0^T A_1 = A_1 + Y(W^T A_1). \quad (6)$$
The cost of this step becomes $4kb(w-b)$ flops. In case the bandwidth equals the block size ($w = b$), A_1 comprises no columns and, therefore, no operation is performed in this step.

4. Apply the orthogonal matrix to $A_2 \in \mathbb{R}^{k\times k}$:

$$\begin{aligned} A_2 &:= Q_0^T A_2 Q_0 \\ &= A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T. \end{aligned} \quad (7)$$

During this step only the lower (or the upper) triangular part of A_2 is updated as follows:

$$\begin{aligned} &(\text{SYMM}) & X_1 &:= A_2 W, & (8)\\ &(\text{GEMM}) & X_2 &:= X_1^T W/2, & (9)\\ &(\text{GEMM}) & X_3 &:= X_1 + YX_2, & (10)\\ &(\text{SYR2K}) & A_2 &:= A_2 + X_3 Y^T + YX_3^T. & (11) \end{aligned}$$

The major cost of this step is in the computation of the symmetric matrix product (8) and the symmetric rank-$2k$ update (11), each with a cost of $2k^2b$ flops. The global cost of this step is $4k^2b + 4kb^2$, which is higher than those of the Steps 1–3.

Provided b and w are both small compared with n, the global cost of the reduction of a full matrix to band form is $4n^3/3$ flops. The bulk of the computation is cast in terms of the BLAS-3 operations in (8) and (11), so that high performance can be expected from routine SYRDB in case a tuned implementation of BLAS is used.

3 OOC Reduction to Band Form

Our algorithm for the OOC reduction to band form on a hybrid CPU-GPU platform is based on the SBR reduction to band form. To achieve high utilization and performance in the OOC algorithm, we have to operate with wide factors W and Y. As the column-size of these panels is upper-bounded by the algorithmic block size b, we set the $w = b$ so that our algorithm only consists of steps 1, 2 and 4 as in Section 2.

Algorithm 1 illustrates the two phases of the reduction procedure. The first phase (line 2) performs a blocked QR decomposition of $\tilde{A} = A(j + w\colon n, j\colon j + w - 1)$ while simultaneously constructing the factors W and Y (steps 1 and 2). The second phase (line 3) computes the two-sided update $\hat{A} = A(j + w\colon n, j + w\colon n) := Q^T \hat{A}\, Q$, with $Q = I + WY^T$ (step 4).

Algorithm 1 Reduction to band form

Input: Real symmetric matrix $A \in \mathbb{R}^{n\times n}$, bandwidth w
Output: A overwritten with the resulting band-matrix

1: **for** $j := 1,\ j < n,\ j := j + w$ **do**
2: QR decomposition of $\tilde{A} \to W, Y$
3: Two-sided update: $\hat{A} := (I + WY^T)^T \hat{A}(I + WY^T)$
4: **end for**

Subsection 3.1 introduces a hybrid algorithm where CPU and GPU collaborate to obtain the QR factorization of $\tilde{A}$ and the construction of the corresponding W and Y. Given our premises, we can safely assume that $\tilde{A}$ fits into the GPU storage so that this algorithm operates in-core. Subsection 3.2 describes the hybrid OOC update of the trailing $k \times k$ block $\hat{A}$.

In our implementations we do not apply overlapping between computation and copying. Our research aims at increasing the ratio between these two factors by reducing the number of transfers and at the same time increasing the amount of computation per GPU kernel. The overlapping of copying with the computation is among future research.

3.1 Hybrid (in-core) QR decomposition

Consider the computation of the QR factorization of the $k \times w$ matrix $\tilde{A}$, $k = n - (j + w) + 1$, and the construction of the corresponding factors of the same dimension. For clarity, in the following discussion $(d)W_i$ stands for $(d)W(i\colon k, i\colon i + \tilde{b} - 1)$ and $(d)Y_i$ for $(d)Y(i\colon k, i\colon i + \tilde{b} - 1)$, where $\tilde{b}$ is the algorithmic block dimension for this phase. The prefix "*d*" identifies matrices that are stored in the GPU memory; all operations on these matrices imply the execution on the GPU without further explicit mention of it. For simplicity, we also assume that w is an exact multiple of $\tilde{b}$.

Our implementation of the hybrid QR decomposition is illustrated in Algorithm 2. The procedure operates on column blocks of width $\tilde{b}$. Consider that the first $i - 1$ columns of $\tilde{A}$ have already been factorized. During the current iteration, we copy the first $\tilde{b}$ columns starting from the $d\tilde{A}(1, i)$ of the GPU to the CPU (line 3), obtain QR factorization on the CPU (line 4), and generate the factors W_i and Y_i there (line 5). Then, W_i and Y_i are transferred to the GPU (line 7) and, to complete the iteration, sub-matrix $d\tilde{A}(i\colon k, i + \tilde{b}\colon k)$ is updated on the GPU from the left (line 8), and the previous transforms are applied to the current factor dW_i (line 9). At the end of the for-loop the updated $\tilde{A}$ and Y reside on both the CPU and the GPU, while dW is stored on the GPU and has to be copied back to the CPU. The cost of this algorithm is $4(k^2w - kw^2 + w^3/3)$ flops.

We have implemented two versions of this algorithm, which differ in the number of data transfers between GPU and CPU and the amount of workspace needed in the GPU memory.

3.1.1 Variant QR-1

This version requires space on the GPU to hold two matrices of size $k \times w$ (with $k \leq n$), for $d\tilde{A}$ and dW, and an additional workspace dZ of size $w \times w$. At each iteration, the factor dY_i is stored overwriting the entries of dA_i since, once dA_i is copied to $\tilde{A}_i$ at the beginning of each iteration, it is not referenced anymore. The update of the

Algorithm 2 Hybrid QR decomposition

Input: Real matrix $\tilde{A} \in \mathbb{R}^{k\times w}$, block size $\tilde{b}$
Output: $\tilde{A}$ overwritten with factor R. $W, Y \in \mathbb{R}^{k\times w}$

1: Copy $\tilde{A} \rightarrow dA$
2: **for** $i := 1,\ i < w,\ i := i + \tilde{b}$ **do**
3: Copy $dA(:, i\colon i+\tilde{b}-1) \rightarrow \tilde{A}(:, i\colon i+\tilde{b}-1)$
4: (GEQRL) Compute QR factorization of block $\tilde{A}_i$
5: (GEWYG) Construct W_i and Y_i
6: $W(1\colon i-1, i\colon i+\tilde{b}-1) := 0$
$Y(1\colon i-1, i\colon i+\tilde{b}-1) := 0$
7: Copy $W_i \rightarrow dW_i$, and $Y_i \rightarrow dY_i$
8: $dA(i\colon k, i+\tilde{b}\colon w) :=$
$(I + dW_i dY_i^T)^T dA(i\colon k, i+\tilde{b}\colon w)$
9: $dW_i := dW_i + dW(i\colon k, 1\colon i-1)\times$
$dY(i\colon k, 1\colon i-1)^T dW_i$
10: **end for**
11: Copy $dW \rightarrow W$

rest of dA (line 8) is split into two parts, with each one being performed via a single invocation of the GEMM routine (matrix-matrix multiplication). Specifically, dZ is used to temporarily hold the partial result of the first matrix product $dZ := dW_i^T dA(i\colon k, i+\tilde{b}\colon w)$ while, in the second product, $dA(i\colon k, i+\tilde{b}\colon w) := dA(i\colon k, i+\tilde{b}\colon w) + dY\ dZ$. The application of the previous transforms to the current W_i can also be divided into two parts, being obtained with two calls of the GEMM routine as both $dW(i\colon k, 1\colon i-1)$ and $dY(i\colon k, 1\colon i-1)$ reside on the GPU.

The advantage of this version is that it performs a reduced number of data transfers to/from the GPU. However, it requires the full $d\tilde{A}$ and dW to reside on the GPU so that, as the matrix dimension n grows, the bandwidth w has to be conformally reduced so that they fit into the GPU memory. For very large matrices, this results in operating with long, narrow column blocks of $\tilde{A}$ as well as narrow factors W_i and Y_i, which may decrease the ratio between computation and transfers, and impair overall performance.

3.1.2 Variant QR-2

This version requires less storage space on the GPU: a workspace of size $k \times w$ for $d\tilde{A}$, a panel of size $k \times \tilde{b}$ for a single column block of dY, and an additional workspace dZ of size $w \times w$. The factor dW_i is now stored overwriting dA once the entries of this panel have been copied to $\tilde{A}_i$ (in the previous version we did this for dY_i).

The update of the rest of the matrix $d\tilde{A}$ (line 8) is performed in the same way as in the previous version using dZ as a temporary workspace. As we do not hold the full factor Y in the GPU memory in this variant, the application of the previous transformations to the current W_i differs from that of the previous version. In particular, this computation is split into the following two parts:

$$dZ := dY(i\colon k, 1\colon i-1)^T\ dW_i, \quad (12)$$

$$dW_i := dW_i + dW(i\colon k, 1\colon i-1)\ dZ, \quad (13)$$

where dW is stored in $dA(1\colon k, 1\colon i+\tilde{b}-1)$. The update (12) is performed by copying one-by-one the column blocks $Y(i\colon k, p\colon p+\tilde{b}-1)$ to dY and then computing $dZ(p\colon p+\tilde{b}-1, :) := dY^T\ dW_i$, for $p = 1, \tilde{b}+1, 2\tilde{b}+1, \ldots, i-\tilde{b}$. Once all blocks of Y_j have been processed and the full dZ is computed, the update in (13) can be performed by single call of GEMM.

The advantage of this approach is that it requires less memory on the GPU and, therefore, the bandwidth (i.e. column width of matrix $\tilde{A}$ and the factors) can be larger. However, the workspace on the GPU is still proportional to number of rows in the panel k and, as the problem size grows, the bandwidth w has to be conformally reduced. Nevertheless, for our test cases, this version permits the solution of symmetric eigenproblems in the desired range, with n up to 300,000 (provided it fits into the CPU memory). Compared with the previous version, the drawback of the Variant QR-2 is that it requires more transfers to/from the GPU as, for each p, one panel of size $(k-i) \times \tilde{b}$ is copied to the GPU, which results in additional overhead.

3.2 Hybrid OOC two-sided update

The OOC algorithm for the two-sided update of the symmetric trailing matrix $\hat{A} = A(i+w\colon n, i+w\colon n)$ implements step 4 from Section 2. The update is divided into 4 operations, see (8)–(11), and each one of these has to be reimplemented to work with out-of-core data (i.e., matrices that reside out of the GPU memory).

Consider a partitioning of matrix $\hat{A}$ into square blocks of size $\hat{b} \times \hat{b}$, with the the algorithmic block size for this phase, $\hat{b}$, being fixed to be small enough so that a "few" blocks fit into the GPU memory (check in Subsections 3.2.1 and 3.2.2 for the exact number). For simplicity, we introduce $\hat{A}_{ip}$ as substitution for the block $\hat{A}(i\colon i+\hat{b}-1, p\colon p+\hat{b}-1)$, W_i and Y_i for $W(i\colon i+\hat{b}-1, 1\colon w)$ and $Y(i\colon i+\hat{b}-1, 1\colon w)$ respectively, and X_i for $X(i\colon i+\hat{b}-1, 1\colon w)$.

We have implemented two versions of the OOC two-sided update matrix which differ in the requirements of storage in the GPU. This affects the performance of the global algorithm, especially when the size of the matrix grows.

3.2.1 Variant Update-1

This version requires two panels of size $k \times w$, and one block of size $\hat{b} \times \hat{b}$. The block size $\hat{b}$ is calculated at the beginning of the algorithm so that these data fit into the GPU memory.

Consider first the computation of X_1 in equation (8). The OOC implementation for this operation is given in Algorithm 3. Note that, because of the symmetry of $\hat{A}$, only the lower triangle of this matrix is referenced. Thus, once we copy the block $\hat{A}_{ip}$ to the GPU, we can update both X_i and X_p (lines 9–10). The diagonal blocks are updated in advance (lines 2–4).

Algorithm 3 OOC version 1: $X_1 := \hat{A}W$

Input: Real symmetric matrix $\hat{A} \in \mathbb{R}^{k\times k}$, $W \in \mathbb{R}^{k\times w}$
Output: Real matrix $X_1 \in \mathbb{R}^{k\times w}$

```
1:  Copy W → dW
2:  for p := 1, p < n, p := p + b̂ do
3:      Copy Â_pp → dA
4:      dX_p := dA dW_p
5:  end for
6:  for p := 1, p < n, p := p + b̂ do
7:      for i := p + b̂, i < n, i := i + b̂ do
8:          Copy Â_ip → dA
9:          dX_p := dX_p + dA^T dW_i
10:         dX_i := dX_i + dA dW_p
11:     end for
12: end for
```

The next two operations, corresponding to equations (9) and (10), can be computed with two calls to the GEMM routine provided X_1, X_3 and W (Y) are small enough to fit into the GPU memory. Note that the factor Y has to be copied to the GPU before the third operation (10) (we can reuse the storage space dW).

After the first step, both matrices W and X_1 are stored in the GPU memory, while matrix $\hat{A}$ is unchanged. In the second phase, X_2, of size $w \times w$, is stored in the workspace dA as w is chosen so that $w \leq \hat{b}$. For the third step, X_3 overwrites dX; therefore no additional workspace on the GPU is required.

The last operation, corresponding to (11), is performed by copying the blocks in the lower triangular part of $\hat{A}$ to the GPU and updating them as:

$$\hat{A}_{ip} := \hat{A}_{ip} + (X_3)_i Y_p^T + Y_i (X_3)_p^T. \qquad (14)$$

Now, as the block size $\hat{b}$ is chosen so that only one block $\hat{A}_{ip}$ fits into the GPU memory, after each update the corresponding block has to be retrieved back to the CPU.

The advantage of this version is that we keep dW and one of X_1, X_2 or X_3 in the GPU memory, and only the block $\hat{A}_{ip}$ is copied to/from the GPU. This reduces the number of memory transfers and enables data reuse once it resides on the GPU. However, with the increase of the n, the block size/bandwidth have to be reduced which leads to a loss of efficiency as the number of transfers grows and the algorithm operates on smaller blocks.

3.2.2 Variant Update-2

The second version of the algorithm requires less memory space on the GPU, but performs a higher number of memory transfers to the GPU. Nevertheless, the memory transfers can be compensate with the computation as the block size remains large enough, as presented in the section 5.

This version requires one block of size $\hat{b}\times\hat{b}$ to keep block $\hat{A}_{ip}$, one panel of size $k \times w$, and one additional panel of size $\hat{b} \times w$. Therefore, this version requires less storage than the previous variant so that the block size/bandwidth can be larger. The drawback of the new version is that it increases the number of memory transfers as we will now hold only one factor, W or Y, in the GPU memory, while X_1 or X_3 will be transferred by blocks.

Consider first the computation in (8). As we keep only one block of size $\hat{b} \times w$ of X_1 on the GPU, some changes are required to Algorithm 3 in lines 5–11. These lines are exchanged with the lines presented in Algorithm 4. Each

Algorithm 4 OOC version 2 difference to version 1

```
1: for i := 1, i < n, i := i + b̂ do
2:     if i ≠ p then
3:         Copy Â_ip → dA
4:         dX := dX + dA dW_p
5:     end if
6: end for
7: Copy dX → X_1(p: p + b̂ − 1, : )
```

block $\hat{A}_{ip}$ has to be transferred to the GPU which results in twice as much copying as occurs in OOC version 1. In each pass of p-loop one block of $\hat{b}$ rows X_1 (stored in dX) is calculated (line 8). As we do not have storage space on the GPU to keep full X_1, the block has to be saved in the CPU memory. Therefore, an additional workspace on the CPU of size $k \times w$ is required.

At the beginning of the second step/equation (9), the matrix X_1 is stored on the CPU and, therefore, $X_1(i\colon i + \hat{b} - 1, 1\colon w)$ is copied to dX block-by-block, and then X_2 is updated as $X_2 := X_2 + \frac{1}{2}\ dX^T\ dW_i$, for $i = 1, \hat{b}+1, 2\hat{b}+1, \ldots, k-\hat{b}$. Block dA is used to keep X_2, so no additional workspace is necessary. The only requirement is that $\hat{b} \geq w$, so the dA can hold the $w \times w$ block X_2.

Before equation (10), factor Y is saved in dW (as the later is not required in future computations). Then, $X_1(i\colon i + \hat{b} - 1, 1\colon w)$ is copied to dX and the block $X_3(i\colon i+\hat{b}-1, 1\colon w)$ is computed as $dX := dX + dW_i\ dA$, for $i = 1, \hat{b} + 1, 2\hat{b} + 1, \ldots, k - \hat{b}$. Once each block is updated, the resulting dX is transferred to $X_3(i\colon i + \hat{b} - 1, 1\colon w)$. X_3 is then saved in the same workspace as X_1 (X_1 is not required anymore).

In the last step, corresponding to equation (11), blocks $X_i = X_3(i\colon i + \hat{b} - 1, 1\colon w)$ and $X_p = X_3(p\colon p + \hat{b} -$

$1, 1\colon w)$ are needed in order to update each block $\hat{A}_{ip}$. Thus, two additional transfers of blocks to the GPU are required to update A_{ip}, as shown in Algorithm 5. can be updated as follows:

Algorithm 5 OOC version 2: Symmetric 2k update

1: Copy $\hat{A}_{ip} \rightarrow dA$
2: Copy $X_i \rightarrow dX$
3: $dA := dA + dX\, dY_p^T$
4: Copy $X_p \rightarrow dX$
5: $dA := dA + dY_i\, dX^T$
6: Copy $dA \rightarrow \hat{A}_{ip}$

4 Remaining Stages

In this section, we briefly describe the hybrid CPU-GPU implementation of the reduction of the band matrix resulting from the initial stage. We focus on how to exploit the GPU in this second stage, organizing the computations carefully to reduce the number of data transfers between the memory spaces of the host and the GPU.

Our algorithm is a hybrid CPU-GPU implementation of routine SBRDB from SBR. For a full description of the corresponding CPU algorithm, refer to [3].

The algorithm operates on the band matrix B stored into a $w_1 \times n$ array, where the w_1 is the band size and n is the dimension of the matrix. The main diagonal occupies the first row of the array, and the second, third, ... subdiagonals are saved in successive rows of this matrix. In other words, if $\hat{B}$ is a band storage representation of B, then each element $B(i, j)$ is saved in entry $\hat{B}(1 + i - j, j)$.

Assume the $n \times n$ band matrix B_{w_1}, of bandwidth w_1, is to be transformed into a band matrix B_{w_2}, with narrower bandwidth $w_2 < w_1$. The goal is thus to annihilate the elements of $d = w_1 - w_2$ subdiagonals. The procedure is based on a "annihilate-and-chase" strategy and employs Householder transforms for this purpose. The first step annihilates the d outermost subdiagonals from the first n_b columns. In particular, consider Figure 2 (left). After the computation of the QR factorization of the block labelled as "QR", the transforms accumulated in $Q = I + WY^T$ are applied from the left to block "PRE", from both sides to the symmetric block "SYM" (only the lower triangular part is actually updated/referenced), and from the right to block "POST".

The application of the transforms to block "POST" fills-in d diagonals below the band. Next, the first n_b columns of the space that was filled-in –see Figure 2 (right)– are removed by a second QR decomposition, and the corresponding transforms are applied to the blocks marked with "PRE", "SYM", and "POST" in the figure. This generates new fill-in below the "POST" block, and the process is repeated, chasing down the fill-in until it is pushed off the matrix. After the completion of this, a new annihilate-and-chase loop starts in order to eliminate a new block of n_b columns.

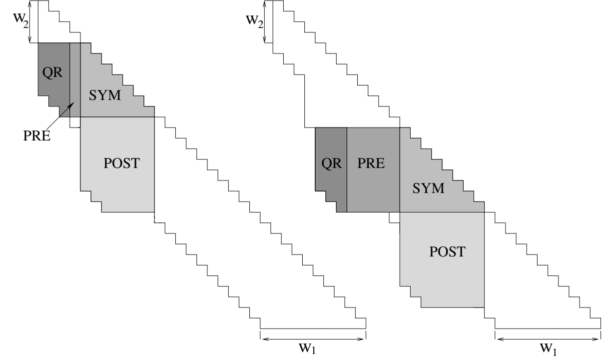

Figure 2. Left: Annihilation of the outermost sub–diagonals and updating the rest of the matrix. Right: bulge chasing part of the first fill-in block.

During the reduction to narrower band form, the updates marked as "SYM" and "POST" are candidates for being computed on the GPU, while the QR decomposition ("QR") is better suited for the CPU. The block "PRE" can be updated in either the CPU or the GPU but, to reduce memory transfers, in our code this computation is performed in the GPU as well. Now, assume that initially the entire matrix B_{w_1} resides in the GPU memory and that in the first $i - 1$ column blocks of size n_b all sub–diagonals below w_2 are annihilated. The algorithm then proceeds by repeating the "annihilate-and-chase" step as shown in Algorithm 6.

Algorithm 6 i^{th} Annihilate-and-chase step

Input: Band matrix $B_{w_1} \in \mathbb{R}^{w_1 \times n}$
Output: Matrix B_{w_2}
1: Copy "QR" from GPU to CPU
2: Compute QR of the "QR" $\rightarrow W, Y$
3: Copy W, Y to the GPU and update "PRE"
4: **for** $k = i + w_2,\ k < n,\ k = k + w_1$ **do**
5: Update "SYM" and "POST" on the GPU
6: Copy fill-in block "QR" from GPU to CPU
7: Compute the QR of the "QR" $\rightarrow W, Y$
8: Copy W, Y to the GPU, and update "PRE"
9: **end for**

Upon completion, there exists an updated copy of the band matrix B_{w_2} in the CPU. (The updated parts of the matrix were transfered at the beginning of each step.) Additional storage is required in the GPU to keep the W and Y factors as well as an $d \times n$ workspace for the fill-ins generated during the successive annihilate-and-chase steps.

The last stage of the algorithm, the reduction from banded to the tridiagonal form, is done using routine SBRDT.

5 Experimental Results

The target platform was a workstation with two Intel Xeon E5520 QuadCore CPUs running at 2.27 GHz, with 24 GB RAM, and an aggregated theoretical peak performance of 74.6 GFLOPS in double-precision (1 GFLOPS = 10^9 flops/second). The workstation was also equipped with an NVIDIA Tesla C2050 running at 1.15 GHz, with 2.8 GB RAM, and a theoretical peak performance of 515 GFLOPS in double-precision. The Intel chipset and the Tesla board are connected via a PCI-Express Gen2 interface with a peak bandwidth of 4.6 GB/second. GNU C compiler (gcc 4.1.2) and GotoBLAS2 were employed for all computations performed on the Intel cores. NVIDIA CUBLAS (version 4.0) built on top of CUDA (version 4.0) were used in our tests. Double precision was employed in all our experiments and no page-locked memory on the CPU was used.

In our results, we consider that the cost of the OOC reduction from full to banded form is $2n^3/3$ flops, and that of the full multi-stage reduction from full to tridiagonal form (either using SBR SYRDB+SBRDT or OOC banded form reduction+SBRDB+SBRDT) is $4n^3/3$ flops. This count may be considerably smaller than the actual number of flops performed, but it allows a fair comparison of the different variants that we designed for any bandwidth. We do not build the orthogonal factors in our experiments. The cost of all data transfers between main memory and GPU memory is included in the timings.

Note that the matrices with $n >$ 70,000 can not fit into the main memory of our platform. Therefore we have simulated the computation on such matrices by generating the matrix that can fit into the main memory (i.e. one panel). The calculations and copying to the GPU are done as we calculate with the large matrix (but always operating on the same panel). In that way we have made the correct number of data transfers and the computations although the final result is not correct.

Our first experiments analyze the performance of the OOC reduction of a full symmetric matrix to banded form. These experiments evaluate the algorithms described in Subsection 3.1 and 3.2. Specifically, we consider the following two variants for the reduction from full symmetric matrix to banded form:

Variant 1 employs variants QR-1 and Update-1.

Variant 2 employs variants QR-2 and Update-2.

Variant 2 requires less storage space on the GPU than Variant 1 at the cost of increasing the number of memory transfers from/to the GPU.

In all our tests the parts of the code that are executed on the CPU employ multi-threaded BLAS/LAPACK libraries utilizing all 8 available cores.

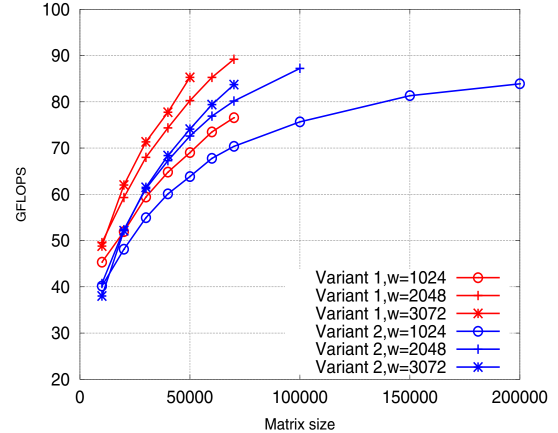

Figure 3. Performance of the OOC Variant 1 (red) and Variant 2 (blue).

Preliminary experiments showed that the optimal block size for the hybrid QR algorithm was $\tilde{b} = 128$. The performance of the reduction from full to banded form mostly depends on the band size w, as shown in Figure 3. In particular, increasing the band size decreases the number of computations and increments the performance. The best performance attained with Variant 1 is 89 GFLOPS ($2,048$ sec.) for n=70,000 and w=2,048. The performance is bounded by the total memory capacity of GPU, as for the matrix of dimension n=70,000 and the maximum bandwidth w=2,272 the largest block size for Variant 1 is $\hat{b}$=3,035. By decreasing the bandwidth, the block size can be made larger, but the overall performance decreases. The problem with the scalability occurs when the matrix size n grows as two matrices of size $n \times w$ have to be stored in the GPU memory and, thus, less storage is available for the workspace of size $\hat{b} \times \hat{b}$. This is illustrated in Figure 3 for w=3,072. In that case, the largest problem dimension that could be evaluated for w=3,072 was 30,000. The performance of Variant 2 (Figure 3 blue) is only slightly lower than that of Variant 1. The reason for that is that we perform more memory transfers from/to the GPU. However, at the same time, we can operate with larger blocks $\hat{b}$. For the matrix of size n=200,000 and the bandwidth w=1,024, Variant 2 achieves 83,9 GFLOPS (58,885 sec). Compared with Variant 1, for the same bandwidth w, we can solve much larger problems. The maximum matrix size that we have tested was n=250,000 with the band size w=1,024. The execution rate for that specific problem is 91 GFLOPS which outperforms Variant 1 for the same bandwidth but smaller problem size (n=70,000).

The impact of the memory transfer to/from the GPU in the total execution time decreases as the band size increases. The impact of the copy is starting from $\approx$46% for w=512 up to 9.5% for w=4256 and does not depend on the problem

size n. For the same problem size, decreasing the band size results in the considerable increase of the transfers while the computational time remains almost the same.

The second experiment compares our OOC multi-stage algorithm for the reduction of a full symmetric matrix to tridiagonal form with the (in-core) SBR two-stage approach (SYRDB + SBRDT) as presented in [1]. Our approach performs the first and second stage (SYRDB and SBRDB) on the GPU, while the rest of the computation is done on the CPU. The flop counts for both alternatives is estimated as $4n^3/3$. Our algorithm for reduction to tridiagonal form consists of three phases: The first phase performs the reduction from full symmetric matrix to banded form employing the OOC algorithm (Section 3); the second phase is the GPU implementation of the SBR routine SBRDB (Section 4) for reduction from band to narrower band; and the third phase invokes the SBR routine SBRDT for reduction of the banded matrix to tridiagonal form. After the first phase, band matrices are stored in compact format using a $w \times n$ array. In Table 1 we compare the performance of the two OOC multi-stage algorithms, that use Variant 1 and Variant 2 of the OOC reductions from full to banded form, with that of the SBR two-stage GPU implementation.

Table 1. Performance in seconds of two OOC versions and SBR two-stage algorithms on the GPU.

Size	SBR 2-stage		OOC version 1		OOC version 2	
	total	copy	total	copy	total	copy
5000	7.25	0.28	8.35	0.32	8.42	1.00
10000	39.60	0.83	58.78	3.02	40.04	5.15
16000	128.80	2.23	108.52	12.95	122.94	18.65
18000	174.20	2.62	150.38	11.16	156.55	22.46

The largest problem size that can be solved in double-precision arithmetic with the in-core GPU implementation of the SBR two-stage approach is n=18,000. Our approach, on the other hand, can be used for problems of size up to n=300,000 (as long as we can maintain $w \geq 512$, the bottleneck is the amount of the CPU main memory). In our algorithm for the reduction to tridiagonal form, the most time-consuming part is the first phase (reduction from full to band form) as this routine performs most of the computations. The best performance for this stage, as presented in the Figure 3 is achieved when the band size is large. The band size for the last phase (reduction from band to tridiagonal) affects the performance as well as should be kept small. Therefore, the best performance is achieved when the band size for the first phase is 2048, and for the third is 64. The second phase is an in-core algorithm that operates on the GPU, and requires only a workspace of dimension $(2w + 1) \times n$. The design of an efficient OOC reduction algorithm from band to narrower band, that solves the current constraints on the bandwidth, is among future research goals.

6 Conclusions

In this paper, we have proposed new OOC algorithms for the reduction of a dense symmetric matrix to tridiagonal form when the problem data does not fit into the GPU memory. The crucial point is the transformation from a full dense matrix to banded form, when most of the computations are performed on the full matrix. The second phase, the reduction from band matrix to narrower band or directly to tridiagonal form, can be executed using in-core algorithms, as the bandwidth can be chosen so that the band matrix, saved in compact form, fits into the GPU memory. The improvements in our algorithms come from a careful reformulation of the two-sided update as an OOC algorithm. Specifically, the input matrix is divided into blocks that fit into the GPU memory and the algorithm is redesigned to amortize the transfers of data with the computation, calculating with large blocks so that the ratio between the computation and transfer remains high.

Our experiments demonstrate that the performance of the OOC algorithm greatly depends on the bandwidth, increasing with it. The experiments also show that, for the reduction to band form, the use of OOC techniques provides higher scalability and higher performance, up to $2-3\times$ for large problems, than their in-core counterparts.

References

[1] P. Bientinesi, F. D. Igual, D. Kressner, M. Petschow, and E. S. Quintana-Ortí. Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures. *Conc. & Comp.: Practice and Experience*, 23(7):694–707, 2011.

[2] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Trans. Math. Soft.*, 26(4):602–616, 2000.

[3] C. H. Bischof, B. Lang, and X. Sun. A framework for symmetric band reduction. *ACM Trans. Math. Soft.*, 26(4):581–601, 2000.

[4] S. Dhillon, N. Parlett, and Vomel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Soft.*, 32(4):533–560, 2006.

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

[6] P. Luszczek, H. Ltaief, and J. Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *IPDPS*, pages 944–955. IEEE, 2011.

[7] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.