Batched sparse iterative solvers on GPU for the collision operator for fusion plasma simulations

Aditya Kashi[1], Pratik Nayak[1], Dhruva Kulkarni[2], Aaron Scheinberg[3], Paul Lin[2], Hartwig Anzt[1,4]
[1]Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2]Lawrence Berkeley National Laboratory (LBNL), Berkeley, USA
[3]Jubilee Development, Cambridge, USA
[4]University of Tennessee (UTK), Knoxville, USA
aditya.kashi@kit.edu, pratik.nayak@kit.edu, dkulkarni@lbl.gov, aaron@jubileedev.com, paullin@lbl.gov hartwig.anzt@kit.edu

***Abstract*—Batched linear solvers, which solve many small related but independent problems, are important in several applications. This is increasingly the case for highly parallel processors such as graphics processing units (GPUs), which need a substantial amount of work to keep them operating efficiently and solving smaller problems one-by-one is not an option. Because of the small size of each problem, the task of coming up with 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, and this presents further challenges.**

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 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 the software architecture can be used to develop an efficient solution for the XGC collision proxy app. 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 OpenMP threads are presented for matrices representative of those required in the collision kernel of XGC. 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 for exascale oriented heterogeneous architectures at the various leadership supercomputing facilities.

***Index Terms*—Sparse linear systems, batched solvers, GPU, performance portability, GINKGO, XGC, ITER, WDMAPP, fusion, simulation**

I. INTRODUCTION

Alternate energy sources based 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 capable architectures as well as the two upcoming US Department of Energy (DOE) exascale capable 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 – the gyrokinetic Particle-In-Cell (PIC) XGC code is used 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 simulating multiple species of particles in a plasma (ions, electrons). The 'collision' step – describing Coulomb collisions between particles in the plasma – has been identified as a bottleneck in XGC, and a proxy app (referred to as the 'collision kernel') has been created for facilitating optimizations and porting to GPUs. Currently, the collision kernel utilizes MPI for multiple CPU nodes, and Kokkos [10] to offload to GPUs as well as utilize OpenMP for intranode 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. The banded solver 'dgbsv' from LAPACK is currently 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 im-

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. 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.

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00024

plementation 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. However, this has to be coupled with management of kernel launch overhead and efficient memory use.

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 solver with a Jacobi preconditioner.
3) We tune the batched BiCGSTAB solver for the matrices from the 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 level implementation of these Batched solvers within GINKGO [3], which is readily available for applications along with examples.

In Section II, 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 solver and batched routines are explored in Section III. 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 Section IV. Comparisons for the solve times on NVIDIA V100 and A100 GPUs with one dual-socket Intel Xeon Skylake CPU compute node with 40 OMP threads are presented in section Section V.

II. MOTIVATION

A. 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. Therefore 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. Production simulations currently employ the LAPACK banded solver **`dgbsv`** on the CPU for the linear solve. However, as more and more of XGC is ported to GPU, the **`dgbsv`** time on the CPU becomes a larger and larger fraction of the run time, hence the need for a fast and efficient linear solver on the GPU.

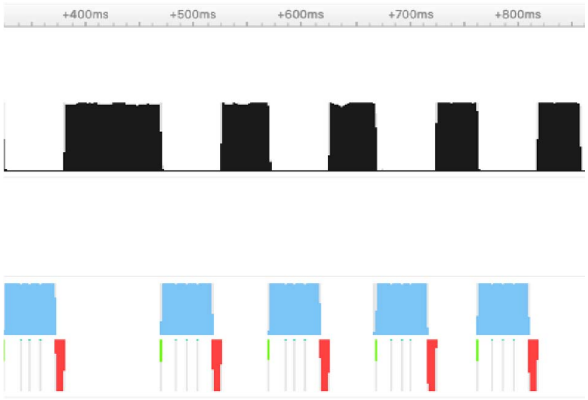


Fig. 1: Profile of one Picard loop of the collision kernel proxy app showing time spent on CPU (black), GPU (blue), and memory transfer (red: Device to Host, green: Host to Device)

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 ($\sim$10) and electrons, the proxy app currently simulates a plasma with one ion species (along with electrons). A backward Euler time discretization and Picard iteration is employed for the two species for every mesh node. The Picard loop typically requires five iterations for convergence. Figure 1 shows the execution timeline of one such Picard iteration captured on one MPI rank with multiple OpenMP threads on the CPU and one GPU - the top half of the figure (black rectangles) shows CPU execution of the linear solver employed in the Picard iteration and associated processing, while the bottom half shows GPU execution (blue rectangles) as well as data transfer (red and green rectangles). As seen in Figure 1, a significant portion of the execution time for the Picard loop ($\sim$48%) is on the CPU - of which a majority of the time is spent in the solve (**`dgbsv`** call) itself ($\sim$66%). In addition, data on the GPU needs to be transferred back and forth between the GPU and the CPU to employ a linear solver on the CPU - which causes additional overhead ($\sim$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 sizes 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 sizes 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 Figure 2 that the matrices for the ion and the

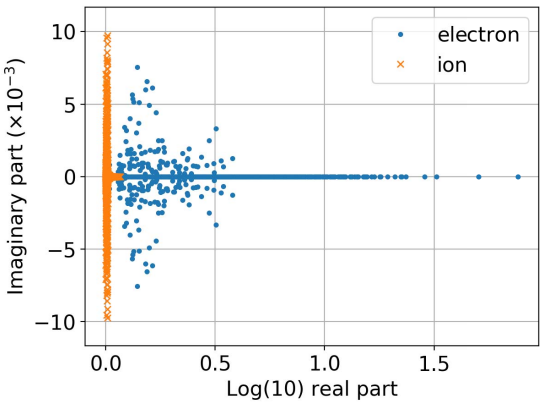


Fig. 2: Eigenvalues of the matrices for the two species

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 lead to very quick convergence, while electrons have a greater range of real parts of the eigenvalues which means it will take some more iterations to converge. That being said, they are both well-conditioned enough to take good advantage 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 block-diagonal matrices with sparse diagonal blocks. This larger system may have a size suitable to utilize the entire device. However, in such a method, the number of iterations over the entire block system would be determined by the most difficult individual 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, internal experiments have shown that such a method is slower than the proposed batched iterative solvers.

III. Related Work

With the advent of parallel computers, there has been some effort in using BLAS and LAPACK routines in a batched fashion to take advantage of the embarrassing parallelism that these batched functionality can provide. Recently, a batched BLAS interface was also proposed [9] 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].

For the GPU, there has also been some work in the direction of batched dense inversions for Block-Jacobi preconditioners [4] and for smaller solving small dense problems in batched mode [11]. Factorizing batched dense matrices has also received attention, particularly for LU factorization, using the `DGETRF()` routine [8].

While batched dense linear algebra has received attention in recent years, batched sparse and iterative solvers remain an unexplored space. In terms of batched sparse direct solvers, there has been some work on tridiagonal and pentadiagonal systems. NVIDIA cuSparse provides the `gtsv2StridedBatch` routine based on variants of cyclic reduction.

Other methods for tridiagonal and pentadiagonal matrices, some of which have been adopted in cuSparse, have also been proposed [6], [12], [17]. These methods, as opposed to the currently proposed algorithms, aim to specifically solve tridiagonal and penta-diagonal systems and thus have algorithms specific to such systems based on the well-known Thomas' algorithm. Further, the solve stages on the GPU are not fine-grain parallel since they aim to have exact solves. Each GPU thread solves an entire linear system, and the storage of the systems in the batch is interleaved to provide coalesced access. 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 [6], [12], 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 [15]. The general dogma has been that with relatively small linear systems, direct solvers are more effective. While this is true for some cases, flexibility provided by the iterative solvers in terms of early stopping, reuse of initial guess and adaptability to matrix properties can make them very attractive even for relatively small problems.

IV. 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. 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 them 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.

A. Batch matrix storage formats

Batched solvers are favorable in cases where the individual matrices are small, and large numbers of these linear systems need to be solved. Additionally, shared sparsity patterns allow for reduced storage, storing the sparsity pattern only once, while storing the values of all the entries.

Even for batched matrices with small individual matrix sizes, sparse storage formats can be beneficial. Batched linear systems may share sparsity patterns eg. if these linear systems arise from similar local physics at many grid points or one wants to solve multiple independent problems that all employ the same discretization scheme. This allows us to store a single common sparsity pattern and cache it if possible to minimize the data movement. 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.

Figure 3 shows 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]`

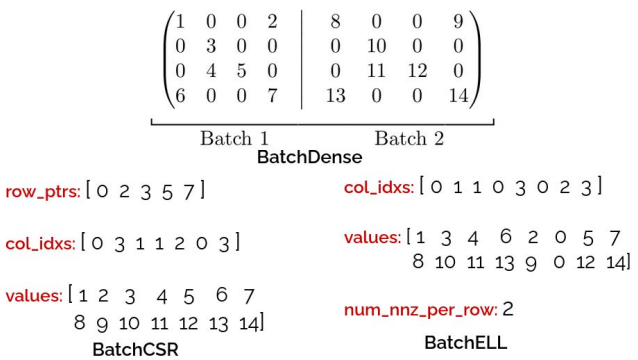


Fig. 3: Batch Matrix Storage formats - **`BatchDense`**, **`BatchCsr`** and **`BatchEll`**

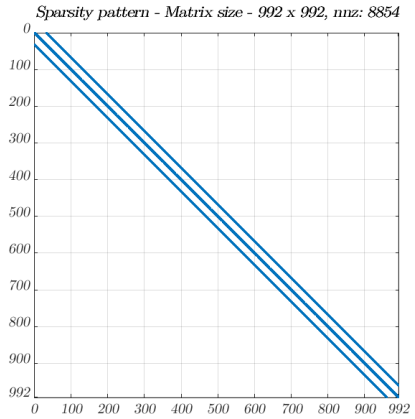


Fig. 4: Sparsity pattern of an individual entry of the batched matrix: 992 rows, 9 nonzeros per row.

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 Figure 4. They originate from a 2D-nine point stencil discretization. The matrices are **not** numerically symmetric. The eigenvalue distribution is shown in Figure 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 Figure 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.

B. Iterative and Direct batched solvers

In contrast with monolithic non-batched solvers, batched solvers can take advantage of the shared sparsity pattern and also solve the different independent linear systems in parallel. Batched dense direct solvers have been widely used in literature [2], [5]. Efforts in the direction of sparse batched

direct solvers have been limited to special systems such as banded, tri-diagonal or penta-diagonal systems [6].

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 batched systems in parallel. 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 solution 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 does 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 preconditionable iterative solvers. The problems we target here have relatively low condition numbers and relatively well-behaved eigenvalue distributions. Empirically, we observed that BiCGSTAB [18] was the most efficient solver and hence we show all our results with the BiCGSTAB solver.

Composability of different preconditioners, solvers, and stopping criteria requires careful design to ensure flexibility while not sacrificing performance. This is particularly important in the batched case because the individual matrices are much smaller than in monolithic solvers. 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. In order 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 own 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 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);
```

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. While this breaks up the SIMD execution style of the batched routine as different items of the batch are potentially handled with a different iteration count, if each of these systems is handled by a distinct compute unit, thereby removing the need to communicate between the compute units, it can provide optimal performance.

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. Currently available stopping criteria include a pre-defined relative residual norm reduction factor, as well as an absolute residual threshold.

C. Parallel execution on GPUs

GPUs are organized into parallel compute units (CU), each having its local data caches and shared memory. For the batched solvers, we would like to saturate these units by maximizing their cache and shared memory usage and reducing unnecessary loads from the global memory. The executing threads are grouped into thread blocks, which are further sub-divided into warps/wavefronts, which operate in a lock-step fashion. As GPUs require this kind of fine-grained parallelism, it is efficient to assign the solution of one batch entry to one thread block. Each thread block consists of threads that execute on a compute unit and work in parallel. Hence the solution of each batch entry also can be performed in parallel.

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, maximizing the data reuse and minimizing cache misses. 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. The local shared memory is shared between the threads of a single thread block, therefore using this for read-write arrays allows for efficient communication between the threads.

Table I shows the characteristics of the three GPUs that we run the batched solvers on. The number of compute units represents the number of independent multi-processors available on the GPU. The L1 data cache + shared memory size per compute unit signifies the amount of 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 upto 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.

D. 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.

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

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. As an example, BiCGStab requires a total of 9 vectors, including the 4 'SpMV vectors'. On the V100, this method allocates 6 vectors in local shared memory, while the remaining 3 vectors are allocated in global device memory.

TABLE I: Some relevant theoretical performance numbers for different processors [7], [14], [16]

Architecture	*Peak FP64* (TFlops)	*Main memory BW* (GB/s)	*(L1 + shared memory) /CU* (KB)	*L2 data cache* (MB)	# of SMs/CUs
NVIDIA A100-40GB (Ampere)	9.7	1555	192	40	108
NVIDIA V100-16GB (Volta)	7.8	990	128	6	80
AMD MI100-32GB (CDNA)	11.5	1230	16+64	8	120
Intel Xeon Gold 6148 (single)	1.0	128	64	20	20

E. 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 Figure 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.

V. Experimental Evaluation

In this section, we report on the performance of the proposed batched iterative solvers on batches of matrices from the XGC mini-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 pre-decided 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 up to 100 iterations. Except for Figure 8 and Figure 9, 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. The CPU platform on which we run the XGC proxy-app solver is an Intel Skylake node with two sockets of the Xeon Gold 6148 processor. Each socket has a total of 20 cores with the total core count in the entire node equal to 40.

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

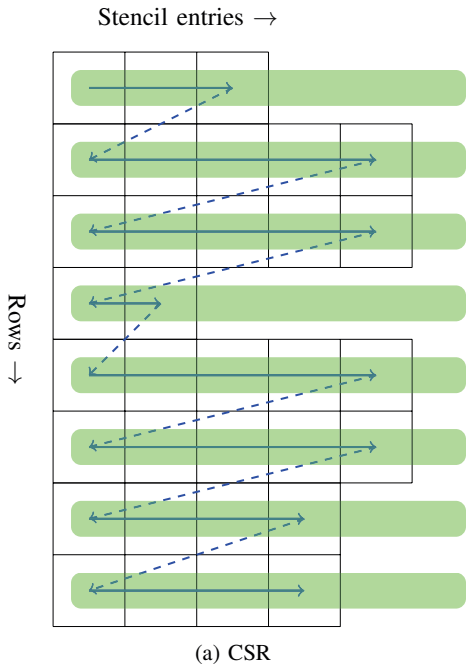
1) V100: On Summit with GCC 9.1 and Cuda 11.0.3.
2) A100: With NVHPC 21.7 compiler suite.
3) MI100: With LLVM/Clang 12.0 and ROCm 4.2.
4) Intel Skylake: With NVHPC 21.7 compiler suite.

In Figure 6, 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, parallelized over different matrices on the Intel Skylake node. We study the effects of using the two sparse matrix formats, and we also compare against the batched sparse direct QR solver **`cusolverSpScsrqrsvBatched`** available in the cuSolver library.

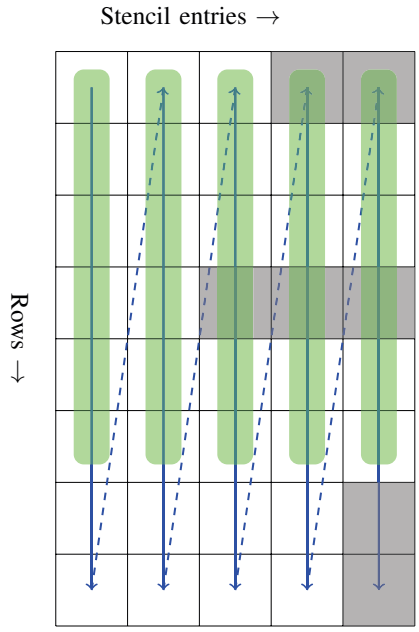
It is immediately clear from Figure 6 that the batch sparse direct solver is not competitive for these problems. These matrices are sufficiently well-conditioned for the BiCGStab solver to converge in just a few iterations and therefore the work done to solve the system using an exact factorization does 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 (Figure 6).

From Figure 6 we also see that LAPACK's banded solver, **`dgbsv`** (the blue line labelled '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 is able to outperform **`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



(a) CSR



(b) ELL

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.

TABLE II: 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 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. Each row is processed sequentially, and hence 9 warp-iterations are needed to process all the columns in each row. The data is stored column-major to make sure we get coalesced memory accesses

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 II).

We also note the clear step-like trend for the AMD GPU. 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 Figure 6. The V100 has a smooth trend in the time to solution and does not exhibit jumps at multiples of its number of compute units (80). It is able to 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 are able to take advantage of the non-uniform convergent behavior of the ion-electron composite batch system.

The right figure in Figure 6 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, time to solution needed per batch matrix entry decreases showing that we are saturating the GPU.

In order to isolate the impact of the sparse matrix format, we show in Figure 7, 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 factors previously mentioned, the **`BatchEll`** format is the superior format for the problem at hand.

With iterative solvers, we have the ability to provide con-

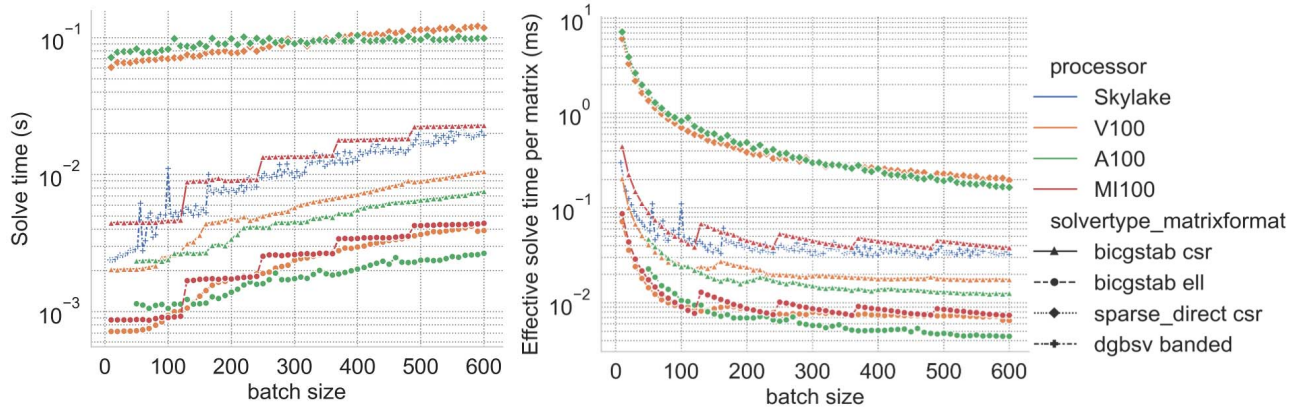


Fig. 6: Time taken by different solvers, with different matrix formats, on different platforms, as a function of the batch size (left: total time per solve, right: time per matrix entry)

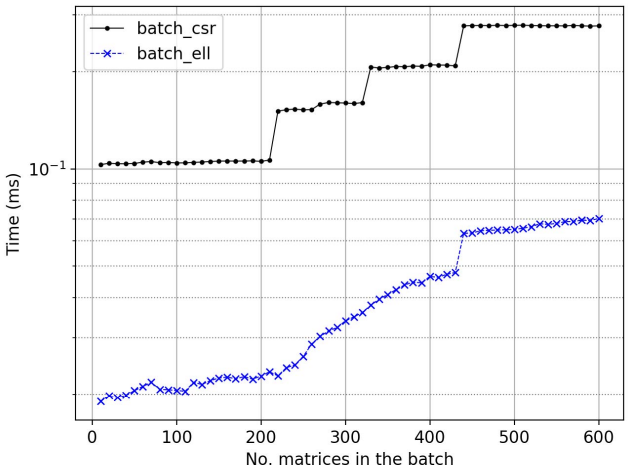


Fig. 7: Total time taken by the SpMV kernels on A100

textual information about the problem to the algorithm in the form of an initial guess. In Figure 8, we investigate the impact of using an initial guess for the linear solves inside the non-linear Picard iteration. A good initial guess can significantly reduce the iteration count needed and hence in solution-based non-linear solvers with an inner linear solver, this can be quite beneficial to reduce 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 only 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 III. We see the significant reduction in

TABLE III: 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

iteration count which translates to an overall faster time to solution.

In Figure 8, 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 linear solve 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 $\sim$ 1.15 to $\sim$ 1.25 in terms of total time, while for ELL format we see speedups between $\sim$ 1.2 up to about $\sim$ 1.6 compared to using a zero initial guess for the A100 GPU with the batched BiCGStab solver.

Finally, in Figure 9, 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 numbers of ion and electron matrices, we get effective speedups between 4x and almost 9x depending on the GPU architecture without sacrificing the accuracy required by the application.

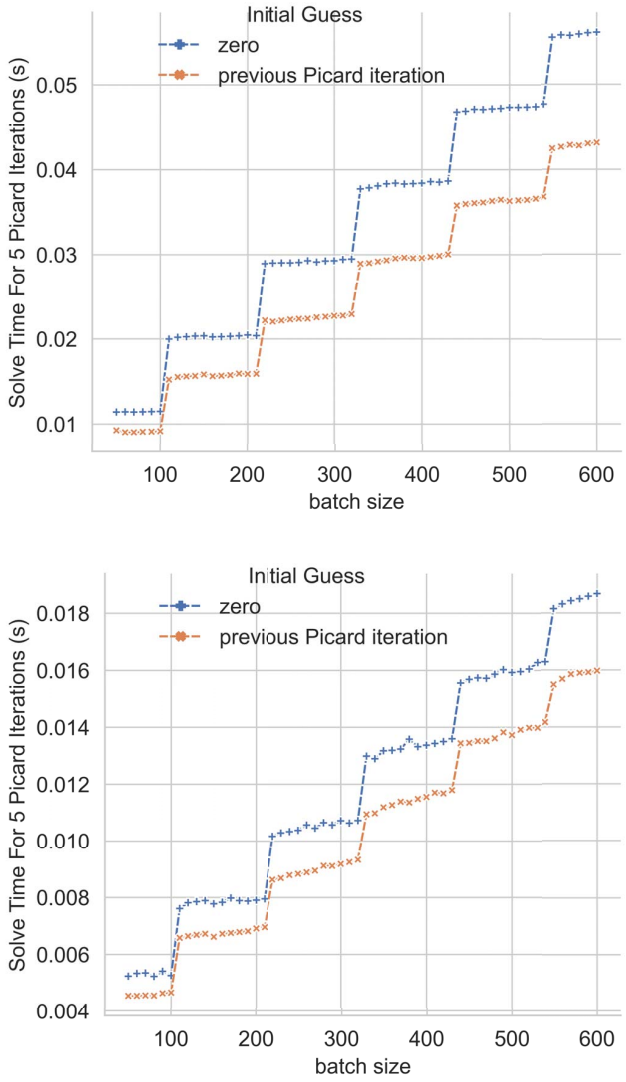


Fig. 8: Effect of using initial guess from the previous Picard iteration on total time to solution (cumulative over all the Picard iterations), *Top*: with **`BatchCsr`** format, *Bottom*: With **`BatchEll`** format.

VI. Conclusion

We have measured the performance of the GINKGO batched sparse iterative solvers for matrices representative of the electron and ion species on V100, A100 and MI100 GPUs and compared them against the Batched banded solvers on the CPU. The results suggest that the batched sparse iterative solvers in GINKGO efficiently utilize the GPU and being performance portable, are well suited for integration into main XGC. The results also underscore the importance of using an efficient sparse matrix format and the benefits of using batched iterative solvers over their batch direct counterparts.

The flexibility offered by GINKGO in terms of composing

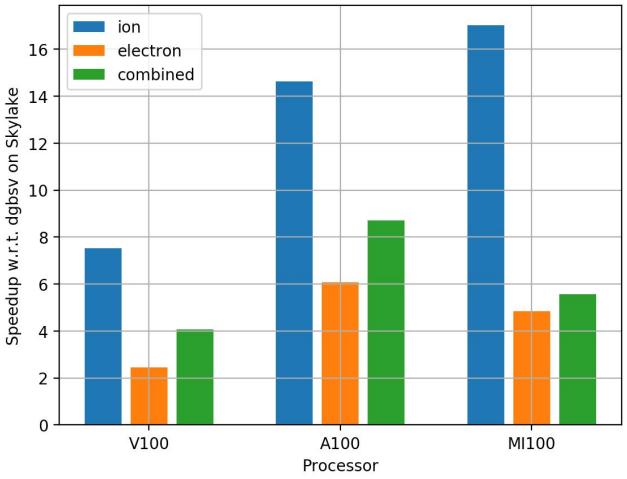


Fig. 9: Speedup for 5 Picard iterations using batched BiCGStab on GPUs over the banded solver on CPU

different batch solver components, while reducing the number of kernel launches allowing for maximum data reuse and increased occupancy, is important. Finally, using GINKGO iterative batched sparse solvers in conjunction with Kokkos would allow for seamless execution of XGC on exascale-oriented heterogeneous architectures at the various leadership supercomputing facilities. Therefore, future work includes tight integration of GINKGO into the main XGC using the flexible interfaces provided by the former, bringing it to production.

References

[1] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines. *ACM Transactions on Mathematical Software*, 47(3):21:1–21:23, June 2021.

[2] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. A set of batched basic linear algebra subprograms and LAPACK routines. *ACM Trans. Math. Softw.*, 47(3), June 2021.

[3] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software*, August 2020.

[4] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique 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, pages 1–10, New York, NY, USA, February 2017. Association for Computing Machinery.

[5] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 91–100, 2017.

[6] Enda Carroll, Andrew Gloster, Miguel D. Bustamante, and Lennon Ó' Náraigh. A batched GPU methodology for numerical solutions of partial differential equations. *arXiv*, 2107.05395, 2021.

[7] Advanced Micro Devices. MI100 white paper. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf.

[8] Tingxing Dong, Azzam Haidar, Piotr Luszczek, James Austin Harris, Stanimire Tomov, and Jack Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pages 157–160, Paris, France, August 2014. IEEE.

[9] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jonathon Hogg, Pedro Valero-Lara, Samuel D. Relton, Stanimire Tomov, and Mawussi Zounon. A Proposed API for Batched Basic Linear Algebra Subprograms. http://eprints.ma.man.ac.uk/2464/, April 2016.

[10] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[11] Nikolay M. Evstigneev, Oleg I. Ryabkov, and Eugene A. Tsatsorin. On the Inversion of Multiple Matrices on GPU in Batched Mode. *Supercomputing Frontiers and Innovations: an International Journal*, 5(2):23–42, June 2018.

[12] Andrew Gloster, Lennon Ó Náraigh, and Khang Ee Pang. cupentbatch—a batched pentadiagonal solver for NVIDIA GPUs. *Computer Physics Communications*, 241:113–121, 2019.

[13] Aditya Kashi, Pratik Nayak, Dhruva Kulkarni, Aaron Scheinberg, Paul Lin, and Hartwig Anzt. Ginkgo source snapshot and data for batched sparse iterative solvers on GPU for the collision operator for fusion plasma simulations, February 2022.

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

[15] NVIDIA. cuSOLVER - gpu accelerated library for decompositions and linear system solutions on NVIDIA GPUs. https://docs.nvidia.com/cuda/cusolver/index.html. Accessed: 2021-08-24.

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

[17] Pedro Valero-Lara, I. Martínez-Pérez, Raül Sirvent, X. Martorell, and Antonio J. Peña. cuThomasBatch and cuThomasVBatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 30, 2018.

[18] H. A. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, March 1992. Publisher: Society for Industrial and Applied Mathematics.

REPRODUCIBILITY APPENDIX

In order to ensure reproducibility of results, we provide the code as a Zenodo archive and elaborate on the settings and parameters used to produce these results. Since the XGC application is not available publicly, we focus on the reproducibility of the results in Figure 6.

Obtaining the source code and installing GINKGO

The source code and matrices are available via Zenodo [13] (https://doi.org/10.5281/zenodo.6280255). To build GINKGO and reproduce results in this paper, the following components are necessary:

1) The CMake build platform ($\geq$3.13).
2) A C++-14 compiler
3) A CUDA installation (CUDA $\geq$ 11.0 or NVHPC $\geq$ 21.7)
4) A ROCm installation (ROCM $\geq$ 4.2)

The Ginkgo library and the batched functionality use the same canonical CMake setup as elaborated in the Ginkgo documentation (https://ginkgo-project.github.io/ginkgo/doc/develop/install_ginkgo.html).

Benchmarking

The different flags available are explained on the benchmarking documentation page: https://ginkgo-project.github.io/ginkgo/doc/develop/benchmarking_ginkgo.html. To reproduce the results in this paper, the Zenodo archive contains the file `run_xgc_matrices.sh`. The header and job launcher in this script are specific to the Summit system, but apart from those, it works for other systems.

The matrices need to be in the Matrix Market format. The matrix class name (in this case, `dgb_2`) should be the main folder, with the matrices and the right hand sides in subfolders with the matrix index as the folder name (‘0’, ‘1’...). In the script, set the folder to benchmark: `export BATCH_MATRIX_FOLDER=/path/to/dgb_2`.

Our hardware and software

The NVIDIA V100 experiments were run on the Summit supercomputer at Oak Ridge National Laboratory in the United States. Each node of Summit consists of 6 NVIDIA V100 GPU’s connected to each other and the CPU sockets with NVLINK bridges. GCC 9.1.0 was used as the host compiler and CUDA 11.0.3 was used as the device compiler.

The NVIDIA A100 experiments were run on a system operated by the National Energy Reserch Scientific Computing Center (NERSC) for the United States government, using NVIDIA’s NVHPC 21.7 toolchain and on the Horeka cluster at Karlsruhe Institute of Technology, Germany using CUDA version 11.4.

The AMD MI100 experiments were run on a local cluster on one MI100 GPU. The system CPU is an AMD EPYC 7302. The ROCm framework version 4.2.0 was used.