The Journal of Supercomputing (2025) 81:663
https://doi.org/10.1007/s11227-025-07145-6

tfQMRgpu: a GPU-accelerated linear solver with block-sparse complex result matrix

Paul F. Baumeister[1] · **Stepan Nassyr**[1]

Accepted: 1 March 2025

Abstract

We present `tfQMRgpu`, a GPU-accelerated iterative linear solver based on the transpose-free quasi-minimal residual (tfQMR) method. Designed for large-scale electronic structure calculations, particularly in the context of Korringa–Kohn–Rostoker density functional theory, `tfQMRgpu` efficiently handles block-sparse complex matrices arising from multiple scattering theory. The solver exploits GPU parallelism to accelerate convergence while leveraging memory-efficient sparse storage formats. By unifying the solution of multiple right-hand side (RHS) block vectors, `tfQMRgpu` significantly improves throughput, demonstrating up to a 3.5× speedup on modern GPUs. Additionally, we introduce a flexible implementation framework that supports both explicit matrix-based and matrix-free operator formulations, such as high-order finite-difference stencils for real-space grid-based Green function calculations. Benchmarks on various NVIDIA GPUs demonstrate the solver's efficiency, in some cases achieving over 56% of peak floating-point performance for block-sparse matrix multiplications. `tfQMRgpu` is open-source, providing interfaces for C, C++, Fortran, Julia, and Python, making it a versatile tool for high-performance computing applications that can benefit from the unification of RHS problems.

Keywords Quasi minimal residual method · Iterative linear solver · Header-only library · Block-sparse matrices · Multi-precision library · GPU acceleration

Mathematics Subject Classification 15-04 · 65F10 · 65F50

✉ Paul F. Baumeister
p.baumeister@fz-juelich.de

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany

Published online: 27 March 2025

1 Introduction

In the context of electronic structure calculation and, in particular density functional theory (DFT) [1, 2], linear-scaling algorithms are needed to make geometries with millions of atoms affordable [3–5], however, most methods rely on the representation of a sparse or truncated density matrix which limits their applicability to non-metallic systems. The linear-scaling truncated Green function method has been developed from the Korringa-Kohn-Rostoker (KKR) theory of multiple scattering [6, 7] and works for materials with a band gap as well as metallic systems. The DFT package `KKRnano` [8] shows promising results with respect to the accessible number of atoms and the efficient usage of high-performance computing (HPC) installations. Different from traditional wave function-based DFT methods using eigensolvers, the Green function method requires the solution of many linear systems. Through the concept of screened KKR [9] the non-Hermitian scattering path operator can be constructed in a block-sparse fashion. The block structure stems from KKR's localized basis, i.e. $(\ell_{\max} + 1)^2$ numerical radial basis functions per atom. Often $\ell_{\max}$=3 is used and an additional factor 2 accounts for the non-collinear treatment of magnetism [10, 11] so 16×16 are 32×32 are typical block sizes. We refer to the block size as n. For each atom in the system, n solutions of linear systems $\mathbf{x}_i \in \mathbb{C}^N$ need to be found such that

$$A \cdot \mathbf{x}_i = \mathbf{b}_i, \qquad i \in [1, n] \subset \mathbb{N} \tag{1}$$

for given vectors $\mathbf{b}_i \in \mathbb{C}^N$. The dimension $N = n_r\, n$ is the product of the number of block rows $n_r \in \mathbb{N}$ and the number of rows per block n. Here, $A \in \mathbb{C}^{N \times N}$ is a block-sparse matrix with blocks of size $n \times n$ representing an atom's truncated view of the global scattering path operator. Due to the block structure of A, it is advantageous to group the column vectors $\mathbf{x}_i$ into a dense block vector $X \in \mathbb{C}^{N \times n}$ and equivalently the right-hand side (RHS) vectors $\mathbf{b}_i$ into $B \in \mathbb{C}^{N \times n}$. So now, X needs to be found such that

$$A \cdot X = B. \tag{2}$$

We refer to Eq. (2) as a single atom problem. From the screened KKR theory it follows that only one block of B is nonzero.

The single atom problem in Eq. (2) can be solved using LAPACK's `zgesv`/`cgesv` [12] if we can afford to store a dense matrix representation of A and to pay the compute costs of matrix factorization which scale as $\mathcal{O}(N^3)$. However, the low number of nonzero blocks in A suggests an iterative solving approach, in particular since the truncation radius R_{tr} is increased in order to converge the results and the filling ratio of the sparse matrix A decreases as R_{tr}^{-3}.

As iterative solver we chose a quasi-minimal residual (QMR) method which looks at the residual vector

$$\mathbf{r} = \hat{A}\, \mathbf{x} - \mathbf{b} \tag{3}$$

and seeks to minimize its norm $\|\mathbf{r}\|_2$ by varying $\mathbf{x}$. Here, $\mathbf{r}, \mathbf{x}, \mathbf{b} \in \mathbb{C}^N$ and $\hat{A}$ is a linear operator which could be a general matrix $A \in \mathbb{C}^{N \times N}$. The transpose-free quasi-minimal residual method (tfQMR) has been investigated by Freund and Nachtigal

in the 1990s [13, 14] as modification of the GCS method. It is a variant of the Krylov subspace methods and should therefore exhibit a similar convergence behaviour as other methods out of that algorithm family. The tfQMR method comes with the striking advantage that is suitable for general, non-hermitian operators and no adjoint operator needs to be provided which makes its usage less error-prone to programming mistakes.

1.1 Related work

The original Fortran77 implementations of the tfQMR method by Freund and Nachtigal [13, 15] can be found at `netlib.org/linalg/qmr/*utfx.f`. This work focuses on complex numbers only, so `*` stands for either `c` or `z`. Many implementations of tfQMR can be found in the literature as, e.g. the `Python` implementation `scipy.sparse.linalg.tfqmr` [16], however, most packages act on single vectors like Eq. (1) and cannot benefit to a comparable extent from vectorization, graphical processing unit (GPU) acceleration nor from the performance improvements through the arithmetic intensity of matrix-matrix multiplications. Several software packages offer the treatment of sparse matrices on GPUs [17], some of them specializing for block-sparse matrices [18, 19], while only some libraries offer linear solvers for sparse problems [20]. For example NIVIDA's own `cuSPARSE` [21] which makes use of the block sparse row (BSR) format and offers a GPU version of a direct (non-iterative) linear solver applicable for Eq. (2) and also offers the ingredients for an iterative solver such as the GPU-optimized block matrix times block vector multiplication. Various projects have been realized, e.g. by Cheik Ahamed and Magoulès [22], however, not with a particular focus on block-sparse matrices.

Liegeois *et al.* investigated [23] batched versions of GMRES to combine several smaller problems that fit into a single GPU's memory, however, to our knowledge, none of the established software packages can benefit from combining different atom problems with a partially shared matrix A as explained in the next sections.

The remainder of this paper is structured as follows:

Section 2 describes how the GPU bandwidth can be exploited better by simultaneously solving for several atom problems. Section 3 discusses the performance and speedups on current GPUs. Section 4 shows a matrix-free instantiation of `tfQMRgpu` and compares its performance to the block-sparse use case. Finally, Sect. 5 provides a summary and outlook. Technical details are given in the appendix Section A.

2 Throughput optimization

The iterative linear solver part solving the single atom problem, Eq. (2), has shown to take up more than 90 % of the total runtime of `KKRnano` on CPUs indicating a large potential for GPU-acceleration and optimization. This work shows how `tfQMRgpu` optimizes the throughput of such problems on GPUs by combining the problems of

several atoms. Combining for two or more atoms results in a block-sparse result array breaking the generality of the `tfQMRgpu` as a linear solver.

When considering the particular problem in Eq. (2) for two spatially adjacent atoms we can observe that their respective views of the scattering path operator A_0 and A_1 have a large fraction of matrix blocks that are the same, c.f. orange matrix elements in Fig. 1. Considering that memory bandwidth is the most valuable resource in HPC we can save memory and compute time by solving the two block column problems simultaneously. This requires the unification of the two dense block column vectors X_0 and $X_1 \in \mathbb{C}^{N\times n}$ into an array X. Then X becomes a block-sparse matrix since many but not all n_r block entries in X_0 correspond in their physical position to block entries in X_1, c.f. green block column vector elements in Fig. 1. Note that if we increased the solution array X into a dense matrix, any operator $\hat{A}$ that is not block-diagonal (and hence trivial) would lead to solutions that differ from X_0 and X_1, the separate solutions of Eq. (2) for the two atoms. Therefore, it is essential that X is a block-sparse result array and that any product $Y = A \cdot X$ is restricted to the original sparsity pattern of X. This poses an essential difference to a general multiplication of two block-sparse operators.

For reasons of clarity we discussed combining the atom problems for two spatially adjacent atoms so far. We will, henceforth, assume that $m_c \in \mathbb{N}$ problems are combined. In order to have a large ratio of shared matrix blocks, it is advantageous to unify the problems of a spatially compact cluster of atoms. This is a constraint towards the load balancer of `KKRnano` required to exploit the optimization discussed here.

The core algorithm of tfQMR works for $M = m_c m$ RHS vectors where $m \in \mathbb{N}$ is the number of columns per block and $m_c \in \mathbb{N}$ is the number of atom problems unified, i.e. the number of block columns in the block-sparse solution array, X. It simultaneously minimizes the norm of all M column vectors of the block-sparse matrix of residuals R defined as

$$R = \hat{A}X - B. \tag{4}$$

The algorithm's implementation is `C++` templated with respect to the class `action_t` that offers the action of $\hat{A}$ on a set of M block-sparse trial vectors X as

Fig. 1 Block-sparse matrix A (orange) with 8×8 blocks and block-sparse matrix X (green) with 2 block columns. (Left) In the left column (col#0) of X row#7 is all zero, so the matrix view A_0 has a 7×7 shape ignoring all blocks of A in row#7 and col#7. (Right) For the right column (col#1) row#0 and col#0 of A are ignored to produce a different 7×7 matrix view A_1. In this example from a 1D geometry 17 of 24 blocks of A are shared between the two views, A_0 and A_1. 3D geometries can produce considerably larger fractions of shared blocks

$$Y = \hat{A}.\texttt{multiply}(X). \tag{5}$$

X, Y, R and B are stored in a block-sparse fashion using the BSR format. The BSR format is a version of the compressed sparse row (CSR) sparse matrix format in which each matrix entry is a dense $\mathbb{C}^{n\times m}$ block. In the case of A blocks are square, $\mathbb{C}^{n\times n}$. The BSR format is also used by cuSPARSE [21].

The number of block rows n_r of all four, X, Y, R and B, is determined by the unification of the m_c problems.

Since we need to subtract B from Y to find the residuals R, c.f. Eq. (4), the sparsity pattern of X must have a nonzero block entry wherever B has one, but not vice versa. With other words, B can have less nonzero blocks than Y. In fact, in the particular screened KKR use case only m_c blocks of B are nonzero.

The details of the tfQMR algorithm and the discussion of its convergence behaviour are beyond the scope of this paper and we refer to [13, 15] for the mathematical part. The tfQMR iterations stop when all M RHSs have converged.

3 Application to block-sparse operators

In this section we will present the applications of the tfQMR core algorithm outlined in Sect. 2 with a linear operator $\hat{A}$ that is defined as a block-sparse matrix A, the original use case for which `tfQMRgpu` was developed.

Assume $\hat{A}$ is given as block-sparse matrix A with blocks $a \in \mathbb{C}^{n\times n}$ it is favourable to define the block sizes of both, X and B, to be $\in \mathbb{C}^{n\times m}$ where $m \in \mathbb{N}$ can be chosen freely. Typically, we choose $m = n$, but for the sake of memory alignment, $m > n$ can be advantageous. Then, the action of A on X defined in Eq. (5) can be implemented by an optimized matrix-matrix multiplication kernel that contracts two blocks as

$$y_{ik} = \sum_{j=1}^{n} a_{ij}\, x_{jk} \qquad \text{for } i \in [1, n],\ k \in [1, m]. \tag{6}$$

A result block y is only computed if the corresponding block x is nonzero in the block-sparsity pattern of X. Note that in general the sparsity pattern of Y would have more nonzero blocks than X. However, we restrict this by demanding that the sparsity pattern of Y matches that of X exactly and compute only those blocks of Y.

3.1 Performance

`tfQMRgpu` comes with the executable `bench_tfqmrgpu`, a mini-app for benchmarking. It supports two functionalities

- benchmark the multiplication of two block-sparse operators
- measure the performance of the full block-sparse solver.

The use case of a block-sparse operator A with relatively large blocks could benefit from calling versions of `gemm` from cuBLAS or the batched `gemm` versions for smaller matrices. However, for very small matrix block sizes, kernel overheads are critical. We therefore investigated hand-written block-times-block multiplication kernels. The strategy is to make data reuse as much as possible. Exactly one CUDA-block is started for each nonzero block $y \in \mathbb{C}^{n\times m}$ of the result operator Y. The number of nonzero blocks in Y, `nnzbY`, equals to `nnzbX` since X and Y share the same sparsity pattern. A pre-calculated index list tells which pairs (a, x) with blocks $a \in \mathbb{C}^{n\times n}$ of A and blocks $x \in \mathbb{C}^{n\times m}$ of X contribute to a nonzero block y of Y. The block multiplication kernel is launched with <<< `nnzbY`, $\{m, n/n_{\text{acc}}, 1\}$ >>> and each CUDA-thread zero-initializes n_{acc} complex numbers as accumulators in its registers. The number of accumulators per CUDA-thread, n_{acc}, is a tuning parameter and for simplicity we assume n_{acc}=1 here. We refer to the notation of Eq. (6). In each step j a sub-group of CUDA-threads loads columns of a (length n, stored column-major) and rows of x (length m, stored row-major) into the shared memory of the GPU's streaming multiprocessor (SM)s. In order to save shuffling operations, real parts and imaginary parts are stored separately in memory allowing coalesced load operations. Then, each CUDA-thread loads both, a_{ij} and x_{jk} from shared memory, performs the complex multiplication and adds the result to the accumulators. After each contributing pair of blocks (a, x) has been visited, the content of the accumulators is stored in y.

3.1.1 Multiplication benchmark

As performance benchmark for the block-times-block multiplication, we use sparsity patterns with `nnzbX`= 4490 nonzero blocks in X and `nnzbA`= 13109 nonzero blocks in A. This combination of patterns forsees n_{pairs}= 50526 block multiplications from a KKR use case. The sparsity of A is $13109/1063^2 = 1.16\,\%$ and of X is $4490/(1063 \cdot 16) = 26.4\,\%$. The problem stems from a unification of 16 atom problems. Operator A has in average 13.8 nonzero blocks per row and the solution X has in average 280.6 nonzero block rows per column. For each single RHS column, in average 11.25 nonzero blocks per row in A are relevant.

For a reliable performance benchmark, the kernel is executed 20 times in a row and timings are averaged over five repetitions.

Some performance numbers are listed in Table 1 and the V100 performance is also shown in Fig. 2. Note that the performance is best for blocks shaped 64×64. For small blocks the performance is lower as there, the limitation by the device memory bandwidth is even stronger. For larger blocks the arithmetic intensity is even higher, however, a different implementation of the block-times-block multiplication would be necessary to achieve a higher performance. We would not be able to accommodate all accumulators in a streaming multiprocessor's register file without costly register spills, so that a division into sub-blocks would be required. As sub-blocking would increase code complexity, we recommend to apply other libraries for the multiplication task with larger blocks ($\gg 50$).

Table 1 Performance data in TFlop/sec for the block-times-block multiplication kernel on two different GPUs. The first and third column show the number of rows per block and the number of columns per block

Rows		Cols	V100 `float`	V100 `double`	A100 `float`	A100 `double`
4	×	4	0.510	0.467	0.493	0.453
8	×	8	1.776	1.446	2.566	1.634
16	×	16	4.482	3.067	4.616	3.124
32	×	32	6.742	4.097	9.975	5.455
64	×	64	8.471	4.254	10.740	5.691
128	×	128	6.200	3.012	8.282	3.795
4	×	32	2.778	2.143	4.414	2.768
8	×	32	4.664	2.988	6.067	3.967
16	×	32	5.546	3.498	6.571	4.834

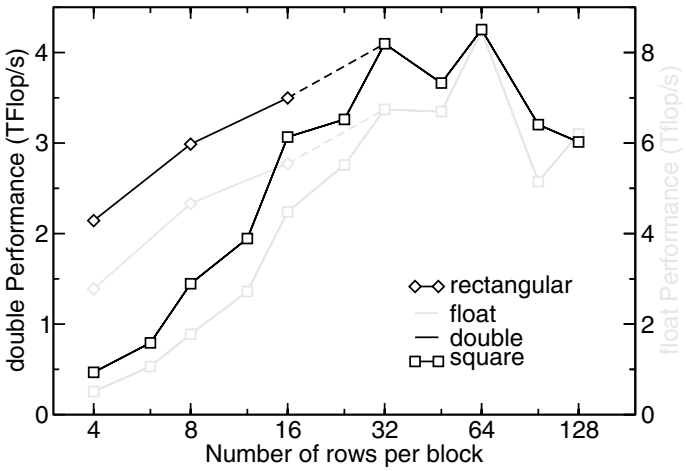

Fig. 2 Performance data for the complex block-times-block multiplication on an NVIDIA V100 GPU. Mind that the scale for `double` performance is on the left axis, i.e. the largest performance here is 4254 GFlop/s (56% of peak `fp64` performance [24]) for 64× 64 blocks. All rectangular cases have 32 columns per block

We also benchmark the case of rectangular blocks, with $m = 32$ columns per block. This matches the warp size of NVIDIA GPUs and benefits from a good ratio of fully coalesced load and store operations. In Fig. 2 an increased performance for the rectangular cases ($m > n$, $m = 32$) can be seen for $n \in \{4, 8, 16\}$ over square block cases.

3.1.2 Benchmark solving

To demonstrate the solving capabilities of **tfQMRgpu** we run the KKR problem from Sect. 3.1.1 with a block size of 32×32. It represents the Matsubara pole

Table 2 Runtime data in seconds for the 32×32-block-sparse `fp64` solver of a KKR problem with 287 rows per column on different GPUs

GPU	1 RHS	16 RHSs	Speedup
V100	0.919 s	8.663 s	1.70
A100	0.940 s	6.119 s	2.46
GH200	0.637 s	2.864 s	3.56

[25] closest to the real axis, i.e. close to the Fermi energy of a metallic system that leads to a moderately large number of iterations due to the large condition number of A. See Fig. 4 for an impression of the spectrum of A and Fig. 3 for the convergence. The tfQMR algorithm usually does not evaluate the residual norm explicitly in every iteration as this requires the potentially costly computation of $AX - B$. We call this *probing*. Typically, probing happens only when approaching the convergence threshold. We forced the algorithm to probe in very iteration to plot the residual norm in Fig. 3. Apparently, probing in every iteration helps to accelerate the convergence slightly in terms of the number of iterations, however, every iteration comes with a 33 % performance penalty, so it is not activated by default. In `double` precision the problem converges to a residual norm below 10^{-6} in 755 iterations requiring about 8.66 s on an NVIDIA V100 GPU. This translates into a full solver performance of 2.355 TFlop/s, i.e. about 57% of the pure multiplication performance (4.1 TFlop/s were achieved on a V100 for 32×32 blocks). The drop in performance is due to the linear algebra operations

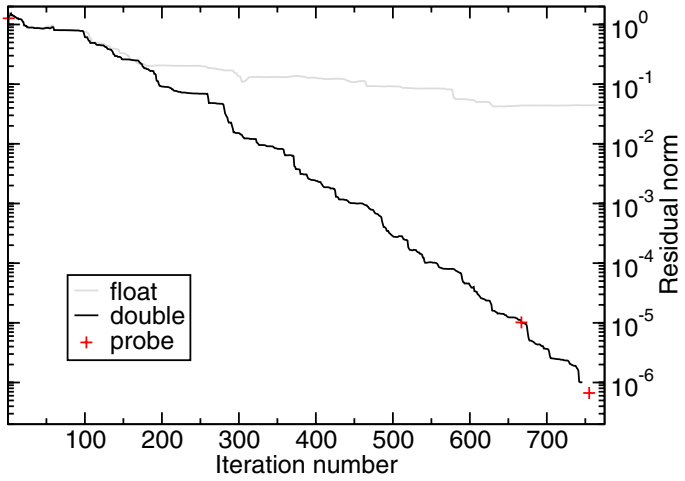

Fig. 3 Convergence of the tfQMR algorithm for a mildly conditioned problem set from KKR multiple scattering theory (condition numbers range from 160 to 655). Solving to a residual norm 10^{-6} took 755 iterations and 8.65 s on a V100. When the residual norm was evaluated in every iteration for this plot, additional effort to compute $AX - B$ required about 33 % more time but convergence was detected nine iterations earlier. Usually, probing occurs rarely, see red crosses. In a mixed precision approach, one could run the first 200 iterations in `float`, however, for this particular problem convergence cannot be reached with single precision

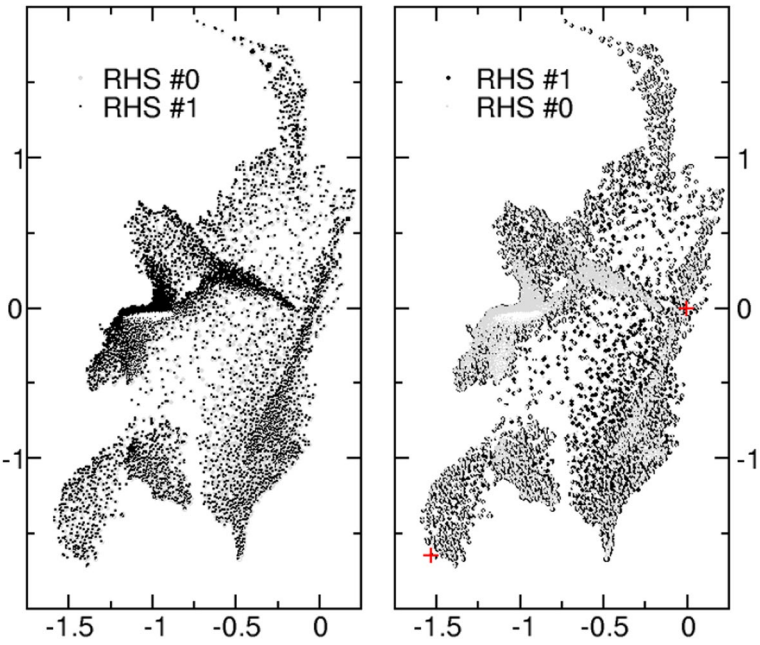

Fig. 4 Eigenvalues of two different views of A from a KKR problem plotted in the complex plane. The spectra were computed with the dense matrix eigenvalue solver `zgeev` from LAPACK. The two red marks in the right subplot indicate the locations of extreme eigenvalues. Although the two spectra have similar features, the condition numbers for these two cases differ by 25 % due to the eigenvalues with the smallest magnitude

(`axpy`, `xpay` and inner products) which require the full device bandwidth but contribute only little to the count of floating point operations.

As visible from Table 2, solving 16 unified problems only takes about 9.4× as long (V100) as solving for a single RHS block column. The latter could also be performed using other libraries. This results in a speedup for the unified problems of 1.7× on a V100. More recent GPUs architectures offer even more floating point performance relative to their GPU memory bandwidth. Here the effect of problem unification becomes even more pronounced with the latest GH200 GPU being up to 3.56× faster than the non-unified reference, see Table 2. This confirms that `tfQMRgpu` can save valuable resources by simultaneously solving the unified problems of several RHS block columns.

The savings can be explained in analogy to the performance characteristics of BLAS routines. As BLAS level 1 operations work on a pair of vectors the arithmetic intensity is low and performance is completely limited by the available device memory bandwidth. BLAS level 2 comprises matrix–vector operations, i.e. some data reuse on the vector can be exploited. Most HPC machines, however, are tuned to maximize their BLAS level 3 performance, i.e. dense matrix multiplications. The arithmetic intensity of dense (square) matrix multiplication grows linear with the matrix dimensions so at a given size, there is enough compute to keep the floating point units busy all the time. We can view the unification of RHS block columns in

a similar way. Although, the block-multiplication requires some floating point operations, the block sizes are too small to saturate the floating point units compared to their bandwidth requirements. When solving for a single RHS block column, the situation is comparable to BLAS level 2. Unification of problems leads to an operation $A \cdot X$, that resembles BLAS level 3 operations more in terms of their data reuse.

4 Implicit operators

The tfQMR core algorithm introduced in the previous sections is templated with respect to a `C++` class `action_t` which as its most important property offers the class method `multiply` performing the operation

$$Y = \hat{A}\, X \tag{7}$$

where $\hat{A}$ needs to be any linear operator that can accept the block-sparse data layout of X and produce Y in the same layout. In Sect. 3 also $\hat{A}$ was a block-sparse matrix operator, i.e. `action_t = blocksparse_action_t`. In this section we will showcase the application of the `tfQMRgpu` header-only library with a matrix-free linear operator.

4.1 Finite-difference derivative

We try to solve for the Green function $\hat{G}_E$ of the 3D Helmholtz equation

$$\left(-\frac{1}{2}\Delta_{\mathbf{r}} - E\right) G_E(\mathbf{r}, \mathbf{r}') = \delta^3(\mathbf{r} - \mathbf{r}') \tag{8}$$

with $E \in \mathbb{C}$ and $\mathbf{r}, \mathbf{r}' \in \mathbb{R}^3$ sampled on a uniform Cartesian real-space grid. Due to the grid sampling, the right-hand side δ^3 becomes a unit operator. The Laplacian $\Delta_{\mathbf{r}}$ is the sum of second derivatives in all three spatial dimensions which, on the grid, can be approximated fairly accurately by a 16th-order finite-difference (FD) stencil [26]. The analytical solution of Eq. (8) is the retarded Green function of the point-shaped wave source

$$G_E(\mathbf{r}, \mathbf{r}') = \frac{\exp(-\imath r\sqrt{2E})}{r} \qquad \text{with} \qquad r = |\mathbf{r} - \mathbf{r}'| \tag{9}$$

which reduces to the electrostatic Green function $1/r$ for E=0 and to the screened Yukawa interaction $\exp(-r\sqrt{-2E})/r$ for negative energies E. For positive energies, the imaginary unit $\imath$ in the exponential function leads to an oscillatory numerator which is relevant for example in acoustics and optics (Helmholtz kernel).

The 3D FD-stencil is implemented in a three-pass procedure, i.e. the second derivative in each spatial dimension is taken in a serial loop [27].

Note that the implementation of the 3D stencil operator is not part of the `tfQMRgpu` repository [28] but can be found at `github.com/real-space/AngstromCube/include/green_kinetic.hxx` for reference.

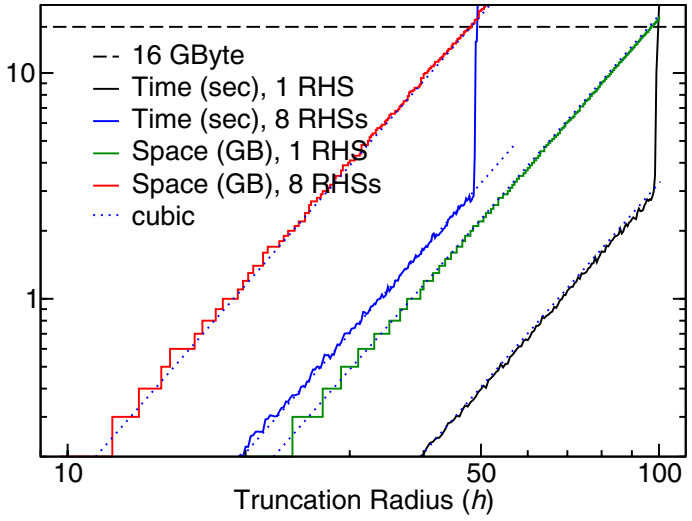

Fig. 5 Time and space requirements for nine tfQMR iterations in single precision as a function of the truncation radius R. R is measured in units of the grid spacing h. The block size 4×4×4 produces steps for small truncation radii. Requirements follow a clear R^3 power law. The curves for 8 RHS blocks are a factor 8 higher and the steep increase in execution time occurs at a factor 2 smaller truncation radius

4.2 Performance

The FD-operator is implemented with a block size 64. Each block corresponds to a cube of 4×4×4 adjacent grid points. At construction time, geometry information is gathered to clarify which cubes are neighbouring to each other. Similar to the KKR use case, an approximate representation of the true solution $\hat{G}_E$ is found by truncating the result Green function at a radius R such that $r < R$ in Eq. (9).

4.2.1 Finite-difference derivative benchmark

We execute exactly nine iterations of the tfQMR algorithm where the FD-operator is called twice per iteration. Each call to the `multiply` routines performs three FD-passes for the three dimensions. In each pass, we need to read the array to be derived and add to the result array (read-write), i.e. the memory needs to be transferred three times. As we can omit the first read of the array, these are 11 instead of 12 operations. In total that means 198 times the memory volume of one Green function array needs to be loaded or stored. Figure 5 shows the timings on a V100 GPU and total memory requirements as functions of the problem size, here given by the cube of the truncation sphere radius, R^3. The truncation radii are given in units of the grid spacing, h. From the fit it seems that only 14.5 % of the nominal maximum device bandwidth (900GByte/sec [24]) is used for the derivatives. Also, when the total memory requirement exceeds the V100 device limit of 16GByte, the unified memory system uses the device memory as a cache and a part of the memory transactions will result in page faults. The computation can still be completed, however, the performance is

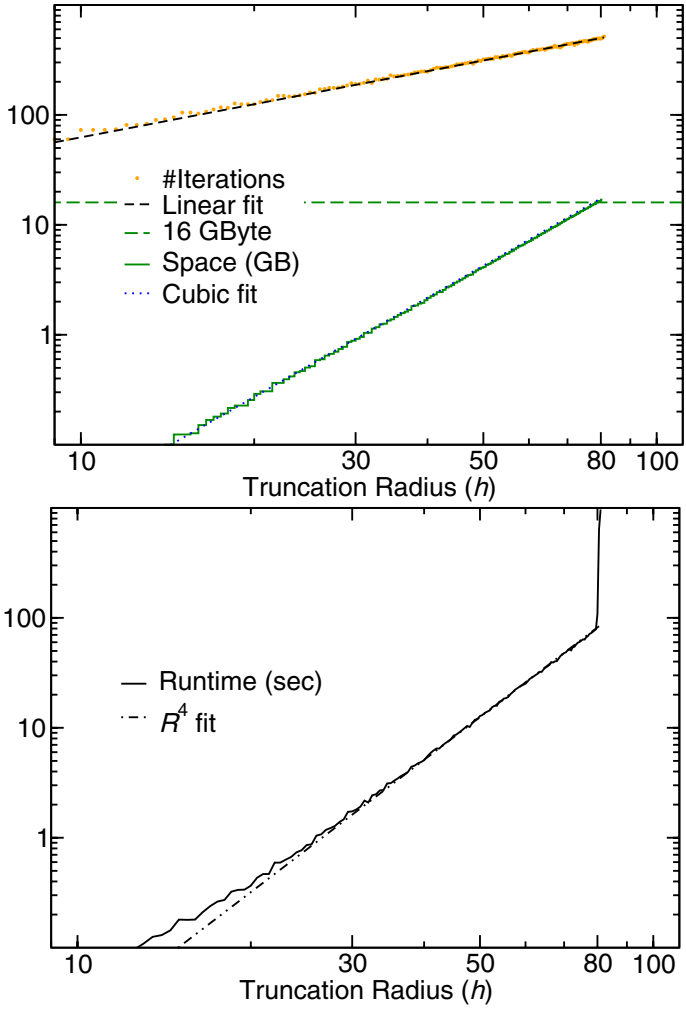

Fig. 6 Space and time requirements for tfQMR solving with an implicit FD-operator in `double` precision as a function of the truncation radius R. The memory requirements follow a clear R^3 power law while the number of iterations increases linearly with R from 60 to about 500. Combined, the runtime scales roughly with R^4. Around R=80, the V100 device memory limit of 16GByte is reached and the execution time skyrockets

reduced by orders of magnitude as the bandwidth towards the host memory becomes a bottleneck. This lets the compute time explode around a truncation radius of 100 grid spacings for a single RHS block and already around 50 grid spacings for 2×2×2 RHS block vectors in single precision, c.f. Fig. 5. Comparing the time needed for a single RHS problem vs. 8 RHSs, we find a factor of 8.0. This indicates that for the implicit operator—as the operator itself does not come with large data arrays—the effect of RHS unification is negligible.

In comparison we can benchmark 18 executions of the FD-operator's `multiply` function alone. Here, we find a V100 bandwidth utilization of 568GByte/sec, i.e. about 77 % efficiency at a truncation radius R=100 h. This means that the tfQMR iterations spend a considerable fraction of their execution time in other bandwidth-limited linear algebra operations. Referring to Fig. 5, the case of a single RHS block vector and R=100 h requires 3.25 s for 9 iterations, but only 0.61 s are needed for 18 operator calls, so the linear algebra fraction accounts for more than 81 % of the execution time.

4.2.2 Benchmark solving

We execute the **tfQMRgpu** solver instantiated with a 64×64 block size and `double` precision complex numbers. The implicit operator is the FD-Laplacian from Eq. (8) with E=0. Figure 6 shows how the number of iterations needed to converge to a residual norm of 10^{-9} as a function of the truncation radius, R. The total GPU memory requirement is 34kByte·$(R/h)^3$. The solving time needed is roughly proportional to R^4 which is a product of the bandwidth-limited kernels that need $\mathcal{O}(R^3)$ time for each iteration and a linear growing number of iterations needed for larger problems (for E=0).

For energy parameters E other than zero, Fig. 7 shows the behaviour of the number of iterations needed until convergence. Negative energies or a positive imaginary part (red line) lead to a sub-linear dependence. In particular for E=−10^{-1}Hartree (dash dotted green line) the solver operates deep inside the Yukawa regime and the number of iterations saturates quickly with problem size. Positive energies, however, exhibit a super-linear behaviour in terms of costs.

4.2.3 Direct performance comparison

tfQMRgpu offers two ways to solve the Helmholtz problem from Eq. (8): We can create a block-sparse operator with FD-coefficients or we can use the implicit FD-operator discussed in the previous section which can avoid completely to load matrix elements of A from device memory.

Here, we used a 16th-order FD-stencil, i.e. there are 8 nonzero coefficients towards each of the six Cartesian directions $\pm x, \pm y, \pm z$ and one central coefficient. Due to its lower arithmetic intensity the implicit FD-operator achieves only 437 GFlop/s, i.e. 5.75% of peak on a V100 GPU. Solving with the equivalent 64×64 block-sparse operator performs about 3TFlop/s but only every 17th matrix entry is nonzero and all matrix elements are real. In a direct performance comparison we find that the implicit FD-operator performs about 3.6× more iterations per seconds. Figure 8 shows how the residuals of both approaches shrink over time. The convergence graphs exhibit slightly different features (comparing the solid red to the dotted blue line) which might be related to differences stemming from summation order artefacts, in particular as in both situations the energy parameter E=0, i.e. we try to solve a nearly singular problem. The truncation radius for this direct comparison was chosen as R=49 grid spacings.

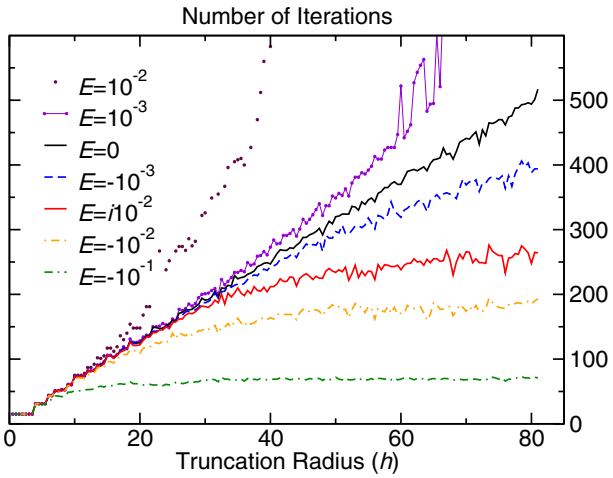

Fig. 7 Number of iterations needed for tfQMR solving Eq. (8) to a residual norm of 10^{-9} as a function of the truncation radius R. The number of iterations depends on the energy parameter E: It is a linear function of R for E=0 (black solid line), it saturates for $E < 0$ (Yukawa regime) and diverges for $E > 0$

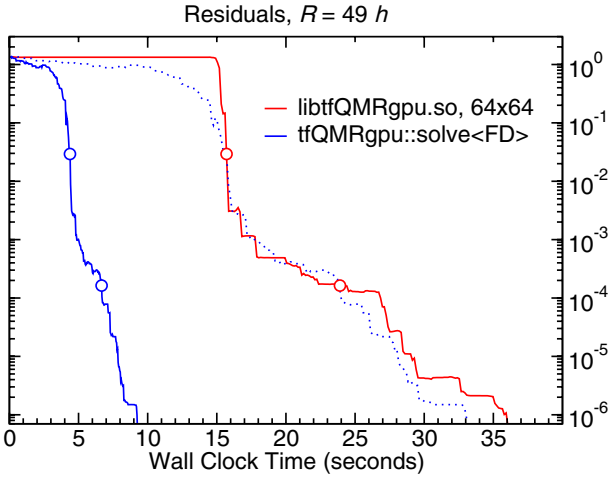

Fig. 8 Convergence of the average residual in the FD-problem with two different approaches: the block-sparse operator with 64×64 blocks as offered by the shared library `libtfQMRgpu.so` (solid red line) takes about 3.6× as long to converge to a similar residual as an *implicit* FD-operator, a stencil derivative action (solid blue line) that makes use of `tfQMRgpu::solve` as a header-only library. The red and blue dots are located where both approaches have the same iteration count (116 and 176, counts are not shown) and a similar residual value. The dotted blue line copies the solid blue line but on a 3.6× -stretched time axis

The tool `tfqmrgpu_generate_FD_example` shipped with **tfQMRgpu** allows to create XML files describing block-sparse operators with FD-coefficients choosing between 1D, 2D and 3D geometries, different block sizes and truncation

radii. The size of the XML input files is kept at a moderate level by storing only the unique stencil blocks of the block-sparse operator A and providing an indirection list. In memory, however, the block-sparse operator is uncompressed into a general form. Hence it consumes a considerable part of the memory capacity and bandwidth requirements. An optimization towards similar stencil operators with constant stencil coefficients would be to incorporate the indirection list into the planning phase of the multiplication of the two block-sparse operators, A and X.

5 Summary and outlook

We present `tfQMRgpu`, a GPU-accelerated library designed to accelerate linear-scaling KKR calculations. It offers iterative solving of linear equations using the tfQMR algorithm. When unifying the linear problems of multiple RHS block vectors, a block-sparse complex-valued solution vector set is required. The performance of GPUs is leveraged to accelerate all operations possible. For the standard use case of block-sparse operators, a shared library `libtfQMRgpu.so` can be built with pre-selected combinations of block sizes when installing the software package. Users may adjust and augment the list of available block sizes according to their needs. Block-sparse matrices with very small blocks (block dimensions up to 50) benefit from hand-written block-times-block multiplication CUDA kernels that can make use of up to 56% of the available floating point performance of NVIDIA V100 and A100 GPUs while fully exploiting their generous bandwidth. The unified linear problems exhibit a large potential of saving resources on the most recent NVIDIA GPUs. Our showcase of 16 unified problems executed more than 3.5× faster on a GH200 GPU compared to 16 single problems. This speedup can be explained by the increased arithmetic intensity of matrix-matrix-multiplication compared to matrix–vector-multiplication, although the vectors are in our case vectors of small dense blocks.

Furthermore, we demonstrated how a custom `CUDA C++` implementation of a linear operator can be fed into the templated algorithm core `tfQMRgpu::solve`. We demonstrated this matrix-free approach with a finite-difference stencil operator which outperforms the equivalent 64×64 block-sparse matrix operator by far. Interfaces to `C`, `Fortran`, `Julia` and `Python` are available for the block-sparse library. Sources, examples, tools and benchmarks are included in the open-source software repository [28] at `github.com/real-space/tfQMRgpu` and the zenodo archive [29].

5.1 Outlook

tfQMRgpu is implemented in CUDA. A generalization would be port to HIP in order to address both, NVIDIA and AMD devices.

A major performance improvement in particular when dealing with ill-conditioned problems would be a proper preconditioner. For the application to KKR problems, a very effective block-circulant preconditioner using fast Fourier transforms

has been tried [30], however such a preconditioner comes at the prices of increased code complexity and inferior parallel scalability. Currently, the algorithm is prepared for preconditioning but a general preconditioner is still missing. Suggestions are welcome.

Since **tfQMRgpu** has been tested with single and `double` precision (`fp64`), mixed-precision recipes could accelerate the convergence. In particular the first iterations of many problems could benefit from being executed in `float` (`fp32`). Furthermore, recent hardware versions of NVIDIA GPUs feature TensorCores to contract small matrices. As also the GPU memory bandwidth has grown with newer hardware generations it would be interesting to see if using TensorCores in the block-sparse operator application could increase the performance even further.

In the case of stencils stored as block-sparse operators, some room for performance improvement is given by compression. If the block-sparse stencils are independent of the row index, as in our finite-difference showcase, an indirection list could reduce the memory capacity requirement of A substantially and increase the performance through higher L2-cache hit rates.

Since none of the performance optimizations for the block-sparse operator or the matrix-free approach are specific to the tfQMR algorithm, it could be worthwhile to try other iterative solving algorithms, such as, e.g. BCGROT, BiCGSTAB, or GMRES, and to compare their convergence behaviours. In particular BCGROT has demonstrated to deliver faster solutions than tfQMR for electronic structure problems [31].

Appendix

Memory usage

The tfQMR algorithm makes use of six additional arrays of the same size as X. Furthermore, `tfQMRgpu` stores an array of random numbers always as `floats`. Since the values are random, precision is not required here. The memory requirement of the RHSs B can range between almost none (B fully determined by the index of the unit vector) up to B and X having the same memory footprint. In total, the memory requirement ranges between 7.5× and 9× the memory requirement of X, including the memory of X itself. As a rule of thumb, about 10× the size of X is safe to assume. Note that this memory count does not yet include any space needed for A.

`tfQMRgpu` offers a memory counting function which should be used before the allocation of a GPU memory buffer. Internally, the buffer is managed avoiding any call to `gpuMalloc`, a macro that maps to `cudaMalloc`.

Interfaces

The block-sparse use case (see Sect. 3) of **tfQMRgpu** is offered as a shared object library `libtfqmrgpu.so`. In order to be useful to applications written in various programming languages the following interfaces are offered.

C Interface

`libtfqmrgpu.so` is shipped with a `C` interface. Before building the library, the file `allowed_block_sizes.h` can be modified to contain the desired block size combinations (n, m) that lead to corresponding `C++` template instantiations at compile time. Furthermore, complex numbers based on single and double precision (`float` and `double`) are supported. See Appendix A.3 for an overview of **tfQMRgpu**'s application programming interface (API).

To lower the adaption barriers the single function APIs `tfqmrgpu_bsrsv_c` and `tfqmrgpu_bsrsv_z` offer library initialization, setup of sparse matrix support structures and solving in one function for single and double precision, respectively. More details can be found in Appendix A.4 and `example/tfqmrgpu_C_example.c` in the repository [28].

C++ Interface

The tfQMR core algorithm is designed as header-only library written in templated `C++` and `CUDA`. This offers the possibility to instantiate the tfQMR core with a user-written `action_t`-class as demonstrated in Sect. 4. However, `C++` applications can also link against the `C`-interface of the precompiled library `libtfqmrgpu.so`.

Fortran Interface

The `C`-API in `tfQMRgpu/include/tfqmrgpu.h` declares all functions

```
tfqmrgpuStatus_t tfqmrgpuNAME(...);
```

to return a scalar integer status variable (`tfqmrgpuStatus_t=int32_t` from `cstdint`). Here, `NAME` is a placeholder for various function names. Please see Appendix A.3 for a complete list of available function names. The wrapper functions defined in `tfQMRgpu/source/tfqmrgpu_Fortran_wrapper.c` augment the `C`-API by declaring `void` functions:

```
void tfqmrgpuNAME_(..., tfqmrgpuStatus_t*);
```

with an appended underscore to the routine name as this matches the naming convention on most Linux/Unix platforms. All scalar arguments are passed by pointer, also the status as a trailing argument. This allows Fortran users to `call` them as `subroutines`, a wide-spread pattern used to call MPI libraries from Fortran [32].

The `void` functions are wrapped again in the Fortran90 `module` defined in `tfQMRgpu/include/tfqmrgpu_Fortran_module.F90`, see `examples/tfqmrgpu_Fortran_example.F90`. Also for Fortran, the single function API is available, c.f. Sects. A.2.1 and A.4 for details.

Table 3 Full API for `tfqmrgpu_bsrsv`

Name	Functionality
`tfqmrgpuCreateHandle`	A library handle is created
`tfqmrgpuSetStream`	Attach a GPU stream to `handle`
`tfqmrgpu_bsrsv_createPlan`	Plan sparse matrix multiplication
`tfqmrgpu_bsrsv_bufferSize`	Compute device memory requirement
`tfqmrgpuCreateWorkspace`	Allocate device memory
`tfqmrgpu_bsrsv_setBuffer`	Attach memory buffer to plan
`tfqmrgpu_bsrsv_setMatrix`	Upload operators A and B
`tfqmrgpu_bsrsv_solve`	Solve the problem
`tfqmrgpu_bsrsv_getInfo`	Check for convergence and stats
`tfqmrgpu_bsrsv_getMatrix`	Download operator X
`tfqmrgpuDestroyWorkspace`	Free device memory
`tfqmrgpu_bsrsv_destroyPlan`	Free `plan` structure
`tfqmrgpuDestroyHandle`	Free library resources
`tfqmrgpuGetErrorString`	Convert status into message
`tfqmrgpuPrintError`	Print error message to standard out
`tfqmrgpuGetStream`	Query GPU stream attached
`tfqmrgpu_bsrsv_allowedBlockSizes`	Query allowed block sizes
`tfqmrgpu_bsrsv_blockSizeMissing`	Check for missing block size
`tfqmrgpu_bsrsv_getBuffer`	Query memory buffer attached

Julia and Python Interface

Functions from the shared object `libtfqmrgpu.so` can be called from `Julia` (julialang.org) and `Python` by adding proper type annotations to the arguments, see `example/tfqmrgpu_Julia_example.jl` and `example/tfqmrgpu_python_example.py`, respectively.

tfQMRgpu block-sparse `C`-API

In Table 3 we present the names of the full API as offered in `include/tfqmrgpu.h` roughly in order as a regular usage could look like. You may refer to `examples` to see them in their function. The goal of this fine-granular 13 steps or more is to avoid redundant operations when integrating **tfQMRgpu** into your application. The API design has been chosen to offer extensibility, i.e. allowing for additional library functionality besides `bsrsv`. Although the creation of the library handle is currently very light-weight you may want to reduce the overheads that arise from performing the finalization and initialization steps in between two `solve`-calls. For example in the application `KKRnano` the shapes of the block-sparse matrices X and B only change when atom positions or the truncation radius are changed or the block columns are redistributed among the parallel processes. Therefore, the analysis step `createPlan` is separated from the solve process as the `plan` stays

unchanged. Similarly, if also the block dimensions stay the same, the `bufferSize` function will compute the same memory requirement and we can even avoid to call `DestroyWorkspace` (=`cudaFree`) and `CreateWorkspace`(=`cudaMalloc`) in between two solver calls. In some applications, only A is changed, in other use cases, only B might change. Therefore, the user can choose independently when to upload the data arrays of operators A and B using the `setMatrix` function. Similarly, we can avoid to download the operator X using the `getMatrix` function when the `getInfo` function signals that convergence has not been reached. The lower part of Table 3 lists some helper functions that allow to deal with error codes produced by **tfQMRgpu** and inquiry functions. The most important ones of them being `allowedBlockSizes` and `blockSizeMissing`. These two interfaces enable the user to check at runtime if a given pair of block dimensions (n, m) was listed in `include/allow_block_sizes.h` during compile time. It could be helpful to avoid the wasting of compute time on HPC machines when integrated into the initialization steps of an application.

Easy integration helper

The full API described in the previous section is meant to be integrated into various parts of an application which means a substantial programming effort. However, before integrating, the users need to assess if **tfQMRgpu** is beneficial for their application. For this, a single function API is offered:

```
int tfqmrgpu_bsrsv_z(int nRows, int ldA, int ldB,
    const int32_t *rowPtrA, int nnzbA, const int32_t *colIndA,
    const double  *Amat,  char transA,
    const int32_t *rowPtrX, int nnzbX, const int32_t *colIndX,
          double  *Xmat,  char transX,
    const int32_t *rowPtrB, int nnzbB, const int32_t *colIndB,
    const double  *Bmat,  char transB,
    int32_t *iterations, float *residual, int echo);
```

The meaning of these arguments is explained in detail in `docs/tfQMRgpu_manual`, however, for quick reference we present explanations in Table 4. Similarly, the single precision version `tfqmrgpu_bsrsv_c` is offered with `Amat`, `Xmat` and `Bmat` being `float`-pointers instead of `double`-pointers.

The integer type `int32_t` has been chosen as it matches with `INTEGER(kind=4)` in Fortran, a reasonable tradeoff between range ($[-2^{31}, 2^{31}-1]$) and data volume (4Byte per number). Although all values passed in these lists are non-negative, Fortran does not support `unsigned` integer types natively, hence we only use the range $[0, 2^{31}-1]$. Internally, **tfQMRgpu** uses `uint16_t` for the block column indices which limits their number to $2^{16} = 65\,536$.

The pointers to `doubles` mark the beginning of complex arrays. **tfQMRgpu** expects real parts and imaginary parts back to back in memory,

Table 4 Arguments of the single function API `tfqmrgpu_bsrsv_z`. The character '?' stands for `A`, `X` or `B` in the argument names. The column "rw" marks if the fields are read, written or both. # is short for "number of"

Type	Name	rw	Functionality
`int`	`nRows`	r	# block rows in A, X, B and # block columns in A
`int`	`ldA`	r	# columns per block and # rows per block in A
`int`	`ldB`	r	# columns per block in B and X
`int32_t*`	`rowPtr?`	r	list of row starts in the BSR format, layout `[nRows+1]`
`int`	`nnzb?`	r	# nonzero blocks of A, X, B, respectively
`int32_t*`	`colInd?`	r	List of column indices of the blocks, layout `[nnzb?]`
`char`	`trans?`	r	Transposition of blocks of A, X, B, respectively
`double*`	`Amat`	r	Matrix entries of A, layout `[nnzbA` × `ldA` × `ldA` × `2]`
`double*`	`Xmat`	w	Matrix entries of X, layout `[nnzbX` × `ldA` × `ldB` × `2]`
`double*`	`Bmat`	r	Matrix entries of B, layout `[nnzbB` × `ldA` × `ldB` × `2]`
`int32_t*`	`iterations`	rw	In: maximum # iterations, out: # iterations needed
`float*`	`residual`	rw	In: threshold for convergence, out: residuum reached
`int`	`echo`	r	Verbosity level for logging, 0: no output, ..., 9: debug
`int`	(returned)	w	`tfqmrgpu_bsrsv_z` returns a status, 0: no errors

i.e. $\{\{r_0, i_0\}, \{r_1, i_1\}, \dots\}$. Since the `C` standard does not offer an equivalent of `std::complex` the user should pass the pointer to the real part of the first complex number.

The data layouts are given for block transpositions `trans?='n'` (non-transpose). With `trans?='t'`, block dimensions and indices are interchanged. We refer to the manual for more options such as complex conjugation.

The number of block columns in X and B is derived from the indices in `colIndX`.

Hardware and middleware details

The compiler versions used for benchmarking were `GCC/9.3.0` and `CUDA/11.0` in a `RockyLinux/8.7` operating system. The host processor to the NVIDIA V100 GPUs was an AMD EPYC 7742 with 256 GByte of DDR4 memory (JSC system "JUSUF"). For the benchmarks on NVIDIA A100 an AMD EPYC 7402 host CPU (JSC system "JUWELS_Booster") was used.

Open source

`tfQMRgpu` is publicly available under the MIT license at `github.com/real-space/tfQMRgpu`. This publication has been produced using the version tagged `v0.9` [28].

Acknowledgements PFB thanks Jiri Kraus (NVIDIA) for support on the interface design. Also, PFB thanks Marc Vandelle (EPFL) for a first version of the `Python` interface.

Funding Open Access funding enabled and organized by Projekt DEAL. This work has been funded by the German Federal Ministry of Education and Research through SiVeGCS.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Hohenberg P, Kohn W (1964) Inhomogeneous electron gas. Phys Rev 136:864–871. https://doi.org/10.1103/PhysRev.136.B864
2. Kohn W, Sham LJ (1965) Self-consistent equations including exchange and correlation effects. Phys Rev 140:1133–1138. https://doi.org/10.1103/PhysRev.140.A1133
3. Mohr S, Ratcliff LE, Genovese L, Caliste D, Boulanger P, Goedecker S, Deutsch T (2015) Accurate and efficient linear scaling dft calculations with universal applicability. Phys Chem Chem Phys 17:31360–31370. https://doi.org/10.1039/C5CP00437C
4. Nakata A, Baker JS, Mujahed SY, Poulton JTL, Arapan S, Lin J, Raza Z, Yadav S, Truflandier L, Miyazaki T, Bowler DR (2020) Large scale and linear scaling dft with the conquest code. J Chem Phys 152(16):164112. https://doi.org/10.1063/5.0005074
5. VandeVondele J, Borštnik U, Hutter J (2012) Linear scaling self-consistent field calculations with millions of atoms in the condensed phase. J Chem Theory Comput 8(10):3565–3573. https://doi.org/10.1021/ct200897x
6. Korringa J (1947) On the calculation of the energy of a Bloch wave in a metal. Physica 13(6):392–400. https://doi.org/10.1016/0031-8914(47)90013-X
7. Kohn W, Rostoker N (1954) Solution of the Schrödinger equation in periodic lattices with an application to metallic lithium. Phys Rev 94:1111–1120. https://doi.org/10.1103/PhysRev.94.1111
8. Thiess A, Zeller R, Bolten M, Dederichs PH, Blügel S (2012) Massively parallel density functional calculations for thousands of atoms: KKRnano. Phys Rev B 85:235103. https://doi.org/10.1103/PhysRevB.85.235103
9. Zeller R, Dederichs PH, Újfalussy B, Szunyogh L, Weinberger P (1995) Theory and convergence properties of the screened Korringa-Kohn-Rostoker method. Phys Rev B 52:8807–8812. https://doi.org/10.1103/PhysRevB.52.8807
10. JuKKR Repository. Forschungszentrum Jülich GmbH. https://iffgit.fz-juelich.de/kkr/jukkr Accessed 09 Sept 2023
11. Bornemann M, Grytsiuk S, Baumeister PF, Santos Dias M, Zeller R, Lounis S, Blügel S (2019) Complex magnetism of B20-MnGe: from spin-spirals, hedgehogs to monopoles. J Phys Condens Matter 31(48):485801. https://doi.org/10.1088/1361-648X/ab38a0
12. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898719604

13. Freund RW (1993) A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. SIAM J Sci Comput 14(2):470–482. https://doi.org/10.1137/0914029
14. Freund RW, Nachtigal NM (1991) QMR: a quasi-minimal residual method for non-Hermitian linear systems. Numerische Math 60(1):315–339. https://doi.org/10.1007/BF01385726
15. Freund RW, Nachtigal NM (1996) QMRPACK: a package of QMR algorithms. ACM Trans Math Softw 22(1):46–77. https://doi.org/10.1145/225545.225551
16. Kelley CT (1995) Iterative methods for linear and nonlinear equations. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. https://doi.org/10.1137/1.9781611970944
17. Anzt H, Cojean T, Flegar G, Göbel F, Grützmacher T, Nayak P, Ribizel T, Tsai YM, Quintana-Ortí ES (2022) Ginkgo: a modern linear operator algebra framework for high performance computing. ACM Trans Math Softw. https://doi.org/10.1145/3480935
18. Borstnik U, VandeVondele J, Weber V, Hutter J (2014) Sparse matrix multiplication: the distributed block-compressed sparse row library. Parallel Comput 40(5–6):47–58
19. OpenAI Blocksparse. GitHub. https://cdn.openai.com/blocksparse/blocksparsepaper.pdf
20. cuSOLVER v12.8 (2025). https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf
21. cuSPARSE, the CUDA sparse matrix library v12.8 (2025). https://docs.nvidia.com/cuda/cusparse/
22. Cheik Ahamed A-K, Magoulès F (2012) Iterative methods for sparse linear systems on graphics processing unit. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, 836–842. https://doi.org/10.1109/HPCC.2012.118
23. Liegeois K, Rajamanickam S, Berger-Vergiat L (2023) Performance portable batched sparse linear solvers. IEEE Trans Parallel Distrib Syst 34(5):1524–1535. https://doi.org/10.1109/TPDS.2023.3249110
24. NVIDIA-Corporation: (2017) NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU. Technical report. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
25. Matsubara T (1955) A new approach to quantum-statistical mechanics. Progr Theor Phys 14(4):351–378. https://doi.org/10.1143/PTP.14.351
26. Keller H, Pereyra V (1978) Symbolic generation of finite difference formulas. Math Comput 32(144):955–971. https://doi.org/10.1090/S0025-5718-1978-0494848-1
27. Baumeister PF, Hater T, Pleiter D, Boettiger H, Maurer T, Brunheroto JR (2017) Exploiting in-memory processing capabilities for density functional theory applications. In: Euro-Par 2016: Parallel Processing Workshops. Lecture Notes in Computer Science, vol 10104, pp 750–762. Springer, Cham. Chap. 60. https://doi.org/10.1007/978-3-319-58943-5_60. https://juser.fz-juelich.de/record/830547
28. Baumeister PF (2023) tfQMRgpu GitHub respository. https://github.com/real-space/tfQMRgpu Accessed 09 Sept 2023
29. Baumeister P, Nassyr S (2023) Real-space/tfQMRgpu: stable for reference publication. Zenodo. https://doi.org/10.5281/zenodo.8333498
30. Bolten M, Thiess A, Yavneh I, Zeller R (2012) Preconditioning systems arising from the kkr green function method using block-circulant matrices. Linear Algebra Appl 436(2):436–446. https://doi.org/10.1016/j.laa.2011.05.019
31. Yu R, Sturler E, Johnson DD (2002) A block iterative solver for complex non-hermitian systems applied to large-scale, electronic-structure calculations. Technical report, USA. https://dl.acm.org/doi/10.5555/871118
32. MPI-Forum: MPI (1994) A message-passing interface standard. Technical report, USA. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf Accessed 09 Sept 2023