Out-of-Core Solution of Eigenproblems for Macromolecular Simulations on GPUs

José I. Aliaga[1], Davor Davidović[2], and Enrique S. Quintana-Ortí[1]

[1] Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain. {aliaga,quintana}@icc.uji.es
[2] Institut Ruder Bošković, Centar za Informatiku i Računarstvo - CIR, 10000–Zagreb, Croatia. ddavid@irb.hr

Abstract. We consider the solution of large-scale eigenvalue problems that appear in the motion simulation of complex macromolecules on desktop platforms. To tackle the dimension of the matrices that are involved in these problems, we formulate out-of-core (OOC) variants of the two selected eigensolvers, that basically decouple the performance of the solver from the storage capacity. Furthermore, we contend with the high computational complexity of the solvers by off-loading the arithmetically-intensive parts of the algorithms to a hardware graphics accelerator.

Keywords: Macromolecular motion simulation, eigenvalue problems, out-of-core computing, multicore processors, GPUs

1 Introduction

Coarse-grained models (CGM) combined with normal mode analysis (NMA) has been applied in recent years to simulate biological activity at molecular level for extended time scales [2], [3], [15]. Concretely, iMod [9] is a tool chest that exploits the advantage of NMA formulations in internal coordinates (ICs) while extending them to cover multi-scale modeling. Despite the reduction in the degrees of freedom offered by ICs, the diagonalization step remains the major computational bottleneck of this approach, specially for large molecules. In particular, the eigenproblem that has to be solved in this step of CGM-NMA is given by

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

where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times n}$ correspond, respectively, to the Hessian and kinetic matrices that capture the dynamics of the macromolecular complex, $\Lambda \in \mathbb{R}^{s\times s}$ is a diagonal matrix with the s sought-after eigenvalues, and $X \in \mathbb{R}^{n\times s}$ contains the corresponding unknown eigenvectors [8]. Furthermore, when dealing with large macromolecules, A, B are dense symmetric positive definite matrices, $n \geq 10{,}000$, and typically only the $s \approx 100$ smallest eigenpairs are required.

In this paper we address the efficient solution of large-scale generalized symmetric definite eigenproblems arising in the simulation of collective motions of macromolecular complexes using multicore desktop platforms equipped with

graphics processing units (GPUs). While there exist other related work [1], [11], [10], our paper makes the following original contributions:

- The eigenproblems associated with this particular application involve dense matrices that are, in general, too large to fit into the memory of the GPU and, in some cases, even the main memory of the server. To address this, we consider two specialized algorithms that, by applying out-of-core (OOC) techniques [16], amortize the cost of data transfers with a large number of floating-point arithmetic operations (flops). Besides, to deliver high performance, both "OOC-GPU" algorithms off-load the bulk of their computations to the attached hardware graphics accelerator.
- One of our algorithms is the first OOC-GPU implementation that employs spectral divide-and-conquer (SD&C) based on the polar decomposition proposed recently [13]. We enhance this algorithm with *ad-hoc* splitting strategies, that aim at reducing the number of SD&C iterations, and are cheap to compute for the biological target application.
- As an alternative algorithm, we revisit an implementation of the two-stage reduction to tridiagonal form [6], where the first stage is also an OOC-GPU code while the subsequent stage operates on a much reduced compact matrix that fits in-core.
- We perform a comparison of these two approaches using several datasets representative of large-scale macromolecular complexes [10].

Overall the major contribution of this paper lies in that it provides a demonstration that complex macromolecular motion simulations can be tackled on desktop servers equipped with GPUs even when the problem data is too large to fit into the memory of the hardware accelerator and, possibly, even the main memory.

The rest of the paper is structured as follows. In Section 2 we briefly describe the solution of generalized eigenproblems. In Section 3 we review in detail the SD&C method [13], and revisit the two-stage eigensolver, describing our hybrid CPU-GPU approach. Implementations of these eigensolvers are evaluated next, in Section 4, using a collection of cases from biological sources. Finally, we close the paper in Section 5 with a few concluding remarks.

2 Solution of Symmetric Definite Eigenproblems

All the eigensolvers considered in this work initially compute the Cholesky factorization $B = U^T U$, where $U \in \mathbb{R}^{n \times n}$ is upper triangular [8], to then tackle the *standard* symmetric eigenproblem

$$CY = Y\Lambda \quad \equiv \quad (U^{-T} A U^{-1})(UX) = (UX)\Lambda, \tag{2}$$

where $C \in \mathbb{R}^{n \times n}$ is symmetric and $Y \in \mathbb{R}^{n \times s}$. Thus, the standard eigenproblem (2) shares its eigenvalues with those of (1), while the original eigenvectors can be recovered from $X := U^{-1} Y$. The initial Cholesky factorization, the construction of $C := U^{-T} A U^{-1}$ in (2), and the solve for X are known to deliver

high performance on a large variety of HPC architectures, including multicore processors and GPUs, and their functionality is covered by numerical libraries (e.g., LAPACK, `libflame`, ScaLAPACK, PLAPACK, etc.) including some OOC extensions (SOLAR, POOCLAPACK). Therefore, we will not consider these operations further but, instead, focus on the more challenging solution of the standard eigenproblem (2) on a hybrid CPU-GPU platform when the data matrices are too large to fit into the GPU memory (and, possibly, the main memory).

Among the different solvers for the symmetric eigenproblem, we discard those based on the one-stage reduction to tridiagonal form as well as the Krylov methods [8]. From an OOC viewpoint, the major drawback of these two classes of methods is that they cast a significant part of their computations in terms of the matrix-vector product (MVP). For a matrix of size $n \times n$, this kernel roughly performs $2n^2$ flops on n^2 numbers (i.e., a rate of computation to data of $O(1)$), so that an implementation that operates with OOC data (e.g., a GPU MVP routine where the matrix is on the main memory, or a multicore MVP code with data on disk) is intrinsically limited by data movement and will attain very low performance.

Instead, we will investigate a recent SD&C approach [13], with a much higher computational cost than the one-stage/Krylov-based methods, but which consists mainly of matrix-matrix operations that naturally render it as an appealing candidate for OOC-GPU strategies/platforms. As an alternative, we will also consider a classical eigensolver based on a two-stage reduction to tridiagonal form, which first transforms the matrix C from dense to band form, to then refine this intermediate matrix to tridiagonal form. We have previously described [6] an OOC-GPU practical implementation of this two-stage eigensolver and demonstrated how, by carefully orchestrating the PCI data transfers between host and device, in-core performance is maintained or even increased for the OOC solution of general large-scale eigenproblems on hybrid CPU-GPU platforms.

3 OOC Eigensolvers for GPUs

In this section we review the mathematical methods that underlie our GPU eigensolvers, discuss how to refine the SD&C algorithm to reduce its computational cost for the solution of the eigenproblems arising in macromolecular motion simulation, and offer some practical details about the OOC-GPU implementations using one key numerical kernel that appears in the algorithms.

3.1 The SD&C algorithm

Numerical method. For a symmetric matrix $\hat{A} \in \mathbb{R}^{n \times n}$, the following SD&C algorithm [13] starts by computing its polar factor using the QR-based dynamically weighted Halley (QDWH) iterative scheme [12]:

$$\begin{bmatrix} \sqrt{c_j} X_j \\ I_n \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R \quad \text{(QR factorization)}, \tag{3}$$

$$X_{j+1} := \frac{b_j}{c_j} X_j + \frac{1}{\sqrt{c_j}} \left(a_j - \frac{b_j}{c_j} \right) Q_1 Q_2^T, \quad j \geq 0, \tag{4}$$

where $X_0 := \hat{A}/\alpha$ and I_n denotes the identity matrix. In practice, the scalars α, a_j, b_j, c_j require estimates of the smallest singular value and matrix 2-norm of $\hat{A}$, which are cheap to compute and, upon convergence, the sequence X_j yields the sought-after polar factor U_p.

Assume QDWH has been applied to $\hat{A} := C - \sigma I_n$, with σ a user-defined splitting point for the eigenspectrum of the symmetric matrix C in (2). The subspace iteration [8] is next employed to compute an orthogonal matrix $[V_1,\ V_2]$, where $V_1 \in \mathbb{R}^{n \times k}$, such that $(U_p + I_n)/2 = V_1 V_1^T$; therefore,

$$\begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} C \, [V_1,\ V_2] = \begin{bmatrix} C_1 & E^T \\ E & C_2 \end{bmatrix}, \tag{5}$$

where C_1 and C_2 contain, respectively, the eigenvalues of C to the left and right of σ, and $\|E\|_F \approx u$, with u the machine unit roundoff [13].

Choosing the splitting point. The previous method is designed as a recursive SD&C algorithm: after dividing the spectrum of C into those of C_1 and C_2, the method is applied again to these two subproblems, using appropriate shifts σ_1 and σ_2 to further divide the spectrum. Note that our goal is to compute only a few eigenpairs of the problem, specifically the smallest s. We therefore designed three SD&C strategies, with the common purpose of selecting the appropriate value of σ, that separates the eigenspectrum of C into two subsets C_1 and C_2, with the dimension of the former being equal to (or only slightly larger than) s. Specifically, we designed and evaluated three different SD&C strategies:

SD&C-A. $\sigma = \text{trace}\{A\}/n$, where $\text{trace}\{\cdot\}$ denotes the trace of its argument.
SD&C-B. $\sigma = 4\ \text{trace}\{A\}/n$.
SD&C-C. In this case, given a macromolecule, we use IMOD to generate the Hessian and kinetic matrix for a problem of much smaller dimension, say $m \approx 1,024$, and choose σ as the $(100 \cdot m/n)$-th largest eigenvalue of this problem.

Note also that just before the application of the subspace extraction, the value $k = \|U_p + I_n\|_F^2/2$ indicates the number of eigenvalues in C_1. Therefore, in case $k < s$, the QDWH iterate has to be recomputed, with a larger value for σ. After the first successful split, the eigenspectrum of C_1 is completely computed using a direct in-core eigensolver based on the reduction to tridiagonal form (and with a negligible cost compared with that of the initial stage).

3.2 Two-stage reduction to tridiagonal form

The eigensolver based on the two-stage reduction to tridiagonal form performs the major part of the computations in terms of efficient Level-3 BLAS operations, in exchange for a nonnegligible increment in the computational cost when

compared with the direct (one-stage) reduction. The two-stage algorithm first computes the decomposition $Q_1^T C Q_1 = \hat{C}$, where $\hat{C} \in \mathbb{R}^{n \times n}$ is a matrix of bandwidth w, and $Q_1 \in \mathbb{R}^{n \times n}$ is orthogonal. In the subsequent stage, $\hat{C}$ is further reduced to a tridiagonal matrix $T \in \mathbb{R}^{n \times n}$ as $Q_2^T \hat{C} Q_2 = T$ with $Q_2 \in \mathbb{R}^{n \times n}$ orthogonal. Finally, the eigenvalues of T (which are also those of the C) and the associated eigenvectors, in $Z \in \mathbb{R}^{n \times s}$, are computed using, e.g., the MR^3 solver [7], [4]; and the eigenvectors are recovered from $Y := Q_1 Q_2 Z$.

Our implementation of this approach is based on the SBR (Successive Band Reduction) toolbox [5] for the two-stage reduction to tridiagonal form, and employs the LAPACK routine for the MR^3 method, which in general only adds a negligible cost. We have previously described an OOC-GPU implementation of the reduction to band form [6] that carefully orchestrates computation and communication to deliver performance equal or superior to that of an in-core GPU routine. Furthermore, provided w is carefully chosen, the second stage and the solution of the tridiagonal eigenproblem can proceed with data in core [6].

The OOC-GPU code for the first stage consists basically of three major kernels: QR factorization, one-sided update (for the application of orthogonal transforms from the left), and two-sided update (application from both left and right). These kernels are thus conceptually analogous to some of those appearing in the SD&C algorithm.

3.3 OOC kernels

We next illustrate the OOC-GPU implementations using (a specialized case of) the QR factorization as a workhorse. Our OOC-GPU algorithm for this operation encodes a left-looking, slab-oriented factorization [16] that transfers data by column blocks (slabs) of width s. Note that, while there exist linear algebra libraries to obtain the QR factorization on GPUs [11], these lack of the specialized kernels that are necessary for our particular operation.

In particular, let us denote the $2n \times n$ matrix that has to be factorized in (3) as D, and consider a partitioning of this matrix into blocks of dimension $s \times s$ each, where $D[i, j]$ denotes the (i, j)-th block and, for simplicity, we assume that n is an integer multiple of s. Here, the parameter s is chosen so that a slab of size $(n + s) \times s$ can fit into the GPU memory. Routine QR_OOC in Listing 1.1 and Figure 1 (left) describe how to leverage the upper triangular structure of the bottom $n \times n$ half of D during the computation of the QR factorization of this matrix using our OOC-GPU algorithm. For each iteration of the outer loop, the algorithm first updates (part of) the k-th slab of D w.r.t. the transforms that were calculated earlier (as corresponds to a left-looking variant). These transforms are divided into slabs of width s and applied, in the inner loop, to the corresponding fraction of $D[:, k]$ from the left, invoking routine UPDATE_GPU for that purpose. After the update, the algorithm proceeds to factorize the current slab, using routine QR_HYBRID. Note how, at each outer iteration of this loop, one slab of D is transferred from main memory to the GPU, modified there, and the results are sent back to the main memory.

The code for the building kernels UPDATE_GPU and QR_HYBRID is also given in Listing 1.1, and both procedures are illustrated in Figure 1 (right). In these routines matrices E and F are partitioned into blocks of size $b \times b$, so that $E[i,j]$, $F[i,j]$ stand for the (i,j)-th blocks of the corresponding matrix. For simplicity, we assume now that s is an integer multiple of b. The first routine operates with F (a slab of D of width s) stored in-core (i.e., in the GPU memory), and streams blocks of E, of width b, from the main memory to the GPU, in order to update F with the orthogonal transforms contained in them. The second routine computes a QR factorization of F (stored in-core), using a conventional blocked left-looking procedure with block size b, so that the block factorizations and orthogonal transforms are computed in the CPU, while the updates of the trailing submatrices are performed in the GPU.

```
1     FUNCTION D = QR_OOC( n, s, b, D );
2     r = n/s;
3     FOR k = 1:r
4       COPY D[ 1:r+k, k ] to GPU
5       FOR j = 1:k-1
6         D[ j:r+j, k ] = UPDATE_OOC( n, s, b, D[ j:r+j, j ], D[ j:r+j, k ] );
7       END
8       D[ k:r+k, k ] = QR_HYBRID( n, s, b, D[ k:r+k, k ] );
9       COPY D[ k:r+k, k ] to main memory
10    END
11  // ---------------------------------------------------------------------------
12    FUNCTION F = UPDATE_OOC( n, s, b, E, F );
13    r = n/b; t = s/b;
14    FOR k = 1:t
15      COPY E[ k:r+k, k ], containing Qk, to main memory
16      F[ k:r+k, : ] = Qk' *  F[ k:r+k, : ]; // Update in GPU
17    END
18  // ---------------------------------------------------------------------------
19    FUNCTION E = QR_HYBRID( n, s, b, E );
20    r = n/b; t = s/b;
21    FOR k = 1:t
22      COPY E[ k:r+k, k ] to main memory
23      E[ k:r+k, k ] = Rk/Qk = QR( E[ k:r+k, k ] );  // Factorize in CPU
24      COPY E[ k:r+k, k ], containing Qk, to GPU
25      E[ k:r+k, k+1:r ] = Qk' *  E[ k:r+k, k+1:r ]; // Update in GPU
26    END
```

Listing 1.1. OOC-GPU left-looking slab-based algorithm for the QR factorization QR_OOC and the building kernels UPDATE_OOC and QR_HYBRID.

Optimization of QR_OOC**.** In our QR_OOC algorithm, only the orthogonal matrix of the resulting QR factorization of D is built/stored while the upper triangular factor is not referenced/kept. Our QR algorithm is a left-looking algorithm that applies all previous transformations to the current slab —in contrast with the traditional right-looking approach that immediately propagates the transforms to the right of the current slab— since left-looking OOC variants in general incur in a smaller number of transfers [16].

Furthermore, we leverage the special structure of $D[:,k]$ (Figure 1, left) to further reduce the number of transfers. Concretely, at each step of the inner loop of routine QR_OOC, $D[j : r+j, k]$, of size $(n+s) \times s$, is stored in the GPU memory. Now, during the next iteration of loop j, $D[j+1 : r+j+1, k]$ will

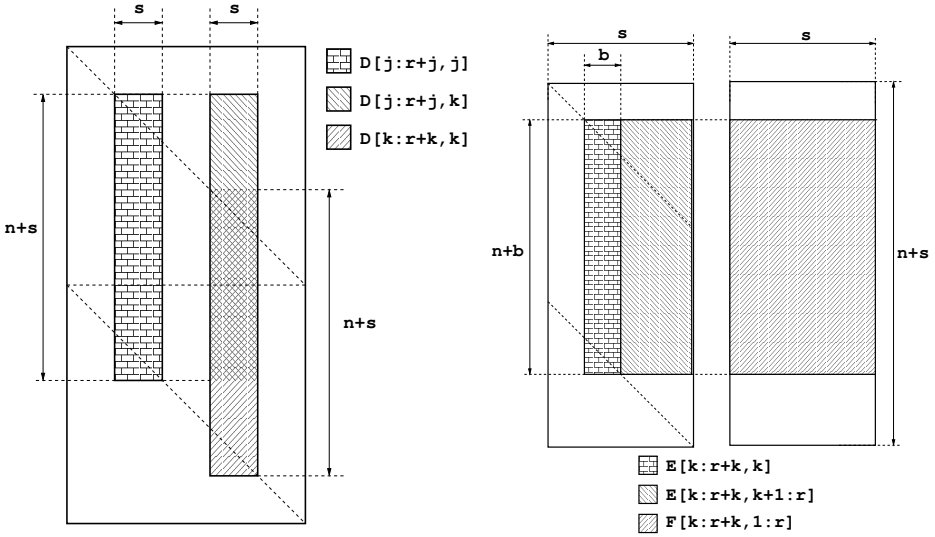

Fig. 1. Accompanying illustrations for the OOC-GPU QR factorization QR_OOC (left), and the building kernels UPDATE_OOC and QR_HYBRID (right).

be required; see Figure 1 (left). Thus, the difference between these two slabs corresponds to $D[j,k]$, which does not need to be sent back to main memory as it belongs to the upper triangular factor; and $D[r+j+1,k]$, which is stored in main memory and will have to be transferred to the GPU. Therefore, at the end of iteration j, $D[j,k]$ is removed from the GPU memory, as it is not required any longer; and the new block $D[r+j+1,k]$ is transferred from main memory to the GPU. Applying this approach, at each update step only one block of size $s \times s$ needs to be sent to the GPU, instead of the whole slab of size $(n+s) \times s$.

Optimization of QR_HYBRID**.** This routine computes the QR factorization of $D[k : r+k,k]$ with the collaboration of both CPU and GPU. This slab is divided into blocks of width b; see Figure 1 (right). At each iteration of routine QR_HYBRID in Listing 1.1, the orthogonal factor for $E[k : r+k,k]$ is computed on the CPU, and transferred to the GPU; and the submatrix to the right is next updated on the GPU. Thus, the QR factorization computed at each iteration only involves $E[k : r+k,k]$. Following this approach, the special structure of E can be efficiently exploited with little overhead, that depends only on the relation between b and s. (In practice, $b \leq 128$ while s is much larger.)

4 Experimental Results

All the experiments were performed on a server with two Intel Xeon E5520 quad-core processors (total of 8 cores @ 2.27 GHz), 48 Gbytes of RAM, and a Tesla C2050 GPU (2.6 Gbytes of memory, ECC on), using IEEE double-precision

arithmetic. The results include the cost of transferring the input data and results between main memory and GPU. The codes were linked to NVIDIA CUBLAS (v5.0) and the BLAS implementation in GotoBLAS2 (v1.13).

For simplicity, we will only consider GPU routines that operate with data residing in the main memory. For matrix decompositions such as the QR factorization and other similar Level-3 BLAS-based kernels, disk latency can be mostly hidden by overlapping it with computation, even in platforms equipped with GPU accelerators [14]. Therefore, we expect these results to carry over to the case where data is stored on disk.

We employed 6 datasets in the experimentation: UTUBSEAM40, UTUBSEAM10, RIBOTIPRE, 1CWP, 1QGT and UTUBSEAM20, leading to eigenproblems of dimension n =24,943, 29,622, 30,065, 30,504, 30,785 and 31,178, respectively, that in all cases do not fit into the GPU memory.

Our first experiment analyzes the scalability of the OOC-GPU algorithms, measured as the ability of these methods to deliver a constant GFLOPS (billions of flops/second) rate as the problem dimension grows to exceed the capacity of the GPU memory. For this purpose, we employed IMOD to generate matrices of varying dimensions for the UTUBSEAM$\{10, 20, 40\}$ benchmarks. Figure 2 shows that the OOC-GPU two-stage and SD&C algorithms are scalable in this sense. At this point, be aware that the much higher GFLOPS ratio of the approach based on the SD&C method do not necessarily imply superior performance since, as we will show in the next experiment, this method also requires a much higher cost than the two-stage alternative.

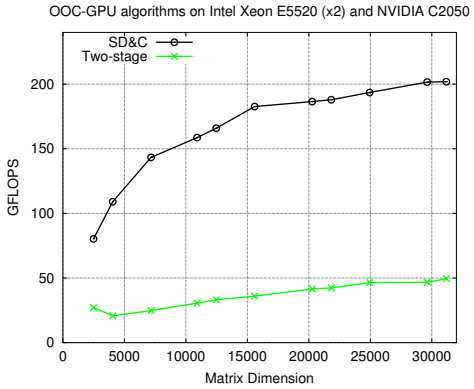

Fig. 2. GFLOPS rate of the OOC-GPU eigensolvers applied to reduced versions of the UTUBSEAM$\{10, 20, 40\}$ test cases.

Table 1 compares the total execution time of the OOC-GPU two-stage and SD&C algorithms, using the three techniques to choose the splitting parameter σ described in subsection 3.1 for the latter. As could be expected, the execution time of the SD&C algorithm strongly varies depending on the properties of the

Case	Two-stage	SD&C-A			SD&C-B			SD&C-C		
	Time	Time	#iter	split	Time	#iter	split	Time	#iter	split
UTUBSEAM40	1534.3	3087.9	7	8678	2402.8	7	712	2428.2	7	1024
UTUBSEAM10	2536.3	4652.3	7	9006	3871.2	7	936	3877.8	7	1034
RIBOTIPRE	2426.4	5868.2	9	11779	3420.6	6	284	4736.9	7	11448
1CWP	2523.1	5949.8	9	7005	4276.4	7	1412	8264.1	12	16721
1QGT	2622.9	6503.7	10	7362	5525.9	9	1562	9650.2	12	20952
UTUBSEAM20	2780.9	7263.4	10	9288	4521.4	7	815	5937.8	9	2511

Table 1. Comparison of eigensolvers. Time is reported in seconds in all cases. For the SD&C variants, "#iter" is the number of QDWH iterations and "split" is subproblem size after the first divided-and-conquer step.

spectrum and the splitting point, and different strategies to select σ greatly affect the convergence speed of the QDWH. For our particular test cases, strategy SD&C-B offers the best results as it combines fast convergence with the decoupling of a subproblem C_1 of reduced size, which renders the cost of the subspace iteration low. However, in all cases, the two-stage approach is clearly superior to the SD&C method.

5 Concluding Remarks

We have presented and evaluated two hybrid CPU-GPU algorithms for the solution of generalized symmetric eigenproblems arising in macromolecular motion simulation, based on the two-stage reduction to tridiagonal form and a new spectral divide-and-conquer approach for the polar decomposition. In both cases, by carefully amortizing the cost of the PCI data transfers with a large number of floating-point arithmetic operations, the implementations attain high performance and, more importantly, offer perfect scalability so that the dimension of the macromolecular problems that can be tackled is not constrained by the capacity of the GPU memory.

Experiments on an desktop platform with two Intel Xeon multicore processors and an NVIDIA "Fermi" GPU, representative of current server technology, illustrate the potential of these methods to address the simulation of biological activity. These results also show the superior performance of the OOC-GPU two-stage approach over the SD&C implementations, despite the former necessarily computes the full eigenspectrum of the problem while the latter can be used, in principle, to obtain only the sought-after part of the spectrum.

As part of future work, we plan to extend these algorithms to operate with data on disk, so that much larger problems can be addressed on desktop platforms with a reduced main memory.

Acknowledgments

D. Davidović's visit to UJI was supported by the COST Action IC0805. The researchers from UJI were supported CICYT TIN2008–06570-C04-01 and FEDER, the EU FP7 318793 "EXA2GREEN", and P1-1B2011-18 of the Fundació Caixa-Castelló/Bancaixa and UJI. We also thank the Structural Bioinformatics Group, from CSIC, for the datasets.

References

1. J. Aliaga, P. Bientinesi, D. Davidović, E. D. Napoli, F. Igual, and E. S. Quintana-Ortí. Solving dense generalized eigenproblems on multi-threaded architectures. *Applied Mathematics and Computation*, 218(22):11279–11289, 2012.
2. G. S. Ayton and G. A. Voth. Systematic multiscale simulation of membrane protein systems. *Curr. Opin. Struct. Biology*, 19(2):138–44, 2009.
3. I. Bahar, T. R. Lezon, A. Bakan, and I. H. Shrivastava. Normal mode analysis of biomolecular structures: functional mechanisms of membrane proteins. *Chem. Rev.*, 110(3):1463–97, 2010.
4. P. Bientinesi, I. S. Dhillon, and R. van de Geijn. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM J. Sci. Comput.*, 27(1):43–66, 2005.
5. 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.
6. D. Davidović and E. S. Quintana-Ortí. Applying OOC techniques in the reduction to condensed form for very large symmetric eigenproblems on GPUs. In *20th Euro. Conf. PDP 2012*, pages 442–449, 2012.
7. I. S. Dhillon and B. N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and its Applications*, 387:1 – 28, 2004.
8. G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.
9. J. R. Lopez-Blanco, J. I. Garzon, and P. Chacon. iMOD: multipurpose normal mode analysis in internal coordinates. *Bioinformatics*, 27(20):2843–50, 2011.
10. J. R. López-Blanco, R. Reyes, J. I. Aliaga, R. M. Badia, P. Chacón, and E. S. Quintana. Exploring large macromolecular functional motions on clusters of multicore processors. *J. Comp. Phys.*, 246:275–288, 2013.
11. MAGMA project home page. `http://icl.cs.utk.edu/magma/`.
12. Y. Nakatsukasa, Z. Bai, and F. Gygi. Optimizing Halley's iteration for computing the matrix polar decomposition. *SIAM J. Matrix Anal. Appl.*, 31:2700–2720, 2010.
13. Y. Nakatsukasa and N. J. Higham. Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD. Technical Report 2012.52, Manchester Inst. Math. Sci., The University of Manchester, 2012.
14. G. Quintana-Ortí, F. D. Igual, M. Marqués, E. S. Quintana-Ortí, and R. A. V. de Geijn. A run-time system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures. *ACM Trans. Math. Softw.*, 38(4):25:1–25:25.
15. L. Skjaerven, S. M. Hollup, and N. Reuter. Normal mode analysis for proteins. *J. Mol. Struct. (Theochem)*, 898(1-3):42–48, 2009.
16. S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1999.