Computer Physics Communications 185 (2014) 1570–1581

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

The density matrix renormalization group algorithm on kilo-processor architectures: Implementation and trade-offs

Csaba Nemes [a,*], Gergely Barcza [c], Zoltán Nagy [a,b], Örs Legeza [c], Péter Szolgay [a,b]

[a] Faculty of Information Technology, Péter Pázmány Catholic University, Budapest, Hungary

[b] Cellular Sensory and Wave Computing Laboratory, Computer and Research Automation Institute, Hungarian Academy of Sciences, Budapest, Hungary

[c] Strongly Correlated Systems "Lendület" Research Group, Department of Theoretical Solid State Physics, Wigner Research Centre for Physics, Hungarian Academy of Sciences, Budapest, Hungary

ARTICLE INFO

Article history:
Received 21 September 2013
Received in revised form
10 February 2014
Accepted 18 February 2014
Available online 26 February 2014

Keywords:
Strongly correlated systems
DMRG
GPU acceleration
FPGA acceleration

ABSTRACT

In the numerical analysis of strongly correlated quantum lattice models one of the leading algorithms developed to balance the size of the effective Hilbert space and the accuracy of the simulation is the density matrix renormalization group (DMRG) algorithm, in which the run-time is dominated by the iterative diagonalization of the Hamilton operator. As the most time-dominant step of the diagonalization can be expressed as a list of dense matrix operations, the DMRG is an appealing candidate to fully utilize the computing power residing in novel kilo-processor architectures.

In the paper a smart hybrid CPU–GPU implementation is presented, which exploits the power of both CPU and GPU and tolerates problems exceeding the GPU memory size. Furthermore, a new CUDA kernel has been designed for asymmetric matrix–vector multiplication to accelerate the rest of the diagonalization. Besides the evaluation of the GPU implementation, the practical limits of an FPGA implementation are also discussed.

1. Introduction and related works

DMRG is a variational numerical approach developed to treat low-dimensional interacting many-body quantum systems efficiently [1–3]. In fact, it has become an exceptionally successful method to study the low energy physics of strongly correlated quantum systems which exhibit chain-like entanglement structure [4]. For example, it can be applied to simulate properties of anisotropic materials, such as polymers [5], or to describe accurately the electronic structure of open *d* shell molecules [6], which is beyond the capability of standard quantum chemical approaches. Additionally, the interacting system of atoms trapped in an optical lattice, proposed as physical implementation of quantum computer, is also tractable via DMRG [7].

Over the past ten years, the DMRG method has been also reformulated in terms of Matrix Product States (MPS) [8] leading to various extensions [9–11] and has been shown to be a special case of a more general set of methods, the so-called Tensor Network States (TNS) [12–14]. A common feature of all these algorithms is that computational tasks can be massively parallelized.

The original DMRG algorithm [1] was introduced in 1992 by Steven R. White and was formulated as a single threaded algorithm. In the past various works have been carried out to accelerate the DMRG algorithm on shared [15,16] and distributed memory [17–20] architectures, however, none of them took advantage of recent kilo-processor architectures: graphical processing unit (GPU) and field-programmable gate array (FPGA).

One of the first parallelizations was [15] converting the projection operation to matrix–matrix multiplications and accelerating them via OpenMP interface. In [19] a similar approach was presented for distributed memory environment (up to 1024 cores) optimizing the communication between the cores, while in [20] the acceleration of the computation of correlation function had been investigated. Recently, [16] presented an acceleration on shared memory architectures exploiting SU(2) symmetries, while [21] proposed a novel direction for parallelization via a modification of the original serial DMRG algorithm.

Graphical processing unit has been successfully employed in neighboring research areas to accelerate matrix operations. In [22] GPU is used to accelerate tensor contractions in Plaquette Renormalization States (PRS), which can be regarded as an alternative

* Corresponding author.
E-mail addresses: csnemes@gmail.com, nemes.csaba@itk.ppke.hu (C. Nemes).

http://dx.doi.org/10.1016/j.cpc.2014.02.021
0010-4655/

technique to Tensor Product States (TPS) or the DMRG algorithm. In [23] the second-order spectral projection (SP2) algorithm has been accelerated, which is an alternative technique to calculate the density matrix via a recursive series of generalized matrix–matrix multiplications.

In this paper we present the first attempt (to our best knowledge) to investigate how the DMRG method can utilize the enormous computing capabilities of novel kilo-processor architectures (GPU, FPGA). In case of GPU a smart hybrid CPU–GPU acceleration is presented, which tolerates problems exceeding the GPU memory size, consequently, supporting wide range of problems and GPU configurations. Contrary to the previous acceleration attempts not only the projection operation is accelerated, but further parts of the diagonalization are also computed on the GPU. In case of FPGA the performance limits of a possible implementation are estimated and discussed.

The rest of the paper is organized as follows. Section 2 describes the models which are used as test cases to demonstrate the operation of the algorithm. Symmetries which can be exploited to decrease the computational requirements of the algorithm and the algorithm itself are presented in Sections 3 and 4, respectively. Acceleration on GPU is presented in three Sections 5–7, while limits of an FPGA implementation are described in Section 8. Finally, implementation results and conclusions are given in Sections 9 and 10, respectively.

2. Investigated models

In order to illustrate the underlying features of the algorithm it is applied to the so-called spin-$\frac{1}{2}$ Heisenberg model and the spin-$\frac{1}{2}$ Hubbard model. The selected models describe how to compute the Hamiltonian of the system of interest, while the main task is to find some of the low-lying eigenvalues and eigenvectors of the Hamiltonian by a diagonalization algorithm. In practice instead of solving the problem for the complete Hilbert space directly, various physical phenomena can be exploited to reduce the complexity of the problem.

2.1. Heisenberg model

The Heisenberg model describes the physics of magnetic systems and provides theoretical description of various experimental measurements. In the model a magnetic system is simulated on a lattice of interacting *spins*. A microscopic magnetic moment (spin) is localized at each lattice site j and described by a quantized, two-valued variable, $\sigma_j \in \{\uparrow, \downarrow\}$, related to the two possible orientations of the spin. Limiting the interactions to only neighboring spins – which is often a good approximation – the Hamiltonian of the model is written as

$$H = \frac{1}{2}\sum_{j=1}^{N-1}\left(S_j^+ S_{j+1}^- + S_j^- S_{j+1}^+\right) + \Delta\sum_{j=1}^{N-1} S_j^z S_{j+1}^z \tag{1}$$

where S_j^+, S_j^- operators change, while S_j^z measures the orientation of the spin on lattice site j. The overall behavior of the system can be tuned via the relevant parameter Δ. The explicit matrix representation of an operator $\mathcal{O}_j$ acting on site j of a chain with N spins is given as

$$\mathcal{O}_j = \bigotimes_{i=1}^{j-1}\mathbb{I}\otimes\mathcal{O}\otimes\bigotimes_{i=j+1}^{N}\mathbb{I} \tag{2}$$

where $\mathbb{I}$ is the identity and $\mathcal{O}$ is one of the followings

$$S^+ = \begin{pmatrix}0 & 1\\ 0 & 0\end{pmatrix}, \qquad S^- = \begin{pmatrix}0 & 0\\ 1 & 0\end{pmatrix}, \qquad S^z = \frac{1}{2}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}. \tag{3}$$

The Hamiltonian of N spins acts on the tensor product space of dimension 2^N, that is the dimension of the complete Hilbert space grows exponentially as the size of the system increases. In the rest of the paper benchmark results are shown for $\Delta = 1$.

2.2. Hubbard model

The Hubbard model was introduced to describe electrons in solids to characterize the transition between insulating and conducting systems. The single-band Hubbard model provides appropriate description of low temperature systems where all particles are in the lowest Bloch band and the long-ranged interactions between the particles can be neglected due to strong screening effects [24]. More recently various multi-band Hubbard models have been applied to high-temperature superconductivity [25] and systems of higher spin to understand the behavior of optically trapped ultracold atoms [7].

In the general spin-F system each lattice site is characterized by $2F+1$ two dimensional vectors. Each vector is assigned with a distinct label (from $\{-F, -F+1, \ldots, F-1, F\}$) called spin polarization value (denoted by σ). A vector assigned to a spin polarization σ describes two orthogonal states: the site is occupied ($[0; 1]$) by the particle of spin polarization σ or not ($[1; 0]$). As a consequence, a lattice site of spin-F possesses 2^{2F+1} internal degrees of freedom.

The lattice model of interacting particles of spin-F consists of two competing terms: the kinetic term, which describes the tunneling of particles between neighboring lattice sites, and the local potential term, which describes on-site density–density interaction measuring the attraction or repulsion between the interacting particles. The single-band, fermionic Hubbard model of spin-F is defined on a chain with N sites as

$$H = -t\sum_{j=1}^{N-1}\sum_{\sigma=-F}^{F}\left(c_{j,\sigma}^\dagger c_{j+1,\sigma}^\dagger + \text{h.c.}\right) + \frac{U}{2}\sum_{j=1}^{N}\sum_{\sigma\neq\sigma'} n_{j,\sigma}^\dagger n_{j,\sigma'}^\dagger \tag{4}$$

where t measures the hopping amplitude between neighboring sites and U is the interaction strength. Creation and annihilation operator acting on site j with spin polarization σ, denoted as $c_{j,\sigma}^\dagger$ and $c_{j,\sigma}$, adds or removes a particle located on site j with spin polarization σ. The particle density of spin polarization σ on site j is measured by operator $n_{j,\sigma}^\dagger = c_{j,\sigma}^\dagger c_{j,\sigma}^\dagger$. The explicit matrix representation of an operator $\mathcal{O}_{j,\sigma}$ acting on site j and polarization σ is constructed as

$$\mathcal{O}_{j,\sigma} = \bigotimes_{i=1}^{F'(j-1)}\Phi\otimes\mathcal{O}_\sigma\otimes\bigotimes_{i=F'(j+1)}^{F'N}\mathbb{I} \tag{5}$$

$$\mathcal{O}_\sigma = \bigotimes_{i=-F}^{\sigma-1}\Phi\otimes\mathcal{O}\otimes\bigotimes_{i=\sigma+1}^{F}\mathbb{I} \tag{6}$$

$$\Phi = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \tag{7}$$

where $F' = 2F+1$, $\mathbb{I}$ is the identity, Φ is the fermionic phase-factor and $\mathcal{O}$ is one of the followings

$$c^\dagger = \begin{pmatrix}0 & 0\\ 1 & 0\end{pmatrix}, \qquad c = \begin{pmatrix}0 & 1\\ 0 & 0\end{pmatrix}. \tag{8}$$

The Hamiltonian describing the spin-F system of N lattice sites acts on the tensor product space of dimension $2^{F'N}$, and similarly to the Heisenberg model, the dimension of the complete Hilbert space blows up exponentially. Comparing to the bosonic operators of the Heisenberg model, the key differences in the construction of operators are the appearance of internal quantum number, σ, and the presence of the phase-factor describing the antisymmetric nature

of fermionic systems. To ease the comparison of the two models only the $F = \frac{1}{2}$ case is presented, however, the observed tendencies are valid for higher F values. In the rest of the paper benchmark results are shown for $U = 1$.

3. Symmetries to be exploited

In many systems the Hamilton operator does not change the value of a measurable quantity, i.e., it commutes with the operator connected to that measurable quantity. These operators are called symmetry operators and can be used to cast the Hilbert space to smaller independent subspaces [26]. Consequently, instead of solving a large matrix eigenvalue problem, the eigenvalue spectrum can be determined by solving several smaller problems. In the Heisenberg model the total spin projection, $S_z = \sum_{j=1}^{N} S_j^z$, is such a symmetry operator. Meanwhile, in the Hubbard model of spin-F the total particle number associated to each spin polarization σ, $N_\sigma = \sum_{j=1}^{N} n_{j,\sigma}$, is conserved. Thus, the distinct quantum numbers helps to partition the Hilbert space into multiple independent subspaces corresponding to a given combination of quantum number values.

A given symmetry operator shares the same eigenvectors of the Hamiltonian, thus the eigenstates of the Hamiltonian can be labeled by the eigenvalues of the symmetry operator (*quantum number*, Q), and the Hilbert space can be decomposed into subspaces (*sectors*) spanned by the eigenvectors of each quantum number value [27]. Introducing a quantum number based representation, the sparse operators (Eqs. (2) and (5)) can be decomposed to a set of smaller but dense matrices, furthermore the Hamiltonian operator (Eqs. (1) and (4)) becomes blockdiagonal.

4. Algorithm

The DMRG approach has two phases, in the *infinite-lattice algorithm* the approximated Hilbert space of a finite system of N interacting spins is built up iteratively, while in the optional *finite-lattice algorithm* the number of the interacting spins is fixed and further iterations are carried out to increase the accuracy of the computed results. As in both cases the iterations are very similar, for the sake of simplicity, we consider only the infinite-lattice algorithm. The detailed description of the algorithm can be found in the original work [1] and various reviews [2,3], here only the key steps of an iteration of the infinite-lattice algorithm are summarized in Algorithm 1 providing the basis of our analysis.

Algorithm 1 One iteration of the infinite-lattice algorithm

1: Load a left and a right block.
2: Form the superblock configuration.
3: Compute the lowest eigenstate of the superblock Hamiltonian H_{SB}. (Davidson method)
4: **for** each block **do**
5: Construct the density matrix for the given block from the lowest eigenstate.
6: Full diagonalization of the density matrix. (Divide-and-conquer eigenvalue method)
7: Renormalize the basis of the block by keeping states with high eigenvalues.

In the two-site DMRG procedure four subsystems (left block describing l sites, 1 site, 1 site, right block describing r sites) compose the finite system of $N = (l+2+r)$ sites called *superblock*. The sites contained in each block are described maximally by m, optimally chosen states, which can be significantly smaller than the exactly required q^l or q^r basis, where q is the degree of freedom of one site. As the central sites of the superblock are represented exactly by

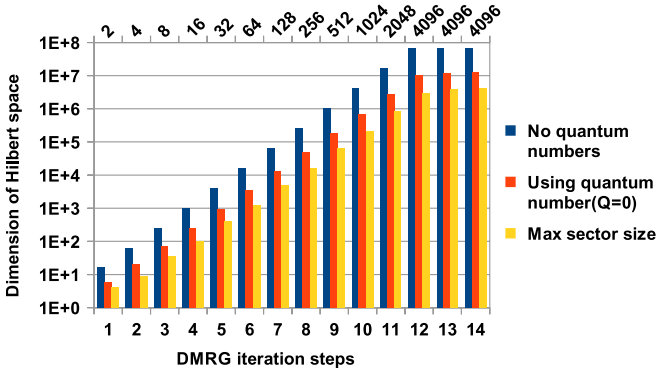

Fig. 1. Exploiting the projection symmetry in the Heisenberg model the Hilbert space of the superblock can be restricted to the subspace corresponding to $Q = 0$. At the top of the chart, labels indicate the number of retained block states ($m = 4096$).

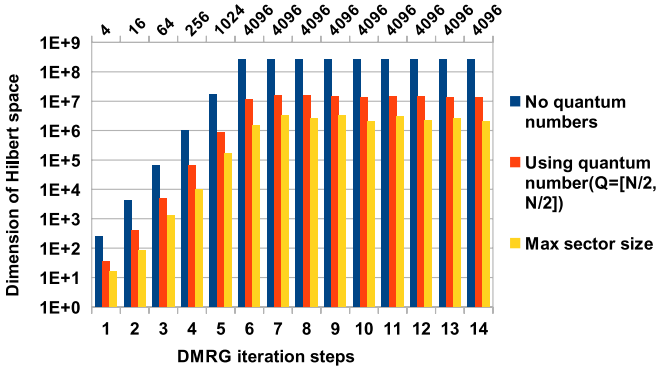

Fig. 2. Exploiting the conservation of particle number in the spin-$\frac{1}{2}$ Hubbard model the Hilbert space of the superblock can be restricted to the subspace corresponding to $Q = [\frac{N}{2}, \frac{N}{2}]$. At the top of the chart, labels indicate the number of retained block states ($m = 4096$).

q–q states, the size of the superblock Hilbert space is q^2m^2. Considering, however, the symmetries mentioned above, the problem can be restricted to a subspace of the superblock corresponding to a particular Q value. E.g., in case of Heisenberg and Hubbard models the size of the superblock Hilbert space can be reduced significantly as demonstrated in Figs. 1 and 2, respectively. It is, however, clear that even using symmetry operators the dimension of the reduced space grows exponentially with the size of the lattice (if no truncation is done).

The infinite-lattice algorithm starts with the four site configuration, where each block contains a single spin. In each iteration step both blocks are enlarged by a single site, making the complete system increase by two, until the desired system size, N, is reached. In each iteration of the DMRG algorithm, the lowest-lying eigenvector of the corresponding superblock Hamiltonian (H_{SB}) is obtained by the iterative Davidson or Lanczos algorithm. (In the paper the Davidson algorithm is considered.) From the lowest eigenstate the density matrix is constructed which carry the information how to optimally truncate the basis of the enlarged block ($m \ll q^{l+1}$) in order to keep the problem size manageable [28].

The Davidson algorithm starts with an initial guess vector (called *starting vector*), which affects the number of necessary Davidson iterations. In case of the finite-lattice algorithm, the transformed wavefunction of a previous DMRG iteration can be used as a starting vector to significantly decrease the number of iterations. However, in case of the infinite-lattice algorithm, the wavefunction transformation cannot be applied. In the presented work, a random starting vector was used as we were focusing on the acceleration of the infinite-lattice algorithm, which runs slowly on the CPU due to the large number of Davidson iterations. In a future work, the finite-lattice algorithm will also be addressed, in

which case further parts of the DMRG algorithm (e.g. density matrix diagonalization) will be considered for acceleration to maintain the speed-up presented here.

The most time-consuming part of a full iteration is the step of the Davidson routine which carries out the projection operation ($X' = H_{SB}X$). Instead of constructing and storing the enormous H_{SB} matrix of size $\mathcal{O}\left(m^4\right)$ explicitly, it is computationally favorable to obtain the projected vector X' directly from the matrices of size $\mathcal{O}\left(m^2\right)$ composing H_{SB}.

The H_{SB} can be directly expressed by the operators of the original four subsystems (*l*-1-1-*r strategy)* or by the operators of two intermediate systems (*LR strategy*), so called *enlarged blocks*, which come from the contraction of each block with its neighboring site. In the current implementation only the second strategy is investigated, however, the first one is also straightforward and will be included in the near future.

There are several practical benefits of these strategies. First of all, the memory bandwidth limited matrix–vector multiplication (BLAS Level 2) is converted to matrix–matrix multiplication (BLAS Level 3) which can be efficiently accelerated. Secondly, skipping of the explicit Kronecker multiplications not only restructures the computation, but decreases the number of operations. Finally, both strategies drastically decrease the size of the matrices which take part in the operations and thus the memory footprint of the algorithm. In case of LR and *l*-1-1-*r* strategy the largest matrix has a size of $O((mq)^2)$ and $O((m)^2)$, respectively. The second strategy is more favorable in extreme situations when the GPU memory is limited and q (internal degrees of freedom) is large (e.g. spin-F Hubbard model with large F).

4.1. LR strategy

In the *LR strategy* the H_{SB} is expressed with operators $A_\alpha^{(L)}$ and $B_\alpha^{(R)}$ defined on the left ($L := l + 1$) and right ($R := r + 1$) enlarged blocks, respectively, as

$$H_{SB} = \sum_\alpha A_\alpha^{(L)} \otimes B_\alpha^{(R)}, \tag{9}$$

where the index α iterates over the distinct operator combinations required to construct the superblock Hamiltonian. Exploiting Kronecker multiplication properties, the projected vector X' can be computed by matrix–matrix multiplications as

$$\tilde{X}' = \sum_\alpha A_\alpha^{(L)} \tilde{X} B_\alpha^{(R)T}, \tag{10}$$

where vector X of size $[B^{col}A^{col}]$ is *reshaped* to matrix $\tilde{X}$ of size $[B^{col}, A^{col}]$ and vector X' of size $[B^{row}A^{row}]$ is reshaped to matrix $\tilde{X}'$ of size $[B^{row}, A^{row}]$.

Algorithm 2 The computation of the projected vector X' in case of *l*-1-1-*r strategy* strategy.

Require: $size(X) = [D^{col}C^{col}B^{col}A^{col}]$
1: **function** PROJECTX_L11R(A, B, C, D, X)
2: X_1=reshape(X) **as** size(X_1)=$[D^{col}, C^{col}, B^{col}, A^{col}]$
3: **for** each (i, j) **do** $X_1'(:, :, i, j) = DX_1(:, :, i, j)$
4: **for** each (i, j) **do** $X_1''(:, :, i, j) = X_1'(:, :, i, j)C^T$
5: X_2=reshape(X_1'') **as** size(X_2)=$[D^{row}C^{row}, B^{col}, A^{col}]$
6: **for** each (i) **do** $X_2'(:, :, i) = X_2(:, :, i)B^T$
7: X_3=reshape(X_2') **as** size(X_3)=$[D^{row}C^{row}B^{row}, A^{col}]$
8: $X_3' = X_3A^T$
9: X'=reshape(X_3') **as** size(X')=$[D^{row}C^{row}B^{row}A^{row}]$
10: **return** X'
Notation: The colon operator indicates the range of all possible indices.

In the practical implementation Eq. (10) operates on even smaller matrices as the operators are decomposed according to quantum numbers. Instead of a sparse matrix $A^{(L)}$ several dense matrices $A^{(L)}_{q_i \to q_j}$ are stored representing how $A^{(L)}$ transforms the subspace (sector) corresponding to q_i to the one corresponding to q_j. To compute X' in case of a given $A_\alpha^{(L)}$, $B_\alpha^{(R)}$ operator pair all possible $A^{(L)}_{\alpha, q_i \to q_j}$, $B^{(R)}_{\alpha, q_k \to q_l}$ transition pairs shall be submitted to Eq. (10) and each time only the corresponding *ik* and *jl* segments of X and X' shall be used as

$$\tilde{X}'_{jl} += A^{(L)}_{\alpha, i \to j} \tilde{X}_{ik} B^{(R)T}_{\alpha, k \to l} \tag{11}$$

where $\tilde{X}_{ik}$ and $\tilde{X}'_{jl}$ indicate the reshaped *ik* and *jl* segment of vector X and X', respectively. Fortunately, the reshape operation has no computational cost as the data in the memory is untouched and only the row/col sizes are changing.

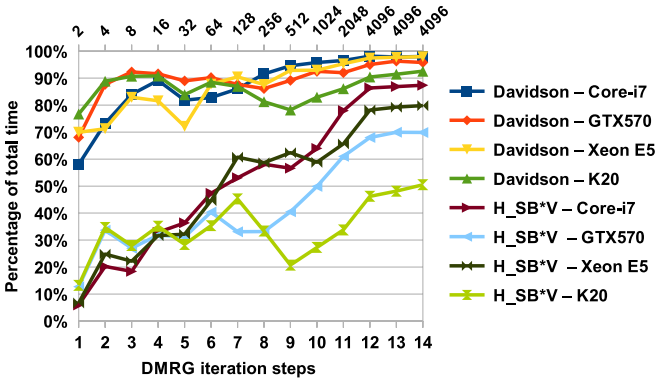

Fig. 3. Heisenberg model: Runtime of the Davidson algorithm and its $H_{SB}V$ operation compared to the total time of a DMRG iteration step during the first 14 iteration steps. CPU-only versions are indicated by Core-i7 and Xeon E5, while hybrid versions are indicated by GTX 570 and K20. At the top of the chart, labels indicate the number of retained block states ($m = 4096$).

4.2. l-1-1-r strategy

In the *l*-1-1-*r strategy* the H_{SB} is expressed by the operators of the four subsystems:

$$H_{SB} = \sum_\alpha A_\alpha^{(l)} \otimes a_\alpha \otimes b_\alpha \otimes B_\alpha^{(r)}, \tag{12}$$

where the index α again iterates over the distinct operator combinations required to construct the superblock Hamiltonian.

Similarly to the *LR* strategy, Kronecker multiplication properties can be exploited to compute the projected vector X' efficiently with matrix–matrix operations, however, in this case a more complicated data storage and several tensor multiplications are needed to avoid unnecessary memcopy operations. Using the procedure PROJECTX_L11R(), which computes the projected vector X' for one matrix quadruplet and is described in Algorithm 2, the H_{SB} can be calculated as

$$X' = \sum_\alpha \text{PROJECTX\_L11R}(A_\alpha^{(l)}, a_\alpha^\dagger, b_\alpha, B_\alpha^{(r)}, X). \tag{13}$$

In the similar manner as shown in *LR* strategy, $A^{(l)}$, a, b and $B^{(r)}$ operators can be decomposed according to quantum numbers and instead of large sparse matrix operations several smaller dense matrix operations shall be submitted to Algorithm 2. Furthermore, none of the reshape operations of Algorithm 2 involves practical data movement, only the size descriptor variables are changing.

5. Runtime & parallelism

In case of the Heisenberg and Hubbard model the runtime analysis of the DMRG algorithm is shown in Figs. 3 and 4, respectively. As the Davidson algorithm, which is summarized in Algorithm 3, is

the most time-dominant part and takes more than 97% of the total time in the CPU-only reference implementation, it has been chosen for acceleration. Unfortunately, the full Davidson algorithm cannot be implemented on the GPU as the problem size in real world simulations usually exceeds the GPU memory size. Instead, a hybrid approach shall be implemented, which can adjust the GPU workload according to the available GPU memory and the CPU–GPU performance ratio.

The second most time-consuming part of the DMRG algorithm is the diagonalization of the density matrices. In this step all the eigenvalues and eigenvectors of the density matrices are computed on the CPU with the Divide-and-conquer eigenvalue method [29] that is implemented in the Intel MKL Library [30]. In the presented experiments, the diagonalization takes only 1%–3% of the runtime of Davidson algorithm, however, if the number of Davidson iterations can be efficiently decreased, it may become a significant part of the DMRG algorithm.

Algorithm 3 One iteration of the Davidson algorithm

Require: Previous i basis vectors are already computed. (The starting vector is used for the first iteration.)

1: **function** DAVIDSONITER(i)
2: $W(:,i) = H_{SB} \cdot V(:,i)$ ▷ BLAS-3: dgemm()
3: $B(:,i) = V^T \cdot W(:,i)$ ▷ BLAS-2: dgemv_trans()
4: $[\lambda, y] \longleftarrow$ get smallest eigenvalue and vector of B
5: $x = V \cdot y$ ▷ BLAS-2: dgemv()
6: $r = -\lambda \cdot x + W \cdot y$
7: **if** norm(r) ≈ 0 **then**
8: return with x and success
9: **else**
10: correct r (*precondintioning*)
11: // orthonormalize r against V:
12: $s = V^T \cdot r$ ▷ BLAS-2: dgemv_trans()
13: $r = r - V \cdot s$ ▷ BLAS-2: dgemv()
14: normalize r and insert into $V(:, i+1)$
15: return without success

Notation: The colon operator indicates the range of all possible indices.

In the Davidson algorithm inherent parallelism can be observed at two levels. First, at low level, all the matrix and vector operations can be accelerated. Secondly, at the level of projection computation (line 2 in Algorithm 3), which is the most time-dominant part of the Davidson algorithm itself taking more than 75% of the total time, the projection can be computed as a sum of the independent $(AX)B^T$ operations (see Eq. (10)).

At low level, the CPU part of the algorithm (regardless the GPU is enabled or not) uses the Basic Linear Algebra Subroutine (BLAS) interface and the Intel MKL Library for algebraic operations including operator contractions, inner operations of both Davidson and full diagonalization algorithms, and operator transformations. Functions of the library are optimized for parallel execution and utilize the CPU cores efficiently for the relevant matrix sizes. Unfortunately, in the Davidson algorithm, all the operations except the projection are BLAS level 2 matrix–vector multiplications, which are bandwidth limited and not ideal for acceleration. There is a block extension [31] of the algorithm, the so called Davidson–Liu, to determine a few of the lowest eigenvalues, where more than one candidate vectors are added at once resulting in BLAS level 3 operations, however, in the current DMRG implementation only the lowest eigenvalue is investigated. The remaining option is to store as much data in GPU memory as possible and execute the corresponding operations on GPU.

At the level of projection operation the independence of matrix multiplications provides a straightforward hybrid parallelization

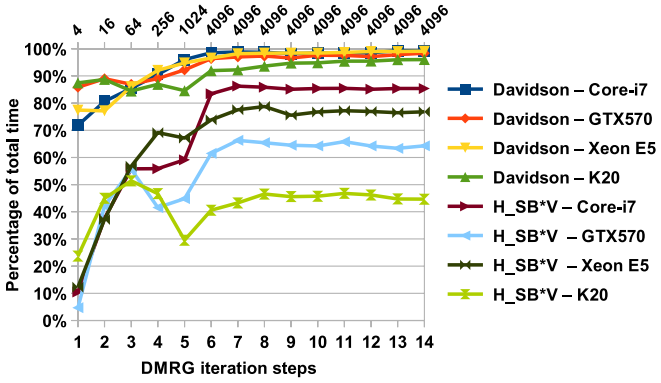

Fig. 4. Similar to Fig. 3 but for the Hubbard model.

and a future multi-GPU modification of the current implementation. Acceleration can be improved by developing the appropriate scheduling of the matrix operations for different matrix sizes and architectures.

6. Accelerating matrix–vector multiplications

Jacobi–Davidson version [32] of the original Davidson algorithm [33] is a preconditioned subspace iteration technique [34] aimed at computing a few of the extreme eigenpairs of large sparse symmetric matrices and commonly used in the DMRG implementations [2,3]. In the presented work the [35] version of the algorithm (available in Netlib [36]) is implemented.

In each iteration the subspace is extended with a new base vector ($V(:,i)$), which is stored in the memory accompanied by its projection ($W(:,i)$). As the size of these vectors can be very large (see Figs. 1 and 2) depending on the model and the number of retained block states (m), they cannot be fully stored in the GPU memory. However, in order to accelerate a matrix–vector multiplication with GPU, at least the matrix shall be stored in the GPU memory. In the current implementation the matrix of the basis vectors (V), which is used four times (see comments in Algorithm 3) in BLAS level 2 operations, has been selected to be stored, although the storage of the projected vector matrix (W) can be added later as well.

In the implemented version of the Davidson algorithm, the number of basis vectors can be limited by a user-defined threshold to keep the memory requirements manageable. When the threshold is reached, the iteration is restarted and the previous estimate vector (see line 5 in Algorithm 3) is used as the first basis vector. In our measurements the threshold was set to 20, hence the width of matrix V was varying between 1 and 20.

In each iteration the new basis vector is loaded to the GPU memory in the background (if there is available space) and the workload of the BLAS level 2 operations is shared between the CPU and GPU: CPU process the new basis vector, while GPU operates on the older ones. With this technique the power of both CPU and GPU can be exploited and the transfer time of the matrix can be hidden. The implementation is flexible: if there is no more space on the GPU or the CPU performance justifies it, more than one base vector can be processed on the CPU leaving less work for GPU. As shown in Fig. 5, where the storage requirement of V is compared with available free space after the projection operation, V cannot be fully stored on a GTX 570 GPU even in case of the simple Heisenberg model ($m = 4096$).

There are two types of BLAS level 2 operations: V^TX and VX indicated by *gemv_trans()* and *gemv()* in Algorithm 3, respectively. In the first case the multiplier vector can be loaded while in the second case the result can be written in smaller parts ($\sim 5e5$) to overlap with the computation.

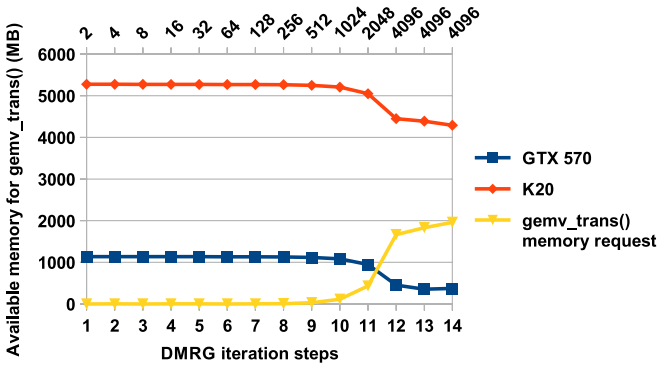

Fig. 5. Remained free GPU memory after the projection operation, which can be used for gemv_trans() and the memory request of the gemv_trans() in case of the Heisenberg model. At the top of the chart, labels indicate the number of retained block states ($m = 4096$).

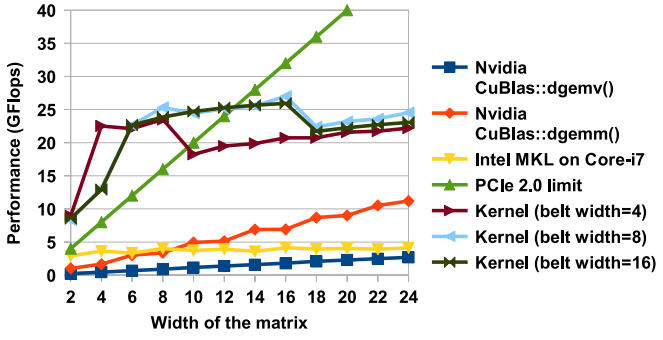

Fig. 6. GTX 570: Performance of the presented *gemv_trans()* kernel with different belt widths is compared to the performance of the available implementations in case of matrix height 5*e*5. Additionally, the PCIe limit is displayed as PCIe throughput limits the performance of the GPU acceleration if the transfer of the multiplier vector cannot be avoided.

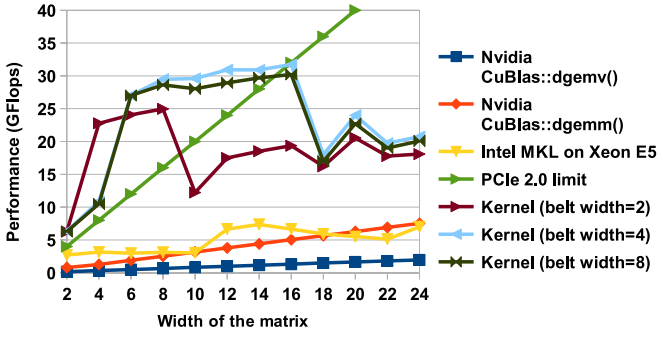

Fig. 7. K20, no shuffle operation in the kernel: Performance of the presented *gemv_trans()* kernel with different belt widths is compared to the performance of the available implementations in case of matrix height 5*e*5. Additionally, the PCIe limit is displayed as PCIe throughput limits the performance of the GPU acceleration if the transfer of the multiplier vector cannot be avoided.

6.1. *gemv_trans()*

In case of gemv_trans() both MKL and CuBlas libraries give poor performance for the special asymmetric matrix size ($\sim 5e5 \times 20$) required in our application (see Figs. 6–10), therefore, a new CUDA kernel has been designed. The presented results are measured without data communication, as in case of line 3 the multiplier vector is already in the GPU memory providing ideal acceleration. To estimate the performance of line 12, where the multiplier vector has to be loaded, the limiting factor of the PCIe 2.0 is also displayed. In this case the overall performance cannot exceed the PCIe limit even with overlapped communication. In both cases the estimation

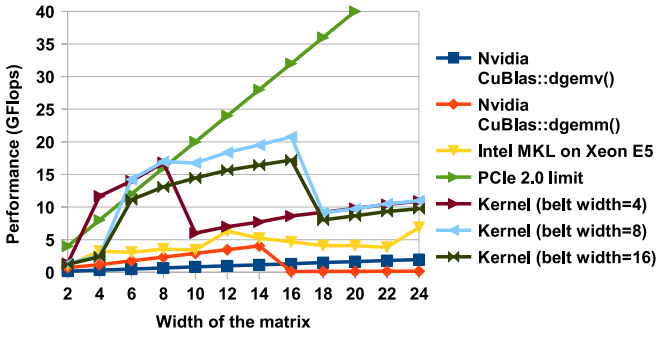

Fig. 8. K20, shuffle operation enabled: Performance of the presented *gemv_trans()* kernel with different belt widths is compared to the performance of the available implementations in case of matrix height 1*e*5. Additionally, the PCIe limit is displayed as PCIe throughput limits the performance of the GPU acceleration if the transfer of the multiplier vector cannot be avoided.

of the overall acceleration shall be carried out in an integral fashion, as in each iteration the thickness of the matrix is increased by one until the user-defined threshold (20 in the presented DMRG test-cases) is reached.

The basic idea of the new kernel (see Algorithm 4) can be summarized as follows. Each thread is associated with a column of the matrix. Each thread loads the corresponding vector element and multiplies the elements of the associated column. As threads of a warp load consecutive elements of the vector and the matrix, the coalesced reading is obvious. If the number of threads (grid size * thread block size) is less than the length of the matrix, each thread is associated with a new unprocessed column (coalesced readings again) as long as there is any. After processing a new column each thread accumulates the results to the results of the first column. Finally, the accumulated results shall be summed across the threads, which can be efficiently done via a sum reduction [37] in shared memory. If the belt is smaller than the width of the matrix, the whole procedure can be repeated (outer loop).

Algorithm 4 Pseudocode of the proposed kernel for asymmetric gemv_trans()

Require: *thread_number* is the global index of a thread
1: **function** GEMV_TRANS(MTX, MTX_WIDTH,
2: MTX_LEN, VEC)
3: Allocate shared memory for a thread block(*s_block*)
4: **for** each *belt* **do**
5: Fill *s_block* with zeros
6: **for** (i = thread_number; i < mtx_len;
7: i += num_of_threads) **do**
8: Load vec[i] to a private memory (*p_vec*)
9: **for** each element of *i*th column of *belt* **do**
10: Load element, multiply with *p_vec* and
11: accumulate the product to *s_block*
12: Sum reduction in *s_block*
13: **if** this is the first thread of the block **then**
14: Save first column of *s_block*

The size of the shared memory requirement of a thread block, which is equal to the size of a thread block multiplied with the height of the belt, can be a limiting factor of the performance because in case of large shared memory usage less thread blocks can be assigned to one physical multiprocessor. In the presented measurements the optimal height of the belt has been investigated, however, even with the optimal height the performance decreases as the width of the matrix increases. For extreme, asymmetric matrices, which are used in our application, significant speed-up (x4-5) can be reached compared to the CuBlas library, however, as the matrix tends to be more symmetric the performance

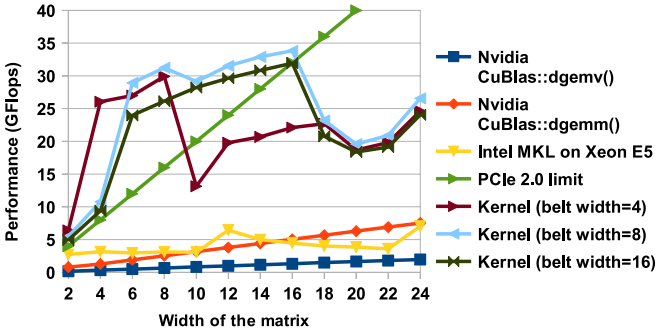

Fig. 9. Similar to Fig. 8 but for matrix height 5*e*5.

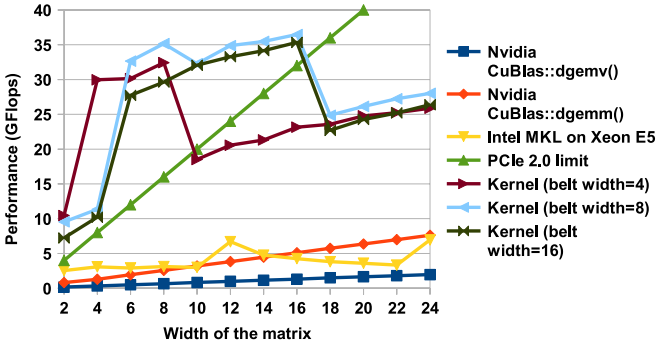

Fig. 10. Similar to Figs. 8 and 9 but for matrix height 1*e*6.

of the CuBlas::dgemm() operation increases and exceeds the performance of the new kernel.

In case of the new Kepler architecture (K20), in which Streaming Multiprocessor has significantly more CUDA Cores than the SM of Fermi GPUs (GTX 570), the per-multiprocessor warp occupancy shall be increased to use all the available cores [38]. A new warp-level intrinsic called the shuffle operation can be used to decrease the shared memory requirement of the sum reduction algorithm to increase the occupancy. If shuffle operation is enabled, each thread accumulates the partial products (line 10 in Algorithm 4) in a private memory, and the shuffle operation is used to summarize the results of the threads of the same warp. Consequently, only one column per warp has to be stored in the shared memory, which decreases the shared memory requirement to the number of warps in a block multiplied with the height of the belt. In practice, the cycle at line 9 can be unrolled with macros as allocation of static arrays in private memory is not possible. In Figs. 8–10 the results of the new kernel extended with the shuffle operation are displayed. The height of the optimal belt is slightly increased as the shared memory request is decreased. Unfortunately, the shuffle operation provides only a small performance gain in case of our kernel (compare Figs. 7 and 9).

The performance of the kernel is mainly dominated by the speed of the coalesced reading of the matrix elements. The gemv_trans() operation is bandwidth limited in both CPU and GPU architectures, however, the memory bandwidth on GPU (e.g. GDDR5 in GTX 570: 152 GB/s or GDDR5 in K20 208 GB/s) is usually higher than on CPU (e.g. DDR3-1333 in dual channel with Core-i7: 21.2 GB/s or DDR3-1066 in quad channel with Xeon E5: 34.1 GB/s). The maximal memory throughput reached by the new kernel (measured with matrix size $16 \times 5e5$) was 114.7 GB/s, 134.8 GB/s and 143.7 GB/s on GTX 570, on K20 without shuffle and on K20 with shuffle, respectively.

In the presented DMRG implementation, the shuffle operation is enabled and the best kernel is invoked based on the matrix width. The results of the acceleration of the selected BLAS level 2 operations of the Davidson algorithm on the Xeon E5 + K20

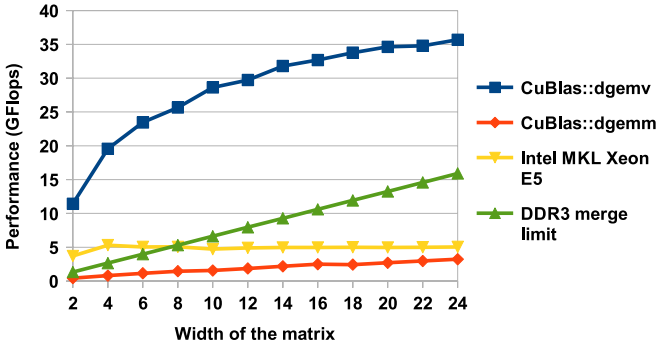

Fig. 11. Performance of the gemv_normal() operation of the available implementations in case of K20. Additionally DDR3 limit is displayed, as in case of a CPU–GPU hybrid implementation, the merge operation is limited by the DDR3 throughput.

Table 1
Total execution time of the accelerated matrix–vector operations of the Davidson algorithm (see Algorithm 3) during the first 14 iterations of the DMRG algorithm ($m = 4096$).

		Xeon E5	Xeon E5 + K20	Speedup
Heisenberg model	Line 3	20.21	4.64	4.36
	Line 5	19.07	10.45	1.83
	Line 12	20.05	5.94	3.38
	Line 13	17.55	9.68	1.81
Hubbard model	Line 3	113.80	22.66	5.02
	Line 5	94.29	54.22	1.74
	Line 12	114.00	37.67	3.03
	Line 13	87.29	50.21	1.74

architecture are summarized in Table 1. (On the GTX 570 card the memory is too small to accelerate other operations besides the projection.) Line 3 is accelerated well as no extra communication is needed, while the other operations are either limited by the PCIe or the DDR3 throughput.

6.2. *gemv()*

The gemv() operation can be efficiently accelerated by the standard CuBlas library even in case of asymmetric matrices (see Fig. 11). Unfortunately, merging of the CPU and GPU results is slow on one CPU thread and is the bottleneck of the acceleration. The implementation can be improved by multithreaded merging to enable quad channel memory or by computing everything on the GPU, however, this is not always possible.

7. Accelerating projection operation

The acceleration of the independent $(AX)B^T$ operations implementing the projection operation is based on the observation that A and B matrices are already available before the Davidson algorithm starts and do not change during the Davidson iterations. The necessary $(AX)B^T$ operations are described by a list of *operation records* in which each record contains all the necessary information to compute an operation like Eq. (11). For example, it stores information from which segment of X (*input*) to which segment of X' (*output*) the operation transforms.

The host side algorithm to handle the operation records is summarized in Algorithm 5. During the construction of the operation records the workload associated to each output is computed. (Multiple operations can use the same input or write the same output segment.) Next, the operation records are partitioned between CPU and GPU based on the performance ratio of the two architectures. To avoid merging of outputs all operation records corresponding to the same output shall be computed on the same architecture, however, to create a balanced workload partitioning, this is not

always possible. During partitioning the output associated to the largest workload is selected for GPU iteratively as long as the desired workload ratio is not exceeded. If the reached workload ratio is far from the desired, the operation records of the output associated to the next largest workload are partitioned between the two architectures.

Algorithm 5 Host side algorithm to handle the operation records

1: Create operation records and determine the workload (FLOP) for each output.
2: Partition the operation records between CPU and GPU based on their performance ratio and the output workload statistics.
3: Selects scheduling strategy for the operations to be computed on GPU.
4: Set-up the workload for GPU based on the selected strategy.

After partitioning the proper scheduling strategy is selected based on the memory requirements of the operation records. Three different strategies can be selected for three different uses cases, however, currently only the first two, more complex strategies have been implemented and tested. The first strategy (*4Streams*) is designed for small problem size, when all A, B, X, X' matrices and temporary matrices T for storing intermediate results can be held in the GPU memory. The second strategy (*NoStreams*) is designed for medium-sized problems, where all A, B and X' matrices can be stored in GPU memory, but from X and T only the processed matrices are allocated. In case of extra-sized problems a third strategy (*NoStreamAndStorage*) can be designed in which even A and B matrices cannot be fully stored in the GPU memory. The memory footprint of the matrices in case of different strategies are shown in Fig. 12. In the demonstrated examples the first strategy was available for all the DMRG iterations as the GPU cards had enough memory.

The A, B matrices used by the operation records associated to the GPU are loaded to the GPU before the Davidson algorithm starts, and the same matrices are used during the iterations. In theory, the transfer of the A, B matrices can be overlapped with computation of the first Davidson iteration as the communication time is typically 10%–45% of the computation time of a single Davidson iteration. Currently, the transfer does not generate overhead because the A, B matrices are transferred in the background during the CPU computes the diagonal of the superblock Hamiltonian used for preconditioning in the Davidson algorithm. If the diagonal computation is moved to the GPU and the number of Davidson iterations is very small, the overlapping can be advantageous, although, in these cases the DMRG iterations run fast anyway. As the X, X' matrices have to be transferred in each Davidson iterations, the overlapping of their transfer time is necessary and implemented in both presented strategies.

The $(AX)B^T$ operations are implemented using the cuBLAS library [39], which is a BLAS implementation dedicated for Nvidia GPUs. In the demonstrated strategies two important features of the GPUs are exploited, which are provided via the CUDA driver [40] and also accessible through the cuBLAS library. The first feature is that multiple CUDA kernels can be executed simultaneously on the GPU, while the second feature is that memory I/O operations can be executed in the background. From the aspect of programming both features can be accessed via the CUDA streams. Streams are sequences of operations that execute in issue-order, but operations in different streams may run concurrently or interleaved.

In the 4Streams strategy there is enough GPU memory to execute several $(AX)B^T$ simultaneously. One stream is created for each output and operations corresponding to a given output are assigned to the same stream to avoid interference. For each stream a sufficiently large temporary matrix is allocated to store the temporary result of AX.

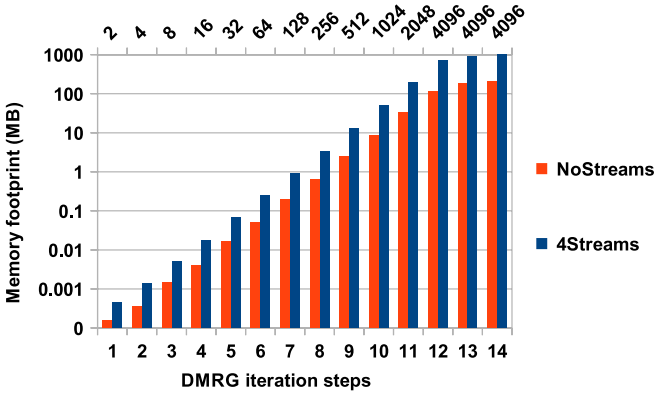

Fig. 12. GPU memory footprints of the two strategies are compared in case of the Heisenberg model. At the top of the chart, labels indicate the number of retained block states ($m = 4096$).

CUDA operations are dispatched to *hardware queues* in issue order [40]. To enable asynchronous concurrent kernel execution in CUDA environment memory transfers and kernels shall be issued in a breadth-first order. Inside the engine (kernel) queue an operation is dispatched if all preceding calls in the same stream have been completed and all preceding calls of the same queue have been dispatched. Consequently, to avoid blocking calls kernels of the same streams shall not be issued immediately after each other. As one $(AX)B^T$ operation consists of two kernels, the kernel calls shall be separated and interleaved with kernels of operations of other streams. To reach four parallel streams (hence the name of the strategy) kernels from four different stream shall be interleaved.

Algorithm 6 Grouping operation records in case of 4Streams strategy

Require: *records* is a vector containing objects of record type. Each record has a *stream* parameter indicating the CUDA stream it is associated to.

```
1: function orderAndGroupRecords(records, maxstream)
2:     Sort records by input frequency.
3:     setVisitedRecords.clear()
4:     for each record i do
5:         if i.stream ∈ setVisitedRecords then
6:             for each record j following i do
7:                 // Limiting swapping is possible
8:                 if j.stream ∉ in setVisitedRecords then
9:                     swap(i,j) and break
10:        if i.stream is ∉ in setVisitedRecords then
11:            vecGroup.last().insert(i)
12:            setVisitedRecords.insert(i.stream)
13:            if setVisitedRecords.size()=maxstream then
14:                vecGroup.add(new Group)
15:                setVisitedRecords.clear()
16:        else
17:            vecGroup.add(new Group)
18:            vecGroup.last().insert(i)
19:            setVisitedRecords.clear()
20:            setVisitedRecords.insert(i)
    return vecGroups
```

Remark: At line 7, in practice, the swapping of the records of the first group can be limited to help I/O overlap.

The overlapping of the transfer time of input segments with kernel execution makes further constraints on the order of the operation records: only those operation records shall be issued which use already loaded input segments. To be able to interleave

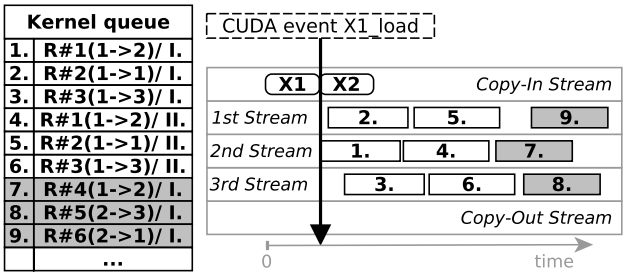

Fig. 13. Interleaved operation records (on the left) and the resulting parallel execution (on the right). Each record contains two kernel calls (see lines 4 and 6 in Algorithm 7), which are indicated by roman letters. The lth kernel of the ith record is named as $R\#i(j \to k)/l$, where jth and kth indicate the affected input and output segments, respectively. The kernel queue illustrates the issue order of the kernels. The kernels of the first group are colored by white, while gray color indicates some of the kernels of the second group. A CUDA event is also displayed to demonstrate that the first kernel does not start until the first segment is loaded. Note that the 9th kernel cannot start until all the previously issued kernels have been started.

different streams, it is favorable to load the input segment first which is used by the most streams.

Algorithm 7 Dispatching operation records

1: **for** each group g **do**
2: **for** each record i in g **do**
3: Init copy of input segment X_i (when first used)
4: Init $T_i = (A_i X_i)$
5: **for** each record i in g **do**
6: Init $X'_i = T_i B_i^T$
7: **for** each output segments of X' **do** Init copy back.

In case of 4Streams strategy the reordered operation records are grouped (see Algorithm 6) such a way that the issuing of the kernels belonging to the same group can be interleaved (see Algorithm 7). First, operation records are sorted to load the more frequently used input segments earlier. Then, the records are iterated and each record is potentially swapped backwards to create groups of four consecutive operations belonging to four different streams. In practice some technical constraints have been added to slightly alter swapping behavior, which is not discussed here for the sake of simplicity.

CUDA operations are launched according to the operation records as summarized in Algorithm 7. For the sake of brevity the synchronization between the streams is not shown as it can be implemented with CUDA events in a straightforward way. (For example, if each operation waits for the transfer event specific to the utilized input segment and the operations are ordered properly, the transfer and computation time can be overlapped.) The same code can be used for both strategies as the NoStreams strategy can be represented by groups which contain only one operation record. In case of NoStreams strategy the preparation of the operation records is much simpler and contains only the sorting by input frequency to reach I/O overlap with computation. To illustrate the interleaved kernel calls, the overlapped I/O communication and the parallel kernel execution, a schematic diagram of a simplified example is shown in Fig. 13.

The performance of the two strategies is compared in Figs. 14 and 15. Significant improvement can only measured at medium sized matrices (100–800 for GTX 570 and 100–1500 for K20), in which case several operations shall be executed concurrently to keep all the CUDA cores busy required for high performance. Slightly bigger gain can be observed in case of K20 GPU which has 2496 Kepler CUDA cores as opposed to GTX 570 having only 480 Fermi CUDA cores. Operations on large matrices (∼1500 for GTX 570 and ∼3000 for K20) provide enough work for each CUDA

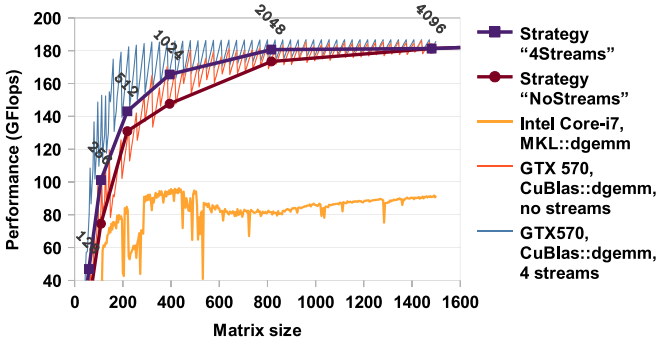

Fig. 14. GTX 570, Heisenberg model: Performance of the two strategies is compared. Additionally, the performance of CuBLAS and MKL dgemm() in reference measurements is displayed as the function of matrix size. Labels indicate the number of retained block states at the displayed DMRG iterations.

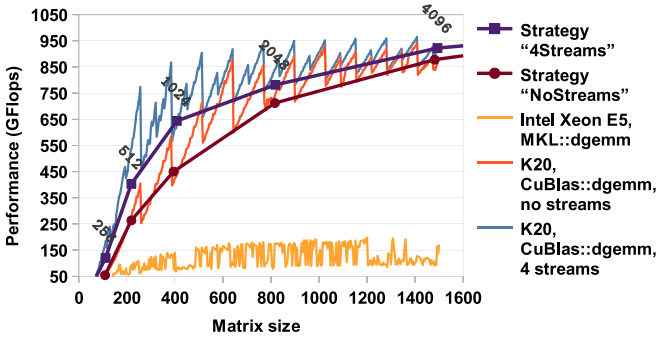

Fig. 15. Similar to Fig. 14 but on K20 architecture.

Table 2
Total execution time of the first 14 DMRG iterations ($m = 4096$) in case of different strategies.

	Model	NoStream (s)	4Streams (s)	Decrease
GTX 570	Heisenberg	671.54	652.58	2.82%
	Hubbard	2980.27	2957.82	0.75%
K20	Heisenberg	244.67	227.33	7.09%
	Hubbard	1056.33	1012.56	4.14%

core to approach the theoretical maximum double performance (180 GFlops for GTX 570 and 1.17 TFlops for K20) without streams.

The two strategies are also compared by the run-time of the simulated models in Table 2. In case of K20 the concurrent kernel execution has a slightly greater benefit, however, in both models operations on larger matrices, where concurrency has no benefit, dominates the run-time. In the models where more symmetries are enabled, the size of the matrices tends to be smaller, consequently, the concurrency also tends to be more effective. In case of the investigated models and $m = 4096$, the matrix sizes fall in the range where the 4Streams strategy is superior to the NoStreams strategy, hence the 4Streams strategy was used in the final benchmarks.

The performance results of the full projection computation including both CPU and GPU computations are shown in Figs. 16–19. The quality of the acceleration is highly affected by the applied workload ratio which depends on the performance ratio of the CPU and the GPU in case of the matrix multiplication. As the performance ratio depends on the size of matrices which are associated to the architectures, it cannot be determined exactly. In our solution a user-defined step function can be described in a configuration file to assign specific ratios to different matrix sizes. In each DMRG iteration the proper user-defined ratio is selected according to the average matrix size of the operation records. Fortunately, for large matrices, which dominate the overall run-time, a robust estimation can be found as demonstrated in the previously referenced

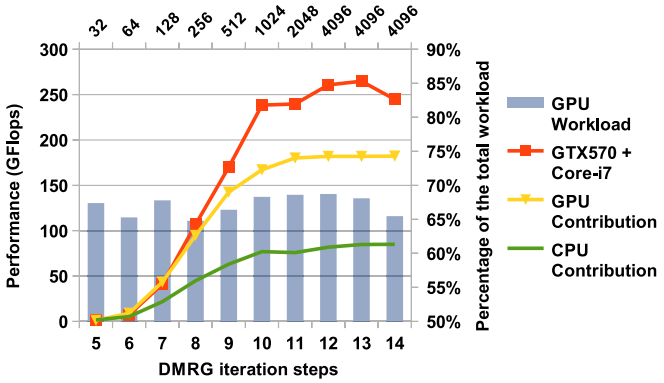

Fig. 16. GTX 570, Heisenberg model: Performance results of the hybrid CPU–GPU acceleration of the projection operation. Blue bars associated to the secondary vertical axis indicate the ratio of the current GPU workload. Via the workload ratios, the user-defined function used for describing the performance ratio of the architectures can also be estimated. At the top of the chart, labels indicate the number of retained block states ($m = 4096$).

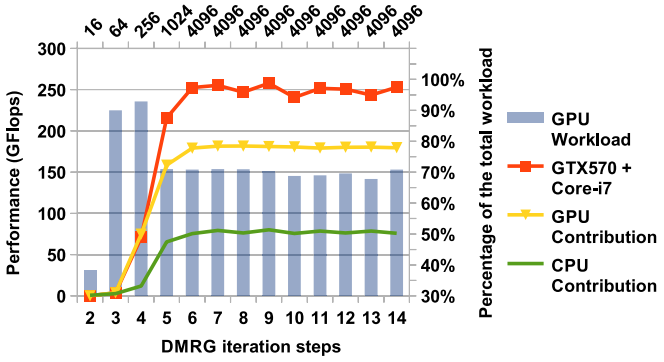

Fig. 17. Similar to Fig. 16 but for the Hubbard model on GTX 570.

figures. For smaller matrices the performance ratio is affected by other circumstances as well, but in this case the execution time tends to be negligible. If the workload is properly distributed 257.8 GFlops (×3.2 speed-up) and 1071.1 GFlops (×6.1 speed-up) can be reached on GTX 570 and on K20, respectively.

8. Limits of FPGA implementation

Field programmable gate arrays (FPGAs) are programmable integrated circuits used to realize reconfigurable digital circuits via configurable logic resources and routing. Although originally developed for telecommunication and digital signal processing applications novel high performance FPGAs (featuring high-speed embedded resources, plentiful on-chip memory, high-level programming tools and significantly lower power consumption than CPU or GPU) are promising candidates for high performance computing (HPC).

To estimate the performance of an FPGA implementation of the DMRG method the acceleration of the projection operation expressed as a series of dense matrix multiplications (see Eq. (11)) shall be investigated. The floating-point matrix–matrix multiplication was already implemented [41] on FPGA very efficiently using the rank-1 update scheme. Kumar et al. demonstrated that the performance is not limited by the PCIe bandwidth, which connects the FPGA to the host CPU, and nearly full utilization of the processing elements can be reached.

The idea behind the rank-1 update approach is that instead of inner products between the rows of the left matrix and the columns of the right matrix outer products between the columns of the left matrix and the rows of the right matrix are carried out

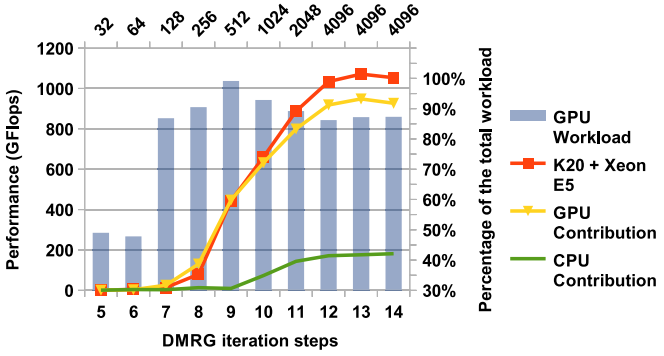

Fig. 18. Similar to Figs. 16 and 17 but for the Heisenberg model on K20.

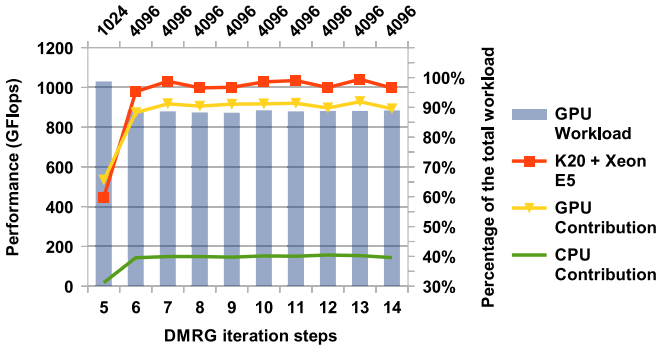

Fig. 19. Similar to Figs. 16–18 but for the Hubbard model on K20.

and resulting matrices are summarized. The advantage of the approach is that instead of multiply–accumulate operations (MACCs) multiply–add operations (MADDs) are used, which are independent from each other and can be pipelined to reach high processing element utilization.

Assuming the previously described design the processing elements can be fully utilized and the following best-case estimations can be made according to the area requirements of the floating-point units. In an ideal case at most 114 or 193 multiply–add units can be implemented on the largest Virtex-6 (XC6VSX475T) and Virtex-7 (XC7VX1140T) FPGAs, respectively. The estimated clock frequency of the architectures are 437.82 and 443.65MHz which would result in 99.82 and 171.2 GFLOPS computing performance. This performance achievement can be compared to the performance of the mid-range GTX 570 GPU used in the paper. As the development time in case of FPGA is still much longer than in case of GPU and the high-end GPUs could significantly outperform the FPGA in this problem class, the GPU architecture is the better candidate for the acceleration.

9. Implementation results

The implemented algorithm has been tested both on a mid-range (Intel Core-i7 2600 3.4 GHz CPU + NVidia GTX 570 GPU) and on a high-end configuration (Intel Xeon E5-2640 2.5 GHz CPU + NVidia K20 GPU); the results are displayed in Tables 3 and 4, respectively. All CPU-only measurements have been executed with multithreading enabled (4 threads on Core-i7 and 6 threads on Xeon E5). The mid-range configuration with GPU is approximately 2.3–2.4 times faster than without GPU, while the high-end configuration is accelerated by 3.4–3.5 times using the GPU.

To support the comparison of the results of the two investigated models the key parameters affecting computational complexity are summarized in Table 5. Using the same number of retained block states the Hubbard model has larger values for all key parameters except the maximum sector size and the maximum matrix size. In

Table 3
Heisenberg model: Total execution time of the first 14 DMRG iterations ($m = 4096$).

	Time (s)	Speed-up compared to	
		Core-i7	Xeon E5
Core-i7	1489.64	1	0.53
Core-i7 + GTX 570	652.58	2.28	1.21
Xeon E5	789.65	1.89	1
Xeon E5 + K20	227.33	6.55	3.47

Table 4
Hubbard model: Total execution time of the first 14 DMRG iterations ($m = 4096$).

	Time (s)	Speed-up compared to	
		Core-i7	Xeon E5
Core-i7	7210.72	1	0.48
Core-i7 + GTX 570	2957.82	2.44	1.16
Xeon E5	3433.16	2.10	1
Xeon E5 + K20	1012.56	7.12	3.39

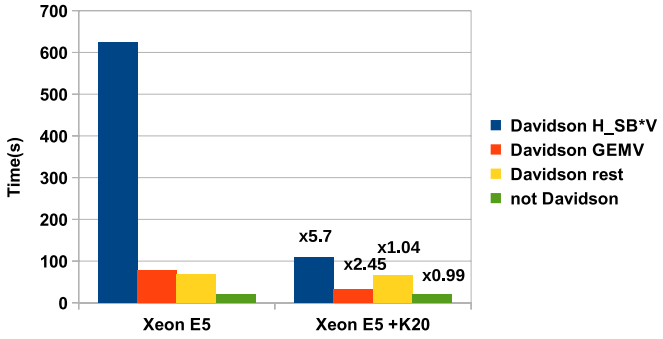

Fig. 20. K20, Heisenberg model: Acceleration of different parts of the algorithm is compared for $m = 4096$.

case of the Hubbard model more symmetries are exploited which results in smaller sectors and, consequently, smaller matrices.

In case of K20 the acceleration of the projection and the matrix–vector operations is compared in Figs. 20 and 21. The projection is accelerated by 5.7 times which is in accordance with the theoretical performance capabilities of the two architectures. Currently on Xeon processor (see Fig. 3) the projection operation is only accounted for 75% of the total run-time, therefore, the overall acceleration is also affected by the rest of the operations of the Davidson algorithm. Fortunately, as the number of retained states (m) increases the time-dominance of the projection also increases, which anticipates even better acceleration for real-world simulations with large m.

As the acceleration of the full Davidson algorithm can be limited by the GPU memory, an adaptive solution shall be implemented which accelerates as much of the algorithm as possible. Currently four matrix–vector operations of the algorithm are accelerated in case of sufficient GPU memory, however, later acceleration of the rest of the operations will be implemented as well.

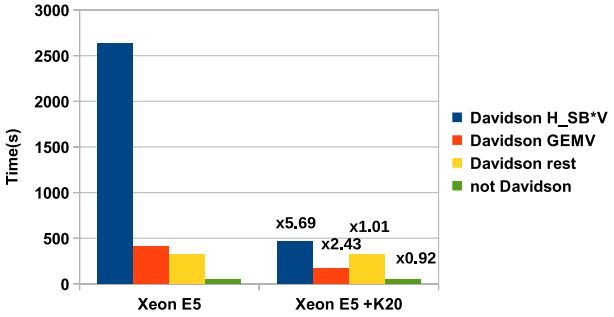

Fig. 21. K20, Hubbard model: Acceleration of different parts of the algorithm is compared for $m = 4096$.

10. Conclusion

In this paper acceleration of the DMRG algorithm using novel kilo-processor architectures (GPU, FPGA) has been investigated. The GPU architecture has been found to a promising accelerator, as the most time-dominant step of the algorithm, the projection operation, can be formulated as independent dense matrix multiplications, which are ideal workload for GPUs. Moreover, in case of high-end GPUs the acceleration of the projection is so remarkable, that it is worth to consider the acceleration of the rest of the algorithm to obtain a decent overall speed-up. In the presented implementation some asymmetric matrix–vector multiplications of the diagonalization have been identified as the second most time-dominant part of the algorithm and a new CUDA kernel has been designed to efficiently accelerate these operations. The resulting parallelized DMRG implementation is a hybrid CPU–GPU solution, which distributes the workload according to the performance and memory capabilities of the configurations and anticipates a straightforward multi-GPU extension, which is part of our current developments.

In subsequent works our DMRG implementation will be utilized to investigate more complex models focusing on physical problems of current interests, which are hard to treat due to their high computational demands. For example, high-spin fermionic systems relevant for ultracold atomic experiments and extensions to treat ab-initio quantum chemical applications [42–44] which are already under progress. Furthermore, a straightforward generalization of the presented algorithm to accelerate tensor network states (TNS) algorithms [45] is another promising research direction.

In general, FPGA chips have lower operating frequencies than in case of GPU architectures and the attached on-board memories are also smaller and slower. FPGAs can outperform other architectures in such problems where a custom arithmetic unit can reach a significantly better utilization, however, in case of dense matrix operations of the DMRG algorithm nearly ideal utilization of the CUDA cores is reached. As the estimated performance of the considered high-end FPGAs is around the mid-range GPUs and the development time for FPGA is significantly longer, the GPU architecture is preferred for the acceleration of the DMRG algorithm.

Table 5
Model comparison in case of the high-end configuration (Xeon E5 + K20).

	Heisenberg	Hubbard	Ratio
Total execution time of the investigated DMRG iterations	227.33	1012.56	4.45
Flop	1.22E+014	4.89E+014	4.01
Max H_{SB} size	12.24E+06	15.32E+06	1.25
Max sector size	4.00E+06	3.47E+06	0.87
Average number of sectors	9.36	50.71	5.42
Max matrix size	1704.23	1145.24	0.67
Peak GPU memory footprint	950.47	1155.48	1.22
Average number of Davidson iterations using a random starting vector	60.79	122.43	2.01

Acknowledgments

This research was supported by TÁMOP-4.2.2.C-11/1/KONV-2012-0004, TÁMOP-4.2.1./B-11/2/KMR-2011-002, TÁMOP-4.2.2./B-10/1-2010-0014 and the Hungarian Research Fund (OTKA) under Grant Nos. NN110360, K100908 and K84267.

References

[1] S.R. White, Density matrix formulation for quantum renormalization groups, Phys. Rev. Lett. 69 (1992) 2863–2866.
[2] R.M. Noack, S.R. Manmana, Diagonalization and numerical renormalization-group-based methods for interacting quantum systems, AIP Conf. Proc. 789 (2004) 93–163.
[3] U. Schollwöck, The density-matrix renormalization group, Rev. Modern Phys. 77 (2005) 259–315.
[4] Ö Legeza, R. Noack, J. Sólyom, L. Tincani, Applications of quantum information in the density-matrix renormalization group, in: Computational Many-Particle Physics, in: Lecture Notes in Physics, vol. 739, Springer-Verlag, Berlin, Heidelberg, 2008.
[5] W. Barford, Electronic and Optical Properties of Conjugated Polymers, Oxford University Press, 2005.
[6] G. Barcza, Ö Legeza, K.H. Marti, M. Reiher, Quantum-information analysis of electronic states of different molecular structures, Phys. Rev. A 83 (2011) 012508.
[7] M. Lewenstein, A. Sanpera, V. Ahufinger, B. Damski, A. Sen(De), U. Sen, Ultracold atomic gases in optical lattices: mimicking condensed matter physics and beyond, Adv. Phys. 56 (2) (2007) 243–379.
[8] S. Östlund, S. Rommer, Thermodynamic limit of density matrix renormalization, Phys. Rev. Lett. 75 (1995) 3537–3540.
[9] G. Vidal, Efficient classical simulation of slightly entangled quantum computations, Phys. Rev. Lett. 91 (2003) 147902.
[10] F. Verstraete, V. Murg, J. Cirac, Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems, Adv. Phys. 57 (2) (2008) 143–224.
[11] U. Schollwöck, The density-matrix renormalization group in the age of matrix product states, Ann. Physics 326 (1) (2011) 96–192 (Special Issue).
[12] V. Murg, F. Verstraete, J.I. Cirac, Exploring frustrated spin systems using projected entangled pair states, Phys. Rev. B 79 (2009) 195119.
[13] G. Vidal, Class of quantum many-body states that can be efficiently simulated, Phys. Rev. Lett. 101 (2008) 110501.
[14] W. Hackbusch, Tensor Spaces and Numerical Tensor Calculus, in: Springer Series in Computational Mathematics, vol. 42, Springer, 2012.
[15] G. Hager, E. Jeckelmann, H. Fehske, G. Wellein, Parallelization strategies for density matrix renormalization group algorithms on shared-memory systems, J. Comput. Phys. 194 (2) (2004) 795–808.
[16] G. Alvarez, Implementation of the su(2) hamiltonian symmetry for the {DMRG} algorithm, Comput. Phys. Commun. 183 (10) (2012) 2226–2232.
[17] G.K.-L. Chan, An algorithm for large scale density matrix renormalization group calculations, J. Chem. Phys. 120 (2004) 3172.
[18] Y. Kurashige, T. Yanai, High-performance ab initio density matrix renormalization group method: applicability to large-scale multireference problems for metal compounds, J. Chem. Phys. (2009) 130.
[19] S. Yamada, T. Imamura, M. Machida, Parallelization design on multi-core platforms in density matrix renormalization group toward 2-d quantum strongly-correlated systems, in: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2011, pp. 1–10.
[20] J. Rincòn, D.J. Garcìa, K. Hallberg, Improved parallelization techniques for the density matrix renormalization group, Comput. Phys. Commun. 181 (8) (2010) 1346–1351.
[21] E.M. Stoudenmire, S.R. White, Real-space parallel density matrix renormalization group, Phys. Rev. B 87 (2013) 155137.
[22] J. Yu, H.-C. Hsiao, Y.-J. Kao, Gpu accelerated tensor contractions in the plaquette renormalization scheme, Comput. & Fluids 45 (1) (2011) 55–58.
[23] M.J. Cawkwell, E.J. Sanville, S.M. Mniszewski, A.M.N. Niklasson, Computing the density matrix in electronic structure theory on graphics processing units, J. Chem. Theory Comput. 8 (11) (2012) 4094–4101.
[24] F. Gebhard, The Mott Metal–Insulator Transition: Models and Methods, Springer, 1997.
[25] P.W. Anderson, The Theory of Superconductivity in the High-Tc Cuprate Superconductors, Princeton University Press, 1997.
[26] A.I. Tóth, C.P. Moca, Ö Legeza, G. Zaránd, Density matrix numerical renormalization group for non-abelian symmetries, Phys. Rev. B 78 (2008) 245109.
[27] J.F. Cornwell, Group Theory in Physics, An Introduction, Academic Press, 1997.
[28] Ö Legeza, J. Röder, B.A. Hess, Controlling the accuracy of the density-matrix renormalization-group method: the dynamical block state selection approach, Phys. Rev. B 67 (2003) 125114.
[29] J.W. Demmel, Applied Numerical Linear Algebra, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
[30] Intel, Math kernel library 11.0, http://http://software.intel.com/en-us/intel-mkl, 2013.
[31] B. Liu, The simultaneous expansion for the solution of several of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, Numer. Algorithms Chem. Algebr. Method (1978) 49–53.
[32] G.G. Sleijpen, H. Van der Vorst, A Jacobi–Davidson iteration method for linear eigenvalue problems, SIAM J. Matrix Anal. Appl. 17 (2) (1996) 401–425.
[33] E.R. Davidson, The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, J. Comput. Phys. 17 (1) (1975) 87–94.
[34] Y. Saad, Numerical Methods for Large Eigenvalue Problems, Manchester University Press, Manchester, 1992.
[35] M. Sadkane, R. Sidje, Implementation of a variable block davidson method with deflation for solving large sparse eigenproblems, Numer. Algorithms 20 (2–3) (1999) 217–240.
[36] Netlib repository, https://netlib.org, 2013.
[37] Cuda c programming guide 5.0, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2013.
[38] Kepler tuning guide 1.0, http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html, 2013.
[39] CUBLAS library 5.0, https://developer.nvidia.com/cublas, 2013.
[40] CUDA library 5.0, http://www.nvidia.com/object/cuda, 2013.
[41] V. Kumar, S. Joshi, S. Patkar, H. Narayanan, Fpga based high performance double-precision matrix multiplication, Int. J. Parallel Program. 38 (3–4) (2010) 322–338.
[42] Ö Legeza, T. Röhwedder, R. Schneider, Numerical approaches for high-dimensional PDE's for quantum chemistry, in: Encyclopedia of Applied and Computational Mathematics, Springer, 2013.
[43] K. Marti, M. Reiher, The density matrix renormalization group algorithm in quantum chemistry, Z. Phys. Chem. 224 (2010) 583–599.
[44] G.K.-L. Chan, Low entanglement wavefunctions, Wiley Interdiscip. Rev. Comput. Mol. Sci. 2 (6) (2012) 907–920.
[45] R. Orus, A practical introduction to tensor networks: matrix product states and projected entangled pair states, ArXiv e-prints arXiv:1306.2164.