Integrating batched sparse iterative solvers for the collision operator in fusion plasma simulations on GPUs

Aditya Kashi [a,d,*], Pratik Nayak [a], Dhruva Kulkarni [c], Aaron Scheinberg [e], Paul Lin [c], Hartwig Anzt [a,b]

[a] *Karlsruhe Institute of Technology, Karlsruhe, Germany*
[b] *University of Tennessee, Knoxville, TN, USA*
[c] *Lawrence Berkeley National Laboratory, Berkeley, CA, USA*
[d] *Oak Ridge National Laboratory, Oak Ridge, TN, USA*
[e] *Jubilee Development, Cambridge, MA, USA*

Keywords:
Sparse linear systems
Batched solvers
Plasma simulation
GPU
Performance portability

A B S T R A C T

Batched linear solvers, which solve many small related but independent problems, are increasingly important for highly parallel processors such as graphics processing units (GPUs). GPUs need a substantial amount of work to keep them operating efficiently and it is not an option to solve smaller problems one-by-one. Because of the small size of each problem, the task of implementing a parallel partitioning scheme and mapping the problem to hardware is not trivial. In recent history, significant attention has been given to batched dense linear algebra. However, there is also an interest in utilizing sparse iterative solvers in a batched form.

An example use case is found in a gyrokinetic Particle-In-Cell (PIC) code used for modeling magnetically confined fusion plasma devices. The collision operator has been identified as a bottleneck, and a proxy app has been created for facilitating optimizations and porting to GPUs. The current collision kernel linear solver does not run on the GPU—a major bottleneck. As these matrices are sparse and well-conditioned, batched iterative sparse solvers are an attractive option.

A batched sparse iterative solver capability has recently been developed in the Ginkgo library. In this paper, we describe how Ginkgo's batched solver technology can integrate into the XGC collision kernel and accelerate the simulation process. Comparisons for the solve times on NVIDIA V100 and A100 GPUs and AMD MI100 GPUs with one dual-socket Intel Xeon Skylake CPU node with 40 cores are presented for matrices from the collision kernel of XGC. Further, the speedups observed for the overall collision kernel are presented in comparison to different modern CPUs on multiple supercomputer systems. The results suggest that Ginkgo's batched sparse iterative solvers are well suited for efficient utilization of the GPU for this problem, and the performance portability of Ginkgo in conjunction with Kokkos (used within XGC as the heterogeneous programming model) allows seamless execution on exascale-oriented heterogeneous architectures.

1. Introduction

Research on magnetically confined fusion plasmas, e.g. the International Tokamak Experimental Reactor (ITER), operate in a parameter space that is currently inaccessible to experiment. Design choices are therefore driven by high fidelity numerical simulations that require exascale computing capabilities. Many of the current pre-exascale computing architectures as well as the three upcoming US Department of Energy (DOE) exascale computing architectures are heterogeneous and incorporate both CPUs and GPUs. As typically 80%-90% of the peak performance of these platforms is in the GPUs, it is critical to efficiently exploit the GPUs to accelerate hotspots in the simulation codes. One such simulation is the WDMApp project, which aims to model the plasma in the entire fusion device. Different application codes in WDMApp are used to simulate the plasma depending on the location within the device [9] – the gyrokinetic Particle-In-Cell (PIC) XGC code is optimized for modeling the plasma close to the edge.

XGC implements a non-linear Fokker-Planck-Landau collision operator on a two-dimensional velocity grid capable of simulat-

* Corresponding author.
E-mail address: kashia@ornl.gov (A. Kashi).

ing multiple species of particles in a plasma (ions, electrons) [15]. The 'collision' step – describing Coulomb collisions between particles in the plasma – has been identified as a bottleneck in XGC. A proxy app that only computes the collision kernel rather than the entire simulation has been created for facilitating optimizations and porting to GPUs. Currently, the collision kernel utilizes MPI for multiple CPU nodes, and Kokkos [12] to offload to GPUs as well as utilize OpenMP for intra-node parallelism.

Within the collision kernel, a linear solver is employed in a Picard iteration. This linear solver is the only remaining part of the collision kernel not yet ported to GPUs. As of today, the banded direct solver 'dgbsv' from LAPACK is used to solve this system on the CPU. As these matrices are sparse with low condition numbers, sparse iterative solvers are a viable option. Batching ensures that the GPU is utilized fully and also fits well within the batching scheme of the collision kernel (batching over spatial mesh nodes). Thus, batched sparse iterative solvers are an attractive option for the XGC collision kernel solver.

Fine-grain parallel implementation of batched sparse iterative solvers is challenging for several reasons. GPUs have a hierarchy of memories, with different bandwidths and access latencies, and a hierarchy of compute cores with different communication mechanisms; this makes batched solver implementation complex. Different types of problems may need different sparse storage formats and different algorithms for solver components, while different optimizations are needed for different sizes of problems. Additionally, different systems within a batch may converge at different rates. Thus, along with efficient algorithms, flexibility is required in the software architecture for iterative solvers. This has to be coupled with management of kernel launch overhead and efficient use of the memory hierarchy.

To this end, we list our key contributions:

1. We develop batched sparse matrix vector kernels for two batch matrix formats, **`BatchCsr`** and **`BatchEll`** for NVIDIA GPUs and AMD GPUs.
2. We integrate these sparse matrix kernels along with specialized, tuned **`BatchDense`** kernels to construct batched iterative solvers and show results for the BiCGSTAB Krylov subspace solver [22] with a Jacobi preconditioner.
3. We tune the batched BiCGSTAB solver for the matrices from XGC and also provide an automatic tuning strategy depending on the size of the matrix.
4. We analyze the performance using NVIDIA Nsight Compute and AMD rocprof and present the performance achieved in the context of the theoretical peak of the GPU.
5. We provide a production-ready implementation of the batched iterative solver functionality within Ginkgo [7], which is readily available for applications along with examples.
6. We discuss and demonstrate how Ginkgo batched solvers are integrated into XGC.
7. We evaluate the performance of the integration and show that it incurs little to no performance overhead.

We had earlier presented the initial part of this research effort [16]. This work is an extension, with advances including details of integration between Ginkgo and XGC, as well as performance evaluations with the complete collision kernel within XGC. Thus, the final two points above represent the extensions.

In section 2, we describe the factors that demonstrate the need for high-performance batched linear solver for the XGC collision kernel. Existing literature and work on batched solvers and batched routines are explored in section 3. A brief overview of Ginkgo's batched capabilities, and algorithmic and other optimizations to improve its performance on the collision kernel, are described in section 4. The integration with Kokkos and XGC is also discussed. Comparisons for the batched solve times on NVIDIA V100, A100 and AMD MI100 GPUs with one dual-socket Intel Xeon Skylake CPU compute node with 40 OpenMP threads are presented in section 5. In addition, the performance of the entire collision kernel including all CPU and GPU operations is also presented and discussed on three different platforms containing the three different GPUs.

2. Motivation and background

2.1. XGC *proxy app*

XGC is a 5D full-function gyrokinetic particle-in-cell (PIC) application code that numerically simulates fusion edge plasmas. A nonlinear collision operator is required to accurately model edge plasmas. For this, XGC employs a nonlinear Fokker-Planck-Landau operator in the 2D guiding-center velocity space for multiple particle species. An implicit time integration method is employed and a Picard method for the nonlinear solver. At each configuration space grid node, the nonlinear operator is solved on the 2D velocity space grid. For details, the reader is referred to the publications on XGC development [23,15]. Production simulations currently employ the LAPACK banded solver **`dgbsv`** on the CPU for the linear solve.

A proxy app for the collision kernel has been developed using Kokkos for providing a performance portable layer for GPU offloading. The proxy app is parallelized over spatial mesh nodes and is embarrassingly parallel. While the future XGC application is expected to simulate multiple ion species (~10) and electrons, the proxy app currently simulates a plasma with one ion species (along with electrons). A backward Euler time discretization and Picard iteration are employed for the two species for every mesh node. The Picard loop typically requires five iterations for convergence. Fig. 1 shows the execution timeline of one such Picard iteration captured on one MPI rank with multiple OpenMP threads on the CPU and one GPU - the top half of the figure (black rectangles) shows CPU execution of the linear solver employed in the Picard iteration and associated processing, while the bottom half shows GPU execution (blue rectangles) as well as data transfer (red and green rectangles). As seen in Fig. 1, a significant portion of the execution time for the Picard loop (~48%) is on the CPU - of which a majority of the time is spent in the solve (**`dgbsv`** call) itself (~66%). In addition, data on the GPU needs to be transferred back and forth between the GPU and the CPU to employ a linear solver on the CPU - which causes additional overhead (~9%) and would limit the possibility of exploiting direct GPU - GPU memory transfer in the future. Thus, XGC would greatly benefit from porting the solver to the GPU.

The matrix dimensions utilized in the main application are on the order of 10^3 and possess a sparsity pattern arising from the use of a nine-point stencil (9 non-zero elements per row). For these dimensions and bandwidth, using dense solvers on the GPU is not enough to beat the gain obtained from exploiting the banded nature of the matrix on the CPU. Thus, sparse solvers on the GPU are required and need to be batched to fully saturate the GPU. Further, as the XGC matrices exhibit a low condition number, iterative batched sparse solvers may prove to be most efficient for utilizing the GPU for these types of problems.

We see in Fig. 2 that the matrices for the ion and the electron have quite different eigenvalue distributions. For ions, the eigenvalues are more or less clustered around 1.0 (note the log real axis) which will likely lead to very quick convergence, while electrons have a greater range of real parts of the eigenvalues which means it will likely take some more iterations to converge. That being said, they are both well-conditioned enough to take good advan-

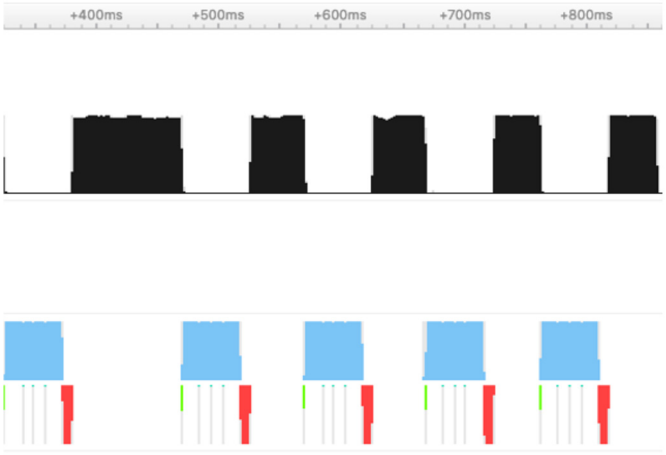

Fig. 1. Profile of one Picard loop of the collision kernel proxy app showing time spent on CPU (black), GPU (blue), and memory transfer (red: Device to Host, green: Host to Device). The heights of the bars refer to the utilization of the relevant hardware - CPU, GPU or PCIe bus for host-device transfers, while the *x*-axis is application run time.

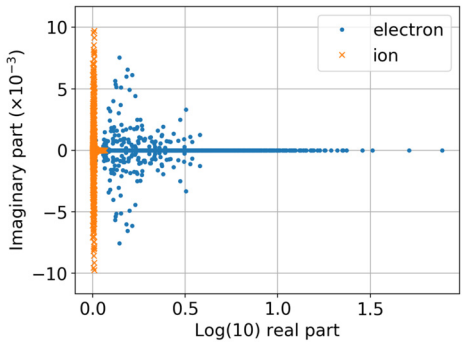

Fig. 2. Eigenvalues of the matrices for the two species.

tage of iterative solvers as neither of them has very large or very small eigenvalues.

One solution for solving a batch of small sparse problems would be to assemble them into a block-diagonal matrix with sparse diagonal blocks. The resulting larger system may have a size suitable to utilize the entire device. However, in such a workflow, the number of iterations over the entire block system would be determined by the most difficult diagonal problem. Global synchronization points would be introduced in the iterations, leading to greater synchronization overheads. Further, even if the problems have a common sparsity pattern, the global sparse matrix format would require duplication of the pattern for every block. While this is not discussed further in this paper, experiments have revealed that such a method is slower than the proposed batched iterative solvers.

2.2. *The* Ginkgo *library*

Ginkgo is a high performance numerical linear algebra library aiming to provide highly tuned performance portable numerical linear algebra algorithms. In particular, Ginkgo focuses on sparse data structures and algorithms and provides building blocks such as SpMV, SpGEMM etc and uses them for high-performant linear solvers and preconditioners.

Ginkgo is written in modern C++ and is designed with software sustainability and reproducibility in mind. Run-time polymorphism is used to dispatch kernels that have been tuned for specific hardware. Currently, Ginkgo has support for multiple GPU backends with CUDA, HIP and DPC++ and also supports multi-core CPU architectures with OpenMP [6,7].

Support for batched functionality within Ginkgo has been recently added [2] and is being expanded to incorporate more matrix formats, solvers and preconditioners.[1]

3. Related work

With the advent of GPU architectures providing fine-grained parallelism, efforts started in developing data-parallel algorithms that provide BLAS and LAPACK functionality in batched fashion. Recently, a batched BLAS interface was also proposed [11,10] to allow the vendors and the library providers to implement a uniform function interface for the batched functionality. This set of batched routines has also been expanded to LAPACK [1].

Beyond BLAS and LAPACK functionality, there has also been some work in the direction of batched dense inversions for Block-Jacobi preconditioning [5] and for solving small dense problems in batched mode [13].

While batched dense linear algebra has received attention in recent years, less efforts have focused on developing batched functionality for sparse and iterative linear algebra. In terms of batched sparse direct solvers, there has been some work on tridiagonal and pentadiagonal systems [21,14,8], and NVIDIA cuSPARSE provides the `gtsv2StridedBatch` routine based on variants of cyclic reduction.

These methods aim to specifically solve tri-diagonal and penta-diagonal systems, and rely on algorithms based on Thomas' algorithm or cyclic reduction. Further, the solve stages are not fine-grain parallel, but each GPU thread solves an entire linear system. In this case performance benefits primarily come from storing the problem data in interleaved fashion to allow for coalesced access. In the context of the Human Brain Project, where one needs to solve multiple of Hines systems, Valero-Lara et al. [20] proposed and implemented methods similar to *cuThomasBatch* [21] to accelerate the solution of batched Hines systems for NVIDIA GPUs. While such a scheme is certainly robust and has advantages for certain applications, it does not provide the best performance when the exact solution (relative to machine precision) is not required and the problem is relatively well-conditioned. In the case of work that includes pentadiagonal systems [14,8], the factorization step is performed on the CPU, necessitating data transfer for the triangular solves.

For general sparse matrices in compressed sparse row (CSR) format, a batched sparse QR factorization and solve is available in Nvidia's cuSOLVER library [17]. The general dogma has been that with relatively small linear systems, direct solvers are more effective. While this is true for some cases, the flexibility provided by the iterative solvers in terms of early stopping, re-use of initial guess and adaptability to matrix properties can make them very attractive even for relatively small problems.

Recently, efforts began towards developing a batched sparse iterative linear solver capability for combustion simulations [3]. That work described the basic idea of batched sparse iterative solvers, and presented a capability limited to the small matrices encountered in the combustion simulations in question. While significantly better performance compared to dense direct solvers was shown, it was not suitable for the medium-sized structured matrices of interest in the XGC collision kernel. In the present article, we describe how flexibility and performance can further be enhanced to handle this case.

[1] See https://github.com/ginkgo-project/ginkgo/blob/batch-develop/examples/batched-solver/batched-solver.cpp for a usage example.

4. Implementation of batched iterative solvers

In this section, we elaborate on the implementation of the batched iterative solvers in Ginkgo. We motivate our design choices, explore sparse matrix formats suitable for the matrices involved in the collision kernel and showcase the optimizations necessary to fully utilize the resources available. As we mainly concentrate on GPU implementations, we discuss terms in the context of GPU programming (CUDA/ ROCm), but most of these ideas carry over to the hierarchical memory multi-core CPU architectures.

The objective of a batched solver interface is to utilize the embarrassing parallelism provided by the problem to the highest extent possible, while taking advantage of fine-grained parallelism to solve the individual systems. Batched solvers are favorable in cases where the individual matrices are small, and large numbers of these linear systems need to be solved. The criteria that influence the design and implementation are the following:

1. The size of the individual batch entries: the number of rows and number of non-zeros.
2. The number of linear systems to be solved.
3. Common sparsity patterns between the batched matrices, if any.
4. Properties of the batched linear systems that influence convergence (condition numbers, etc.)

On the other hand, the following performance considerations influence the design and implementation:

1. Keeping data as close as possible to the GPU compute units. For batched problems, the data corresponding to individual systems may be small enough to persistently store in higher levels of memory, such as local shared memory and L1 cache, which have much lower latency.
2. To a lesser extent, kernel launch overhead needs to be managed when dealing with small individual problems.

4.1. Batch matrix storage formats

Sparse matrices typically store an array of non-zero values, as well as integer arrays encoding the sparsity pattern. If all the matrices in the batch share a common sparsity pattern, one can store the sparsity pattern only once, while storing the values of all the entries. This reduces memory requirements and the data transfer volume. Batched linear systems may share a sparsity pattern e.g. if these linear systems arise from similar local physics at many grid points or if one wants to solve multiple independent problems that all employ the same discretization and mesh. To this end, we implement two batch matrix formats, one general **`BatchCsr`**, and one specialized **`BatchEll`**.

The **`BatchCsr`** matrix format is based on the popular Compressed Sparse Row matrix storage format, where one stores an array of column indices per row corresponding to each non-zero value in the matrix. An accumulated sum of the number of non-zeros per row is additionally necessary. This matrix format is suitable for general matrices with large variations in the number of non-zeros per row and performs generally well for most matrices. The **`BatchCsr`** is an extension of this format where we store the column indices and the row pointers for only one matrix and store the values of all the matrices.

For matrices that have a similar number of non-zeros in every row, we can optimize the storage by padding the rows to a uniform number of non-zeros per row, removing the need for a pointers array. This also gives us additional advantages in terms of coalesced accesses. The **`BatchEll`** matrix format stores one set of column indices and the values of all the batch entries. In contrast to **`BatchCsr`**, we store the column indices and the values in column-major allowing for coalesced accesses which is suitable for GPUs.

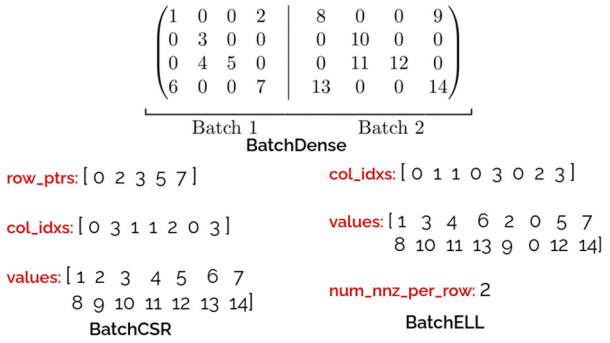

Fig. 3. Batch Matrix Storage formats - **`BatchDense`**, **`BatchCsr`** and **`BatchEll`**.

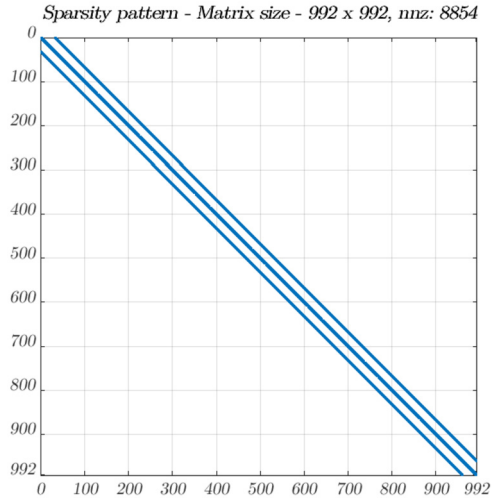

Fig. 4. Sparsity pattern of an individual entry of the batched matrix: 992 rows, 9 nonzeros per row.

Fig. 3 visualizes the schematic and the storage requirements of **`BatchCsr`** and **`BatchEll`** compared to the **`BatchDense`** format. With batched sparse matrix formats, the additional cost of storing the indices and the pointers can be easily amortized over an increasing number of systems in the batch. The storage requirements are:

1. **`BatchDense`**: `num_matrices x num_nnz_per_matrix`
2. **`BatchCsr`**: `[num_matrices x num_nnz_per_matrix] + [(num_rows + 1) x 1] + [num_nonzeros_per_matrix x 1]`
3. **`BatchEll`**: `[num_matrices x num_nnz_per_matrix] + [num_nnz_per_row x num_rows x 1]`

The matrices involved in the XGC simulations all share the sparsity pattern shown in Fig. 4. They originate from a 2D nine-point stencil discretization, and thus have nine diagonals grouped by three. The matrices are **not** numerically symmetric. The eigenvalue distribution is shown in Fig. 2.

The sparse matrix vector product (SpMV) kernel is the workhorse of the iterative solvers we employ, hence it is important to optimize this kernel. From the sparsity pattern in Fig. 4, we see that the matrices have a uniform number of non-zeros per row, which makes the **`BatchEll`** matrix format well-suited for this problem.

4.2. Batched iterative solvers

The XGC proxy app uses the LAPACK banded solver **dgbsv** as a batched solver on the CPU. It employs one CPU core to solve one individual system and utilizes all available cores on the CPU to solve the systems of the batch in data-parallel fashion. On NVIDIA GPUs, the cuSOLVER batched sparse QR routine, which uses the **BatchCsr** matrix format, is the only available batched sparse solver for general (non-banded) matrices. As of writing, there is no other batched sparse solver functionality provided by GPU vendors.

While direct solvers always solve the system to the full precision of the underlying type, iterative solvers come with the option of tuning the tolerance to solve the systems to the required precision. This makes iterative solvers attractive when an 'exact' solve is unnecessary; this is the case in several engineering applications and especially when the linear solve is part of a nonlinear solver. Another advantage of the iterative solvers is that we can provide an initial guess. As the outer non-linear solver in XGC performs Picard iterations, the solution of the previous step proves to be a good initial guess for the subsequent solve.

We implement batched versions of several iterative solvers. The problems we target here have relatively low condition numbers and relatively well-behaved eigenvalue distributions. Empirically, we observed that BiCGSTAB [22] was the most efficient solver and hence we show all our results with the BiCGSTAB solver.

4.2.1. Balancing flexibility and performance

Providing the possibility to precondition iterative solvers, pass an initial guess and stopping criteria requires a sophisticated design to ensure flexibility while not sacrificing performance. This is particularly important in the batched case as for optimal performance, we would like to (1) reduce the number of kernel launches, (2) minimize the data movement from the global memory and cache as much data as possible, (3) allow the compiler to be able to optimize the composite kernel, (4) maximize the occupancy of the GPU by utilizing as many warps as possible and (5) provide the GPU run-time with the freedom to schedule the different batches as necessary.

Regular (monolithic) iterative solvers are typically implemented to be highly flexible while launching separate kernels for the different components such as preconditioners, matrix-vector products, etc. However, this means that much of the data has to be fetched again from memory when executing the different components one after the other, and also when executing consecutive iterations of the solver. There is an extra latency associated with repeated kernel launches. These considerations are not important for larger problem sizes. However, for problems that are small and when the individual operations in the solver complete very quickly, these kernel launches can incur significant overheads. Therefore, we design a GPU kernel that accumulates the entire iterative solver execution, including all its components and iterations, into a single kernel launch. To avoid the overhead of launching a kernel at every iteration, we place the iteration stepping loop within the solver kernel. Each thread maintains its copy of the iteration count. As one thread-block solves one linear system and can synchronize at a relatively low cost, the value of the iteration count is enforced to be the same for all threads in a thread-block.

To preserve flexibility in the choice of solver components in a single kernel design, we use C++ templating to generate kernels for the different combinations of preconditioners, solver, and stopping criteria. This incurs the cost of instantiation at compile time while keeping the code maintainable and extensible. This also makes sure that the individual SpMV, solver, and preconditioner kernels are inlined, allowing the compiler to optimize the entire kernel as a whole.

Listing 1 CUDA kernel signature.

```
template <typename StopType, typename PrecType,
    typename LogType, typename BatchMatrixType,
    typename ValueType>
__global__ void apply_kernel(int padded_length,
    const StorageConf config, int max_iter,
    remove_complex<ValueType> tol,
    LogType logger, PrecType preconditioner,
    const BatchMatrixType a,
    const ValueType *__restrict__ b,
    ValueType *__restrict__ x,
    ValueType *__restrict__ workspace)
```

Listing 2 Kernel call site.

```
apply_kernel<stop::SimpleRelResidual<ValueType>>
    <<<nbatch, block_size, shared_size>>>(
        shared_gap, config, max_its, residual_tol,
        logger, PrecType<>(), a,
        b.values, x.values, workspace);
```

4.2.2. Monitoring iterations

Iterative solvers do not execute a pre-defined sequence of operations or iterations, but adapt the number of iterations to the problem at hand to provide a solution of the desired quality. Some systems of the batch may require more iterations than others for the same solution quality. Either all the systems need to be iterated until each of them has achieved the desired solution quality, or each system can be monitored individually allowing independent termination and logging for each linear system in the batch. Forcing all the systems to iterate until the "worst" system has converged in a SIMD fashion is inefficient because we are wasting resources on converged systems and additionally can also tend to diverge the already converged systems due to stability issues.

Monitoring the iteration process for all systems individually and scheduling the next system to resources where the iteration process has completed, on the other hand, makes more efficient use of the available resources. This breaks up the SIMD execution style of the batched routine as different systems of the batch are potentially handled with a different iteration count. This is not an issue because each of these systems is processed by one compute unit, thereby removing the need to communicate between the compute units.

For the design and interface of batched sparse iterative solvers, the system-individual convergence monitoring requires a decision on which metric to monitor, and how to define the thresholds. We decided to integrate a simple but customizable stopping criterion for the residual norm. Stopping criteria supported include a pre-defined relative residual norm reduction factor, as well as an absolute residual threshold.

4.3. Parallel execution on GPUs

GPUs are organized into parallel compute units (CU), each having a set of cores or arithmetic units as well as read-only L1 data cache and read/write shared memory. The executing threads are grouped into thread blocks. Threads in a thread block execute on a compute unit and share access to a common shared memory and can synchronize. Thread blocks are further sub-divided into warps/wavefronts, which operate in a lock-step fashion and can synchronize and communicate with very little overhead. For the batched solvers, we would like to saturate the compute units by maximizing their cache and shared memory usage and reducing loads from the global memory. Because different systems in the batch do no need to communicate nor synchronize (because of

Table 1
Some relevant theoretical performance numbers for different processors [18,19,4].

Arch	*Peak FP64* (TFlops)	*BW* (GB/s)	*(L1+SM) /CU* (KB)	*L2* (MB)	*# of SMs /CUs*
A100-40GB	9.7	1555	192	40	108
V100-16GB	7.8	900	128	6	80
MI100-32GB	11.5	1230	16+64	8	120
Xeon Gold 6148	1.0	128	64	20	20

the system-individual convergence monitoring described in section 4.2.2), it is efficient to assign the solution of one batch entry (system) to one thread block.

We have two kinds of data to deal with. First is the read-only data which includes the batch matrix indices, values and pointer arrays, and the right-hand side vector data. Ideally, we would like the entire matrix and RHS vectors to be cached for all the batch entries in the L1 data cache (which is read-only), maximizing the data reuse. Second is the read-write data which includes the auxiliary vectors of the solver and the solution vector. Ideally, we would like to store this data in the local shared memory of the compute unit, therefore minimizing the main memory accesses to these arrays that are frequently written to.

Table 1 shows the characteristics of the three GPUs that we run the batched solvers on. The L1 data cache + shared memory size per compute unit signifies the amount of local memory available in each of the compute units. The NVIDIA GPUs have the freedom to look at the L1+shared memory as a single memory level. For example, on the V100, 32 KB is reserved per CU for the L1 cache, while the shared memory per CU is configurable up to 96 KB. Memory that has not been requested by the kernel as shared memory is automatically used as L1 data cache thereby increasing the amount of L1 data cache available. On the AMD MI100, the shared memory is set to 64 KB per CU and the available L1 cache is 16 KB.

4.4. Automatic configuration of shared memory

Krylov solvers require some intermediate vectors and scalars to perform their iterations. For batched solvers, it is desirable to keep not only the matrix and right-hand side in fast memory close to the compute unit, but also these intermediate vectors. Unlike the matrix and right-hand side, these intermediate vectors are not read-only, but are modified by the kernel.

There are two ways to allocate a certain amount of shared memory for a kernel: static allocation of the memory at compile time, and dynamic allocation at run-time. We utilize dynamic shared memory allocation for all vectors. This allows us to decide at run-time the amount of needed shared memory depending on the size of the linear system to be solved.

Currently, we always allocate the matrix and right-hand side in global memory; since these are read-only, they can be cached in L1 data cache. The allocation of vectors needed by the BiCGSTAB solver is described in Algorithm 1. Any left-over vectors that could not be allocated in shared memory are allocated for all the linear systems in a block of global memory. Finally, a structure object is generated, which contains a few integers that encode the information about which vector is assigned to what memory space. This is passed to the GPU kernel where it is used to assign pointers correctly to dynamic shared memory and global memory. BiCGSTAB requires a total of 9 vectors, including the 4 'SpMV vectors'. On the V100, for example, this method allocates 6 vectors in local shared memory, while the remaining 3 vectors are allocated in global device memory.

Algorithm 1 BiCGSTAB solver. Vectors in **red** are intermediate vectors involved in matrix-vector products; these are the most preferred to be allocated in shared memory if space remains as SpMVs account for a large part of the batched solver execution time. Those in **blue** are other intermediate vectors which are allocated in shared memory only if space remains after all the **red** vectors have been allocated. Those in **green** are constant matrix or vectors.

$\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}, \hat{\boldsymbol{r}} \leftarrow \boldsymbol{r}, \boldsymbol{p} \leftarrow \boldsymbol{0}, \boldsymbol{v} \leftarrow \boldsymbol{0}$
$\rho' \leftarrow 1, \omega \leftarrow 1, \alpha \leftarrow 1$
for $i < N_{iter}$ **do**
  if $\|\boldsymbol{r}\| < \tau$ **then**
    Break
  end if
  $\rho \leftarrow \boldsymbol{r} \cdot \boldsymbol{r}'$
  $\beta \leftarrow \frac{\rho' \alpha}{\rho \omega}$
  $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta(\boldsymbol{p} - \omega \boldsymbol{v})$
  $\hat{\boldsymbol{p}} \leftarrow$ PRECOND$(\boldsymbol{p})$
  $\boldsymbol{v} \leftarrow \boldsymbol{A}\hat{\boldsymbol{p}}$
  $\alpha \leftarrow \frac{\rho'}{\hat{\boldsymbol{r}} \cdot \boldsymbol{v}}$
  $\boldsymbol{s} \leftarrow \boldsymbol{r} - \alpha \boldsymbol{v}$
  if $\|\boldsymbol{s}\| < \tau$ **then**
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \hat{\boldsymbol{p}}$
    Break
  end if
  $\hat{\boldsymbol{s}} \leftarrow$ PRECOND$(\boldsymbol{s})$
  $\boldsymbol{t} \leftarrow \boldsymbol{A}\hat{\boldsymbol{s}}$
  $\omega \leftarrow \frac{\boldsymbol{t} \cdot \boldsymbol{s}}{\boldsymbol{t} \cdot \boldsymbol{t}}$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \hat{\boldsymbol{p}} + \omega \hat{\boldsymbol{s}}$
  $\boldsymbol{r} \leftarrow \boldsymbol{s} - \omega \boldsymbol{t}$
  $\rho' \leftarrow \rho$
end for

4.5. The workhorse: `BatchCsr` and `BatchEll` SpMV kernels

To reduce the data movement to and from the global memory, we would like to avoid communication between thread blocks. Therefore, we assign one thread block to solve one system. With the fine-grained parallelism in GPUs, it is desirable that each thread block contains a number of threads proportional to the size of an individual linear system. This means that we need to tune our thread block sizes according to the problem size. However, based on the register usage by the kernel, there is a limit to how many threads can be used to solve one batch entry.

For the `BatchCsr` SpMV, we assign one warp to a row to enable coalesced access to the values in the row. Therefore, we configure the number of warps in the thread-block to be proportional to the number of rows, up to the limit imposed by the register use. For matrices with many rows and few non-zeros per row, this makes sub-optimal use of each warp, while requiring many warp iterations to traverse all the rows. For such cases, the `BatchEll` SpMV kernel is a better option. Each row is handled by one thread sequentially, thereby removing the need to communicate between the threads and the need for warp-parallel reductions. This is illustrated in Fig. 5. For our case of a 9 point stencil, this approach works well with each thread handling 9 elements per row and achieving good load balance. For matrices with more elements in a single row, it might be necessary to have multiple threads working on one row.

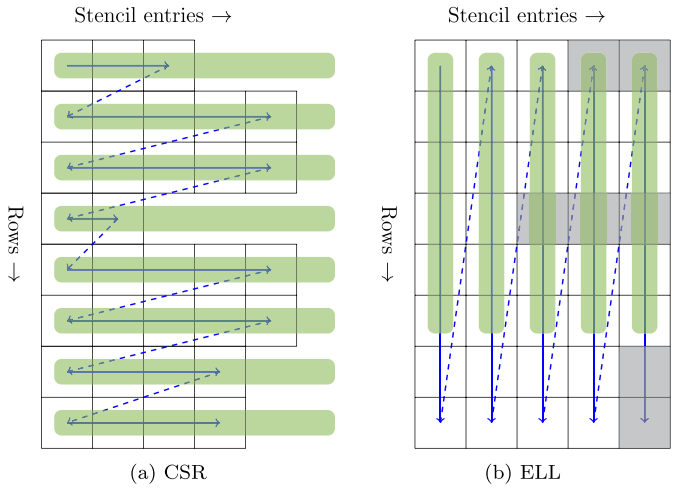

Fig. 5. Two possible layouts of the non-zero coefficients' array of a matrix with some arbitrary sparsity pattern. Green bars show how the warps are oriented for a fictitious warp length of 6.

4.6. Abstraction and integration into the XGC application

XGC uses a compressed sparse column (CSC) matrix format and assembles its matrices directly into this format. As we have seen, depending on the sparsity pattern of the matrix, it is essential to support a variety of different formats. The size of the velocity grid dictates the sizes of the arrays of column pointers and row indices. These are stored on the host before the nonlinear solver (the Picard iteration) starts. During the solve, at every nonlinear step, the CSC nonzero values array is converted to the banded format required by the LAPACK routine **`dgbsv`**, and then solved in a batched manner on the host with repeated calls to **`dgbsv`** in an OpenMP loop.

To keep the above approach as an option as well as enable the integration of faster GPU solvers and reduce data movement, an abstraction layer was developed within XGC to switch between matrix formats and linear algebra backends. The application scientist can now select **`BatchEll`** or **`BatchCsr`** as the matrix format and Ginkgo as the linear algebra backend at runtime. This is enabled with C++ polymorphic inheritance, by which the following abstract operations are provided by all possible matrix formats:

- `subtract_from_identity`, responsible for the operation $\boldsymbol{A} := \boldsymbol{I} - \boldsymbol{A}$, where $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{A}$ is the collision operator matrix. This is required to assemble the left-hand side for every Picard iteration.
- `add_identity_multiply`, the operation $\boldsymbol{y} := \boldsymbol{x} + \boldsymbol{A}\boldsymbol{x}$, required at the first Picard step.
- `apply_solve` solves the linear system associated with a Picard step.

Note that each of these functions carry out the respective operation for all matrices in the batch and hence they can be performed on the GPU without any transfers to the host.

In addition, each matrix type also needs to provide access to its values array and the map from the four-dimensional collision grid operator index space into the values array. This is required for updating the matrix values and is realized via Kokkos `View`. The four dimensions are the mesh point index, the coordinates in the 2D velocity grid and the species (ion/electron) index. The sparsity pattern and the required index map (from the collision grid operator space into the nonzero values array) are calculated in the respective matrix-type constructors, which are given read-access to the collision grid object. In this way, a new matrix type `ELLMatrix` is introduced, which uses Ginkgo's **`BatchEll`** internally.

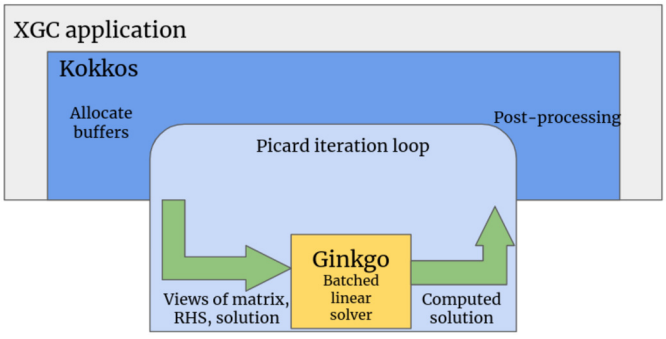

Fig. 6. A schematic that shows the integration and interaction between the three libraries, XGC, Kokkos and Ginkgo.

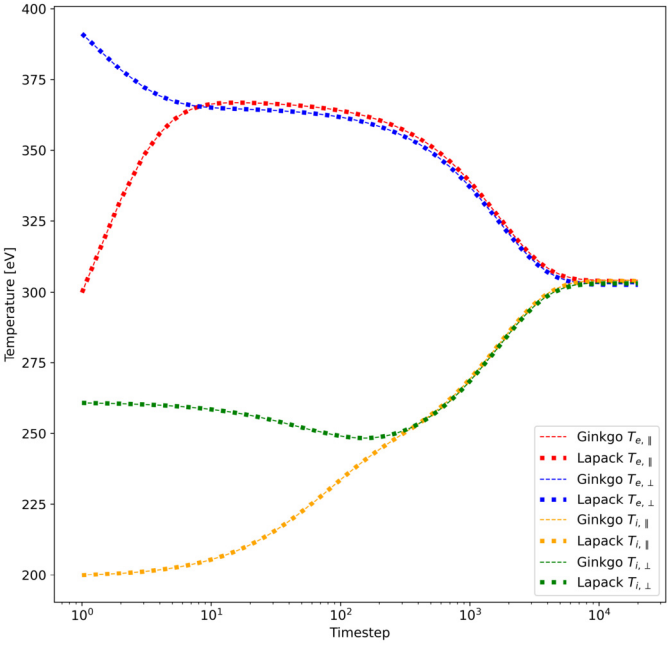

Fig. 7. Relaxation of a two species perturbed plasma towards thermal equilibrium. The parallel and perpendicular components of the temperature of the ion and electron species calculated using Ginkgo and LAPACK solvers are shown. Please refer to Fig. 3 of Ref. [15] for the results of the relaxation test used to validate the collision operator.

Fig. 6 shows the interaction between the three libraries involved. XGC uses Kokkos to allocate and manage data. The nonlinear iterations are also performed with the Kokkos API. To reduce allocation and copy overhead, Ginkgo does not allocate any memory for the input data, but uses the input data provided by Kokkos/XGC as a view, performs the linear solve using the batched routines and returns the solution back to XGC.

4.7. Correctness

To verify the correctness of the solver, we ran a relaxation test that has previously been used [15] to validate the collision kernel. In this test, a two species (ion, electron) perturbed plasma is verified to decay to thermal equilibrium by observing the convergence of the parallel as well as the perpendicular components of the temperature of both species. We have used the same test to verify the correctness of the collision kernel with the batched iterative solvers in comparison to the direct solver.

Fig. 7 shows the time evolution of the parallel and perpendicular components of temperature for the ion and electron species with the Ginkgo solver as well as the LAPACK solver, while Fig. 8

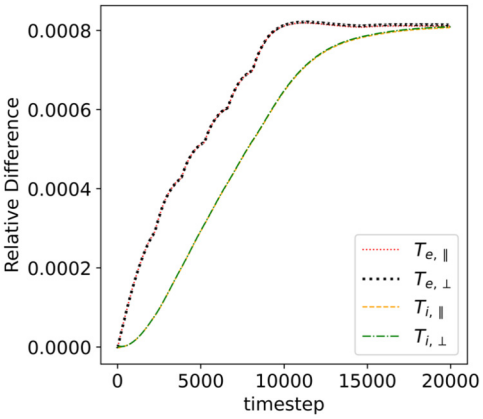

Fig. 8. Relative difference in the parallel and perpendicular components of the temperature of the ion and electron species using the GINKGO and LAPACK solvers.

Table 2
Hardware characteristics of the machines.

Machine name	GPU ; GPU BW(GB/s)	CPU ; CPU BW(GB/s) ; Sockets x cores
Summit	6xV100 ; 900	IBM Power AC922 ; 170 ; 2x22
Spock	4xMI100 ; 1200	AMD Epyc 7662 Rome ; 204 ; 1x64
Perlmutter	4xA100 ; 1555	AMD Epyc 7763 Milan ; 204 ; 1x64
Cori-GPU	8xV100 ; 900	Intel Xeon Gold 6148 ; 128 ; 2x20

shows the relative difference in the respective temperature components using GINKGO and LAPACK. This test also sets a limit on the timescale over which to validate the new solver. The temperatures are initialized according to values relevant to tokamak plasmas. As seen in Fig. 7, three timescales in the test are of relevance: (i) the convergence of the parallel and perpendicular components of the electron temperature, (ii) the convergence of the parallel and perpendicular components of the ion temperature and (iii) the convergence of the ion and electron temperatures. The longest timescale for which the simulations are expected to run depends on the time required for the convergence of the ion and electron temperatures, and the relative differences observed over this timescale reach a maximum of ~ 0.0008, which is acceptable for scientific use of XGC.

5. Experimental evaluation

We run our experiments on the following machines with the respective settings:

1. Summit: GCC 9.1, CUDA 11.0.3.
2. Spock: AMD LLVM/Clang 14.0, ROCm 4.5.
3. Cori-GPU: NVHPC 21.5, CUDA 11.3.0.
4. Perlmutter: NVHPC 22.7, CUDA 11.7.

Table 2 shows the characteristics of the machines used in this paper.

5.1. Linear solver timings

In this section, we report on the performance of the proposed batched iterative solvers on batches of matrices from the XGC proxy app. These batches consist of repetitions of ion and electron matrices similar to XGC runs. At the outset, we note that we let each system converge to an absolute residual tolerance of 10^{-10}. Conservation of relevant physical quantities in XGC to a predecided threshold (10^{-7}) was met with a minimum tolerance of 10^{-10} in the GINKGO batched iterative solver. Increasing the linear solver tolerance above 10^{-10} resulted in the Picard loop not converging within 100 iterations. Except for Fig. 11 and Fig. 12, all the figures show results from batch matrices containing both electrons and ions. The number of electron matrices is equal to the number of ion matrices in every batch that was run. While collecting the timing data, each case was repeated 10 times and averaged. We observed a very low insignificant variance.

In Fig. 9, we show timings obtained on single linear solves on the Nvidia V100, Nvidia A100 and AMD MI100, and how they compare with the LAPACK batched banded solver on the Intel Skylake node. We study the effects of using the two sparse matrix formats, and we also compare the total runtime against the batched sparse direct QR solver **`cusolverSpScsrqrsvBatched`** available in the cuSOLVER library.

Fig. 9 reveals that the batch sparse direct solver is not competitive for these problems. These matrices are sufficiently well-conditioned for the BiCGSTAB solver to converge in just a few iterations and therefore the work done to solve the system using an exact factorization does not pay off. The cuSOLVER implementation only implements the **`BatchCsr`** format which is favorable for factorizations and triangular solves. On the other hand, BiCGSTAB even with the **`BatchCsr`** format is approximately 10 to 30 times faster for our range of batch sizes.

From Fig. 9 we also see that LAPACK's banded solver, **`dgbsv`** (the blue line labeled 'Skylake') is very efficient for this problem. Kokkos is used to parallelize the batch solve: it runs each banded solve as a work-item on one core, distributing the systems in the batch among 38 of the 40 available cores on the Skylake node. It outperforms both the cuSOLVER batched QR on V100 and our batched BiCGSTAB with **`BatchCsr`** format on the MI100 GPU. This can mainly be attributed to the fact that **`dgbsv`** uses a banded storage format and this problem is well suited to it, coming from a 9-point stencil discretization. We can also observe that batched BiCGSTAB with **`BatchCsr`** on NVIDIA GPUs outperforms **`dgbsv`** on Skylake, while batch BiCGSTAB with **`BatchEll`** is significantly faster.

We observe a significant difference in the performance of **`BatchCsr`** and **`BatchEll`** for this problem on all three GPUs. Since this is a banded problem with 9 non-zeros per row except in rows corresponding to boundary points of the grid, (1) it is well-suited to a uniform rectangular storage block with very little padding necessary (only for the boundary points of the grid) and (2) with only 9 non-zeros per row, the warp-parallel reduction used by our **`BatchCsr`** SpMV is not able to utilize the warp completely. For all our experiments with the **`BatchEll`** format, we store 9 non-zeros per row. With different threads in a warp operating on different rows, and with 992 rows in the matrix, the warp is well-utilized. The nonzeros in each row are processed sequentially, and hence 9 warp-iterations are needed to process all the columns in each row. The data is stored column-major to get coalesced memory access.

With **`BatchCsr`**, a warp of 32 threads has only 5 threads (9 divided by 2, rounded up) active in the first reduction stage, therefore the warp is not well-utilized. This is exacerbated in the AMD GPUs which have a warp (wavefront) size of 64, thereby providing us with higher speedups for **`BatchEll`** compared to **`BatchCsr`**. This is corroborated by wavefront (warp) utilization data from the ROCm profiler, **`rocprof`**. On the entire BiCGSTAB solve, we observe an overall high wavefront utilization with batch ELL (Table 3).

We also note the clear step-like trend for the AMD GPU in Fig. 9. There are discrete jumps at multiples of 120 because the MI100 has 120 compute units. To schedule the next system after a multiple of 120, the scheduler needs to wait for one of the compute units to be available. Let us consider the curves for the **`BatchEll`** format on the MI100 (red circles) and on the V100 (yellow circles) in Fig. 9. The V100 has a smooth trend in the time

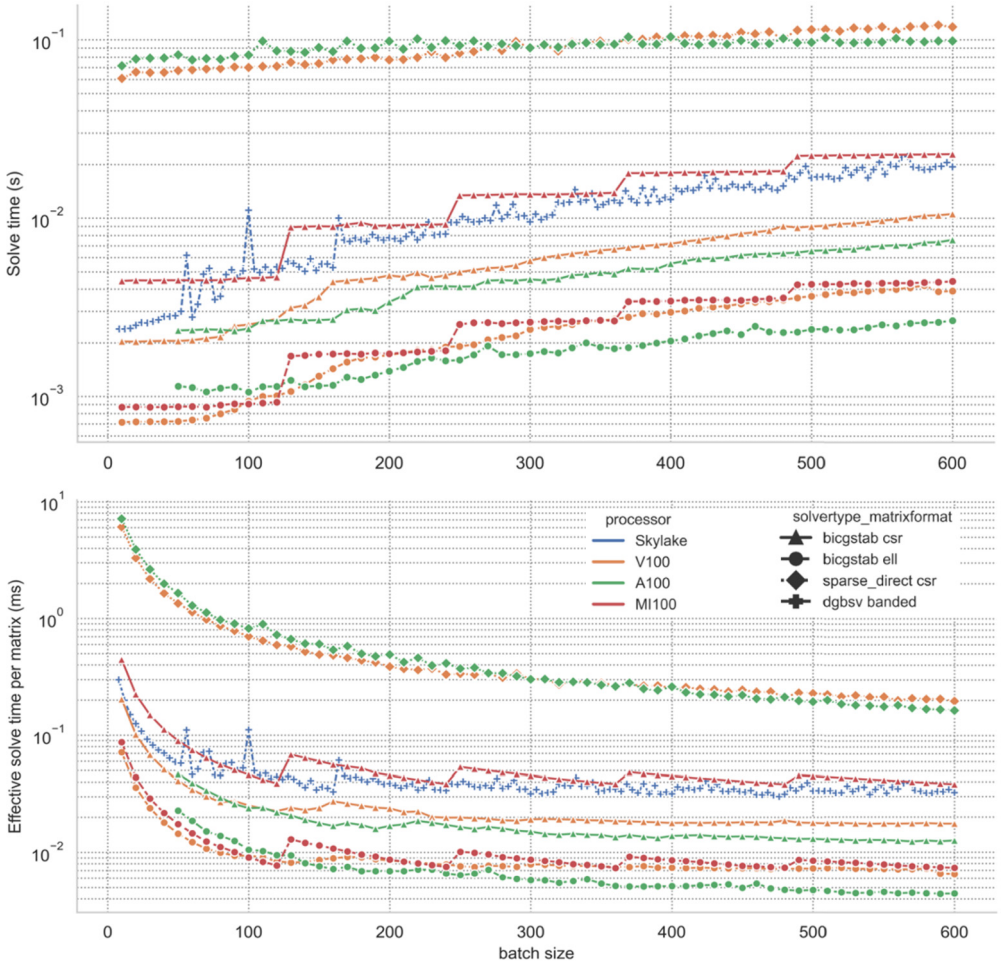

Fig. 9. Time taken by different solvers, with different matrix formats, on different platforms, as a function of the batch size (left: total time per solve, right: time per matrix entry).

Table 3
Performance metrics on different platforms with the two batch matrix formats (L1 cache data was not available for the AMD MI100).

Processor, format	Wavefront /warp use %	L1 hit rate %	L2 hit rate %
V100, CSR	75.1	50.7	63.1
V100, ELL	98.2	24.5	63.1
A100, CSR	72.9	76.6	97.2
A100, ELL	98.2	74.5	94.8
MI100, CSR	52	-	86
MI100, ELL	94	-	88

to solution and does not exhibit jumps at multiples of its number of compute units (80). It can solve the problem in less time than what is needed to schedule and complete an entire grid of thread blocks when there are only a few matrices left to solve. This may be due to more flexible scheduling of the thread blocks on the V100 compute units (SMs) with which they can take advantage of the non-synchronized convergence behavior of the ion-electron composite batch system.

The right plot in Fig. 9 shows the variation of the average time for the solution per batch matrix entry as a function of the batch size. These curves clearly show that with increasing batch size, the time to solution needed per batch matrix entry decreases showing that we are progressively saturating the GPU.

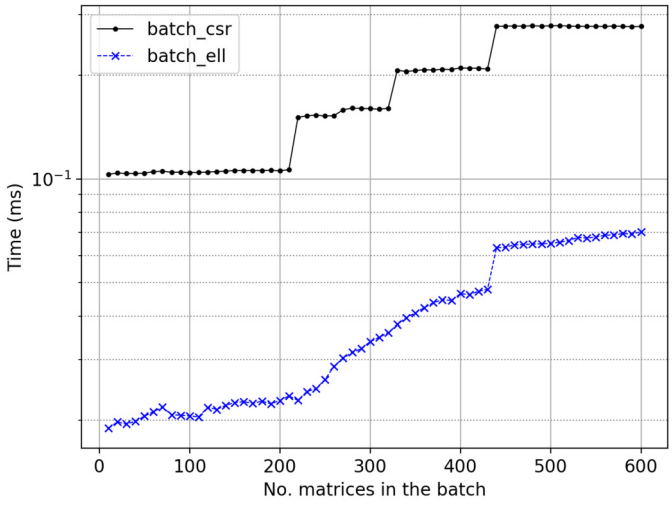

Fig. 10. Total time taken by the SpMV kernels on A100.

To isolate the impact of the sparse matrix format, we show in Fig. 10, the timing plots for the sparse-matrix vector kernel for both the **`BatchCsr`** and the **`BatchEll`** formats on the A100 GPU. We observe that due to the factors previously mentioned, the **`BatchEll`** format is the superior format for the problem at hand.

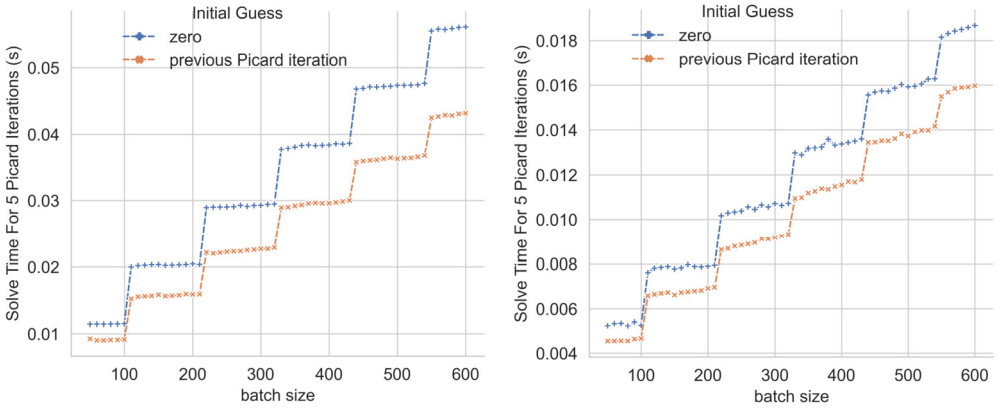

Fig. 11. Effect of using initial guess from the previous Picard iteration on total time to solution (cumulative over all the Picard iterations), *Left*: with `BatchCsr` format, *Right*: With `BatchEll` format.

Table 4
Number of iterations needed for the linear solve inside successive Picard iterations using the previous Picard iteration solution as initial guess (With `BatchEll` format and an absolute tolerance of 10^{-10}).

Picard iteration	#iters for electron species	#iters for ion species
0	30	5
1	28	4
2	20	3
3	16	2
4	12	2

5.2. Picard loop

With iterative solvers, we can provide contextual information about the problem to the algorithm in the form of an initial guess. A good initial guess can significantly reduce the iteration count needed and this is quite beneficial for reducing the overall time to solution.

With batched iterative solvers, when solving independent linear systems with possibly different convergence properties, the effects of initial guess are pronounced for those systems that have a higher iteration count. In our case, the electron system requires a moderate number of iterations, around 35 with an initial guess of all zeros. Using the solution from the previous Picard iteration, we can reduce the iteration count for successive linear solves in the nonlinear solver. In the XGC proxy-app, we have 5 Picard iterations and the linear solver iteration counts for successive Picard iterations are shown in Table 4. We see a significant reduction in iteration count which translates to an overall faster time to solution.

In Fig. 11, we see the time to solution for two different initial guesses. With the solution of the previous Picard iteration as the initial guess for the solution of the subsequent Picard iteration, we obtain a significant speedup due to a reduction in the number of linear solver iterations for the same solution quality. For the CSR format, we see speedups of ~ 1.15 to ~ 1.25 in terms of total time, while for ELL format we see speedups between ~ 1.2 up to about ~ 1.6 compared to using a zero initial guess for the A100 GPU with the batched BiCGSTAB solver.

Finally, in Fig. 12, we show the speedups obtained with the batch iterative solvers on the GPU platforms over the `dgbsv` solver on the Skylake CPU. The total time required for all 5 Picard iterations is used for this plot. As explained in the previous paragraph, we use the solution of the previous Picard iteration as the initial guess for the batched iterative linear solver in the subsequent Picard iteration. The `BatchEll` format is used for these runs. As expected, the speedup for the ion systems is the largest, because they need few iterations. For the combined batches with equal

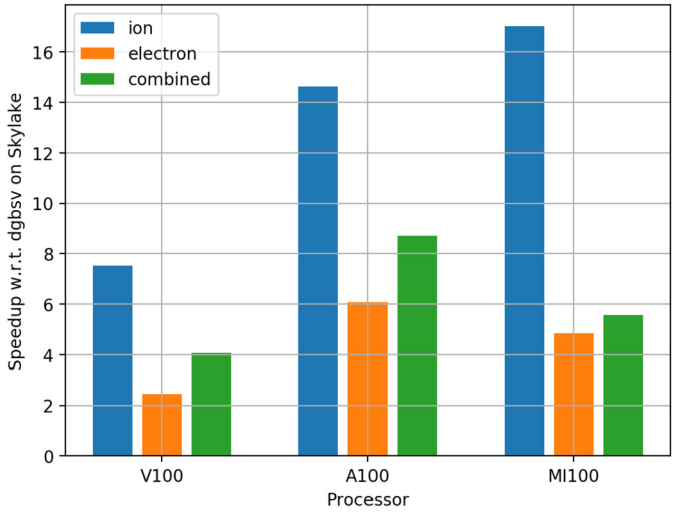

Fig. 12. Speedup for 5 Picard iterations using batched BiCGSTAB on GPUs over the LAPACK banded solver on the CPU (Skylake).

numbers of ion and electron matrices, we get effective speedups between $4\times$ and almost $9\times$ depending on the GPU architecture without sacrificing the accuracy required by the application.

5.3. Collision kernel

In this subsection, we report the performance of the entire XGC collision kernel using the Ginkgo batched solver. This provides an idea of the application speedup after moving the Picard loop to the GPU utilizing the Ginkgo batched iterative solvers. Unlike the results in earlier subsections, the total problem size (the number of physical mesh nodes, and thus the number of matrices to be solved) is now fixed. Instead of collecting all the individual problems in a single batch, we choose a batch size and split the problem into several batches to be solved one after another. Typically, the batch size is chosen to deal with memory constraints in other parts of the collision kernel. Certain operations are still done on the CPU, such as the Picard loop convergence check. This is shown in Fig. 13. The proportion of time taken by the linear solver is significantly reduced by performing it on the GPU using the Ginkgo batched solver.

Fig. 14 shows the overall speedup on a single GPU over a proportional number of CPU cores on the node, of the collision kernel. By 'proportional', we mean that the same fraction of cores per node were used as the fraction of GPUs per node. This has been explained in Table 5.

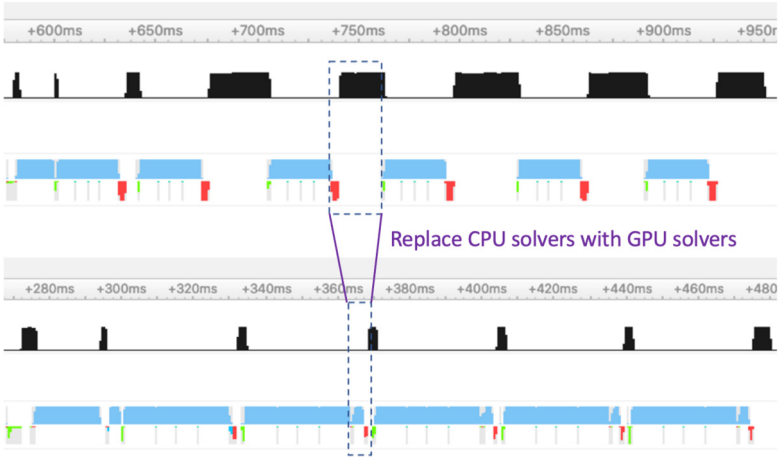

Fig. 13. Trace before and after Ginkgo integration. Blue bars show GPU kernel execution; black bars show work done on the CPU. The proportion of time spent in the linear solver, and thus on the CPU, is greatly reduced. Note that the horizontal time scale is smaller for the bottom plot, thus the speedup is greater than what is visually seen here.

Table 5
Systems and resources used for investigating overall collision kernel performance (Fig. 14).

Machine	CPU	Total cores	GPU	Total GPUs	Cores used	GPUs used
Cori GPU	Xeon 6148	40	V100	8	5	1
Summit	Power 9	42	V100	6	7	1
Perlmutter	Epyc 7763	64	A100	4	16	1
Spock	Epyc 7662	64	MI100	4	16	1

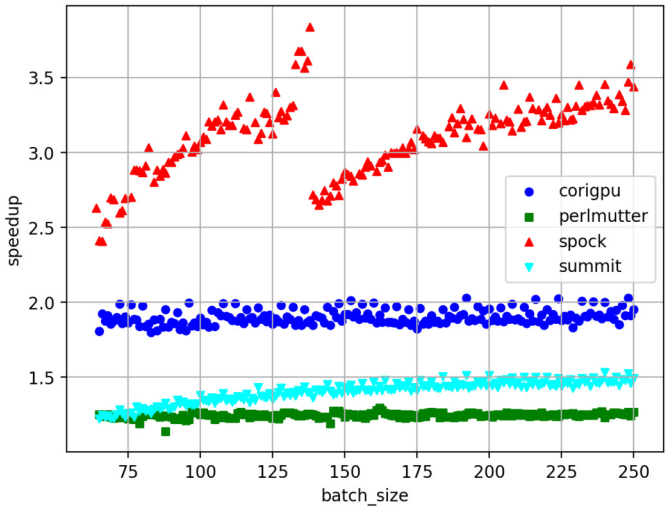

Fig. 14. Speedup of the collision kernel using Ginkgo batched BiCGSTAB (with `BatchEll` matrix format) on GPUs over that using the banded solver on the corresponding CPU on three different supercomputer systems (Table 5).

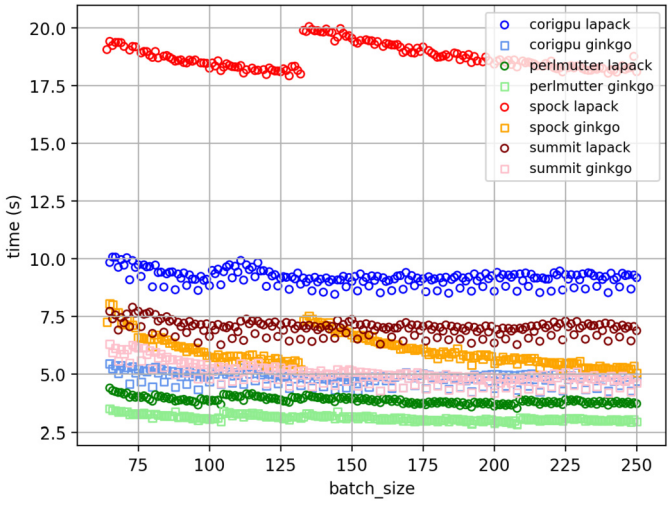

Fig. 15. Time taken by the collision kernel using Ginkgo batched BiCGSTAB (with `BatchEll` matrix format) on GPUs over that using the banded solver on the corresponding CPU.

We chose this way for a fair estimate of the expected speedup because the integrated code does not yet make use of the Message Passing Interface (MPI) for multi-GPU or multi-node runs. This is a topic of ongoing development. Note that this analysis is really relevant for estimating the speedup on different supercomputer systems, not for comparing the CPUs or GPUs since the system configurations are very different.

The kernel is run for a total of 3000 mesh nodes and two species (ion, electron) with different batch sizes. The Picard loop is run to a tolerance of 10^{-7} in the conservation error. The batch sizes in the figure are limited to 256 to take into account memory constraints of the entire collision kernel.

Performance varies with the batch size for a fixed number of total matrices because of multiple factors. It depends on how far the batch size is from a multiple of the number of compute units in the GPU, as was discussed for the step-like trend in Fig. 9. It may also depend on the L2 cache size. For small to moderate batch sizes, a batch may fit in the L2 cache. Among the GPUs used in this work, the A100 has the largest L2 cache. It is not yet clear why the speedup falls for Spock at a batch size of 133. This may be a subject of future investigation. We show the raw timings in Fig. 15, where we observe that both the GPU and CPU on Spock slow down at that batch size. The interesting trends with respect to batch size notwithstanding, even with certain parts of the code

still on the CPU, we obtain a speedup of 1.2x to 1.3x on Perlmutter, 1.8x-2x on Cori GPU, and 2.4x-3.9x on Spock.

6. Conclusion

We have presented the use of Ginkgo batched sparse iterative solver functionality for accelerating XGC simulations. We have evaluated the batched iterative solver functionality for matrices representative of the electron and ion species on V100, A100 and MI100 GPUs and compared them against the batched banded solvers running on the CPU. The results demonstrate that the batched sparse iterative solvers in Ginkgo are effective, efficient, and well-suited for integration into XGC. The results also underscore the importance of using an efficient sparse matrix format and the benefits of using batched iterative solvers over their batched direct counterparts.

We have noted that using Ginkgo's batched iterative solvers in conjunction with Kokkos allows for seamless execution of XGC on exascale-oriented heterogeneous architectures. This enables the execution of the Ginkgo-accelerated XGC simulations on various leadership supercomputing facilities. The overall collision kernel speedups indicate a seamless integration with little overhead that preserves, to a significant extent, the speedups obtained by the linear solver alone.

Future work will focus on complete XGC simulation runs utilizing this new solver and integration. Runs on multiple GPUs will also be enabled via the Message Passing Interface (MPI). Further, development of batched banded solvers for GPU architectures is ongoing. It is of interest to investigate hybrid deployment of linear solvers that complement the use of batched banded solvers in the initial Picard iteration with iterative solution updates in the subsequent Picard iterations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. It used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Some work in this paper was also performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research, Germany. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

References

[1] A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N.J. Higham, J. Kurzak, P. Luszczek, S. Tomov, M. Zounon, A set of batched basic linear algebra subprograms and LAPACK routines, ACM Trans. Math. Softw. 47 (3) (2021) 21, https://doi.org/10.1145/3431921.

[2] I. Aggarwal, A. Kashi, P. Nayak, C.J. Balos, C.S. Woodward, H. Anzt, Batched sparse iterative solvers for computational chemistry simulations on GPUs, in: 2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2021, pp. 35–43, https://doi.org/10.1109/ScalA54577.2021.00010, https://ieeexplore.ieee.org/document/9652814.

[3] I. Aggarwal, A. Kashi, P. Nayak, C.J. Balos, C.S. Woodward, H. Anzt, Batched sparse iterative solvers for computational chemistry simulations on GPUs, in: 2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2021, pp. 35–43, https://doi.org/10.1109/ScalA54577.2021.00010.

[4] AMD, MI100 white paper, https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf.

[5] H. Anzt, J. Dongarra, G. Flegar, E.S. Quintana-Ortí, Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs, in: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–10, https://doi.org/10/gnn4fz.

[6] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.M. Tsai, E.S. Quintana-Ortí, Ginkgo: a modern linear operator algebra framework for high performance computing, ACM Trans. Math. Softw. 48 (1) (2022) 2, https://doi.org/10.1145/3480935.

[7] H. Anzt, T. Cojean, Y.-C. Chen, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.-H. Tsai, Ginkgo: a high performance numerical linear algebra library, J. Open Sour. Softw. (Aug. 2020), https://doi.org/10.21105/joss.02260.

[8] E. Carroll, A. Gloster, M.D. Bustamante, L.O. Naraigh, A batched GPU methodology for numerical solutions of partial differential equations, arXiv:2107.05395, 2021.

[9] J. Dominski, S. Ku, C.-S. Chang, J. Choi, E. Suchyta, S. Parker, S. Klasky, A. Bhattacharjee, A tight-coupling scheme sharing minimum information across a spatial interface between gyrokinetic turbulence codes, Phys. Plasmas 25 (7) (2018) 072308, https://doi.org/10.1063/1.5044707.

[10] J. Dongarra, S. Hammarling, N.J. Higham, S.D. Relton, P. Valero-Lara, M. Zounon, The design and performance of batched BLAS on modern high-performance computing systems, Proc. Comput. Sci. 108 (2017) 495–504, https://doi.org/10.1016/j.procs.2017.05.138.

[11] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N.J. Higham, J. Hogg, P. Valero-Lara, S.D. Relton, S. Tomov, M. Zounon, A proposed API for batched basic linear algebra subprograms, http://eprints.ma.man.ac.uk/2464/, Apr. 2016.

[12] H.C. Edwards, C.R. Trott, D. Sunderland, Kokkos: enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[13] N.M. Evstigneev, O.I. Ryabkov, E.A. Tsatsorin, On the inversion of multiple matrices on GPU in batched mode, Supercomput. Front. Innov. 5 (2) (2018) 23–42, https://doi.org/10.14529/jsfi180203.

[14] A. Gloster, L. Ó Náraigh, K.E. Pang, cupentbatch—a batched pentadiagonal solver for NVIDIA GPUs, Comput. Phys. Commun. 241 (2019) 113–121, https://doi.org/10.1016/j.cpc.2019.03.016.

[15] R. Hager, E. Yoon, S. Ku, E. D'Azevedo, P. Worley, C. Chang, A fully non-linear multi-species Fokker–Planck–Landau collision operator for simulation of fusion plasma, J. Comput. Phys. 315 (2016) 644–660.

[16] A. Kashi, P. Nayak, D. Kulkarni, A. Scheinberg, P. Lin, H. Anzt, Batched sparse iterative solvers on GPU for the collision operator for fusion plasma simulations, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 157–167, https://doi.org/10.1109/IPDPS53621.2022.00024.

[17] NVIDIA, cuSOLVER - GPU accelerated library for decompositions and linear system solutions on NVIDIA GPUs, https://docs.nvidia.com/cuda/cusolver/index.html. (Accessed 24 August 2021).

[18] NVIDIA, Ampere A100 white paper, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[19] NVIDIA Volta, V100 white paper, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[20] P. Valero-Lara, I. Martínez-Perez, A.J. Peña, X. Martorell, R. Sirvent, J. Labarta, cuHinesBatch: solving multiple hines systems on GPUs human brain project, Proc. Comput. Sci. 108 (2017) 566–575, https://doi.org/10.1016/j.procs.2017.05.145.

[21] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, A.J. Peña, cuThomasBatch and cuThomasVBatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA GPUs, Concurr. Comput., Pract. Exp. 30 (2018).

[22] H.A. van der Vorst, Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 13 (1992) 631–644, https://doi.org/10.1137/0913035, publisher: Society for Industrial and Applied Mathematics.

[23] E.S. Yoon, C.S. Chang, A Fokker-Planck-Landau collision equation solver on two-dimensional velocity grid and its application to particle-in-cell simulation, Phys. Plasmas 21 (032503) (2014), https://doi.org/10.1063/1.4867359.