Development of performance portable spline solver for exa-scale plasma turbulence simulation

Yuuichi Asahi
Université Paris-Saclay, UVSQ, CNRS, CEA
Maison de la Simulation
Gif-sur-Yvette, France
yuuichi.asahi@cea.fr

Baptiste Legouix
IRFM
CEA
St.Paul-lez-Durance, France

Emily Bourne
SCITAS
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland

Thomas Padioleau
Université Paris-Saclay, UVSQ, CNRS, CEA
Maison de la Simulation
Gif-sur-Yvette, France

Julien Bigot
Université Paris-Saclay, UVSQ, CNRS, CEA
Maison de la Simulation
Gif-sur-Yvette, France

Virginie Grandgirard
IRFM
CEA
St.Paul-lez-Durance, France

Kevin Obrejan
IRFM
CEA
St.Paul-lez-Durance, France

***Abstract*—This paper describes the development of performance portable spline building kernels on top of Kokkos-kernels. We wish to solve a single matrix equation with multiple right-hand sides. This problem is quite unique and thus neither Kokkos-kernels (direct method) nor Ginkgo (iterative methods) is optimized for this. We develop the required solvers in Kokkos-kernels with a batched serial implementation and optimize them using kernel fusion and sparse matrix storage. We demonstrate that our spline solver works efficiently on NVIDIA A100 and AMD MI250X GPUs, while keeping a reasonable performance on CPUs. This effort significantly reduces the development and maintenance cost of spline solvers on exa-scale supercomputing systems.**

***Index Terms*—Performance portability; Kokkos; Kokkos-kernels; Ginkgo; Splines;**

I. Introduction

Confronting the ever-increasing diversity in top-level supercomputers, performance portability is now considered as one of the most important requirements and challenges for High Performance Computing (HPC) applications. In order to minimize programming efforts when working on divergent supercomputers, application developers seek programming models that allow a single codebase to run efficiently on many architectures, and provide performance, portability, and productivity (P3). There are multiple approaches to provide performance portability over CPUs and GPUs including library-based, directive-based, and language-based approaches [1]–[9]. For application developers, it is also an important task to find a framework that fits best for the code development and maintenance strategy of a group.

Our primary task is to make an existing large scale plasma turbulence simulation code GYSELA [10] exa-scale ready. This code simulates the entire tokamak geometry by solving 5D Vlasov and 3D Poisson equations. Until recently, we have carefully explored a suitable performance portable programming model for this code using its mini-application [9], [11], [12]. Concerning the high performance portability, readability, and productivity, we decided to develop the newer version of GYSELA using Kokkos [5]. Among the performance portable frameworks such as RAJA [13], SYCL [14], and ALPAKA [15], we choose Kokkos [5] for the following reasons. Firstly, we have already an expertise on it [9], [11], [12] and now have a collaboration with core developers. Second, it has a better eco-systems including performance portable mathematical libraries. Finally, it is now under a technical project of the newly created High Performance Software Foundation (HPSF), representing a better sustainability as an open source project.

In addition to programming language, we also need performance portability in numerical libraries. As is often the case for production level simulation codes, GYSELA relies on a lot of libraries. For example, the Vlasov solver relies on BLAS [16] and LAPACK [17] for spline interpolation, and the Poisson solver relies on fftw [18], BLAS and LAPACK. BLAS and LAPACK are also used for the collision operator. For fast fourier transforms (FFTs), we have developed a FFT interface for Kokkos named Kokkos-FFT [19]. The collision operator [20] is implemented using Kokkos-kernels [21]. The remaining question is what libraries are suitable for a Vlasov solver using spline interpolation. From the numerical point of view, spline coefficients can be constructed by solving a linear system with multiple right-hand sides. This system involves different kinds of matrices including a general matrix, banded matrix, positive banded symmetric matrix and positive symmetric tridiagonal matrix, which can be solved efficiently

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00154

on CPUs with dedicated LAPACK functions **getrs**, **gbtrs**, **pbtrs** and **pttrs**, respectively. Thus, we need a performance portable linear algebra solver suitable for this problem to replace BLAS and LAPACK.

Spline interpolation is a powerful tool in numerical fluid simulations for several reasons including smoothness (continuity), accuracy, and stability. In addition, it is known that spline coefficients can be constructed efficiently on CPUs. Thus, spline interpolations have been widely used in scientific simulations. Unfortunately, GPU performance of spline interpolation has not been fully investigated except for image processing purposes [22], [23], and the performance portability aspect has not yet been discussed to the best of our knowledge. From the numerical point of view, we have two reasonable options to solve the linear system for spline constructions: direct and iterative methods. For direct methods, we try Kokkos-kernels [21] using the batched capabilities to handle multiple right-hand sides. For iterative methods, we employ Ginkgo performance portable iterative solvers [24]. After some consideration, we found that the Kokkos-kernels based approach is the most suitable for us since it is more flexible and memory efficient. Unfortunately, the LAPACK functions needed for splines are not implemented in Kokkos-kernels, and thus we integrated these functions into our fork of Kokkos-kernels [25] which we plan to upstream. We also demonstrate the optimizations that speed up the spline building kernel using Kokkos-kernels. We finally evaluate the performance portability of the developed spline solver. The main contributions of this work are as follows:

- Development of batched serial versions of matrix solvers (**getrs**, **gbtrs**, **pbtrs** and **pttrs**)for Kokkos kernels which are not frequently covered even by vendor libraries
- Development of a performance portable spline building kernel based on Kokkos kernels
- Optimization of the spline builder based on kernel fusion and sparsity of sub-matrices
- Summarizing the pros/cons of using Kokkos-kernels and Ginkgo for spline building

II. Splines and libraries

In this section, we describe the physical and numerical properties of splines. We also discuss the relevant libraries for spline building kernels.

A. Semi-Lagrangian scheme

GYSELA [10] employs backward semi-Lagrangian method to solve the advection term of the gyrokinetic Vlasov equation. Usually, the advection solver is a bottleneckand has been subject to optimizations [26]–[28]. Let us explain the Semi-Lagrangian scheme [29] with a simple 1D Vlasov equation, which is the advection equation in phase space.

$$\frac{\partial f}{\partial t} + v\left(x,t\right)\frac{\partial f}{\partial x} + a\left(x,v,t\right)\frac{\partial f}{\partial v} = 0, \tag{1}$$

where f is a distribution function, x is the spatial position, v is the velocity and a is the acceleration. In the Semi-Lagrangian scheme, Eq. (1) is considered to propagate the value of f along characteristic curves $\mathbf{\Gamma} = \mathbf{s}\left(t\right)$, where $\mathbf{\Gamma} = (x, v)$ is an Eulerian position vector in phase space. The Lagrangian characteristic $\mathbf{s}\left(t\right)$ is defined by

$$\dot{\mathbf{s}} = \mathbf{V}\left(\mathbf{s}, t\right), \mathbf{s}\left(0\right) = \mathbf{\Gamma}_0,$$

with the advection field $\mathbf{V}\left(v, a\right)$ and the initial position $\mathbf{\Gamma}_0$. In the Semi-Lagrangian scheme, the value of f at $(\mathbf{\Gamma}, t)$ is derived by following the characteristic curve back to the initial point $(\mathbf{\Gamma}_0, t_0)$. Using the known value of f at time t_n, we can evaluate $f\left(\mathbf{\Gamma}, t_{n+1}\right)$ by the backward characteristics scheme through the evaluation of the value of f at $\left(\mathbf{s}\left(t_n\right), t_n\right)$. With the first order time integral, the backward characteristics is approximated by

$$\mathbf{s}\left(t_n\right) \sim \mathbf{\Gamma} - \Delta t \mathbf{V}\left(\mathbf{\Gamma}, t_n\right),$$

with the previous speed $\mathbf{V}\left(\mathbf{\Gamma}, t_n\right)$. In order to evaluate $f\left(\mathbf{s}\left(t_n\right), t_n\right)$, we need some sort of interpolation scheme, where we employ spline interpolation because of its preferable physical properties. It should be remarked that the grid may not necessarily be uniform. In a new version of GYSELA, the non-uniform mesh is planned to be introduced in order to simulate the whole plasma including a region of steep gradients in equilibrium profile. These regions need finer resolutions than other regions, and thus non-uniform grids are necessary for the new GYSELA accompanied by non-uniform spline interpolation [30].

B. Spline construction

Let us consider 1D periodic B-splines. Spline coefficients can be calculated by solving the following linear system

$$A\mathbf{x} = \mathbf{b}, \tag{2}$$

where A is a spline matrix [31], $\mathbf{x}$ is a vector of spline coefficients and $\mathbf{b}$ is a vector containing the values to be interpolated. For example, the matrix A for degree 3 uniform splines has the sparsity pattern shown in Fig. 1. A is a fixed matrix in time and only $\mathbf{b}$ is time evolving. Higher dimensional B-splines can be obtained by a tensor product of 1D splines. For N-D splines, N equations in the form of equation (2) must be solved. Each of these equations handles one of the dimensions and behaves in the same way as the 1D case, batched over the other dimensions. It is therefore sufficient to consider the 1D case.

In GYSELA, we perform a batched 1D spline interpolation of the high dimensional distribution function along the dimension of interest. Remaining dimensions are regarded as batched dimensions which are embarrassingly parallel. This problem can be defined as solving equation (2) with a fixed single matrix A and multiple right-hand sides. This can be regarded as a batched problem. Assuming we have 10^3 grid points in each dimension and do not apply MPI decomposition, the number of batches can be $10^{12} = (10^3)^4$ corresponding to the total number of grid points in the remaining 4 dimensions and the matrix size is 10^3 equal to the number of grid points in the dimension of interest. The combinations of small matrix size

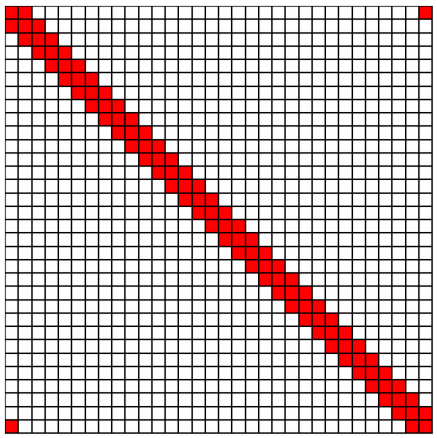

Fig. 1. Matrix A for degree 3 uniform splines.

and huge batch size stem from the high dimensional feature of GYSELA. After some investigation, it turns out that this use case is quite unique and most performance portable libraries are not optimized for this problem. In general, most of the batched solvers are optimized to deal with multiple matrices as well as multiple right-hand sides.

1) Direct method: To solve Eq. (2) with a direct method, we often rearrange the matrix A into the following sub-matrices, which are solved using Schur's complement [32].

$$A = \begin{pmatrix} Q & \gamma \\ \lambda & \delta \end{pmatrix} \tag{3}$$

A blockwise LU decomposition of this matrix gives

$$A = LU = \begin{pmatrix} Q & 0 \\ \lambda & \delta' \end{pmatrix} \begin{pmatrix} I & \beta \\ 0 & I \end{pmatrix}, \tag{4}$$

where δ' is the Schur's complement of Q defined as $\delta' = \delta - \lambda\beta$ and $\beta = Q^{-1}\gamma$. We can solve Eq. (2) in the following steps.

$$\begin{pmatrix} Q & 0 \\ \lambda & \delta' \end{pmatrix} \begin{pmatrix} \mathbf{x}'_0 \\ \mathbf{x}'_1 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \end{pmatrix}, \tag{5}$$

$$\begin{pmatrix} I & \beta \\ 0 & I \end{pmatrix} \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{x}'_0 \\ \mathbf{x}'_1 \end{pmatrix}, \tag{6}$$

where $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1)^T$ and $\mathbf{b} = (\mathbf{b}_0, \mathbf{b}_1)^T$. In summary, we get the following algorithm 1.

Algorithm 1 Solving spline matrix with Schur's complement

1: **Input:** A, $\mathbf{b}$, **Output:** $\mathbf{x}$
2: Solve $Q\mathbf{x}'_0 = \mathbf{b}_0$
3: Solve $\delta'\mathbf{x}_1 = \mathbf{b}_1 - \lambda\mathbf{x}'_0$
4: Compute $\mathbf{x}_0 = \mathbf{x}'_0 - \beta\mathbf{x}_1$

In order to use direct methods on GPU, we need to get the factorization of the matrix A on the device. As this matrix happens to be fixed in time and is small compared to the right-hand side, the factorization is only done once at initialization. Thus, we take advantage of existing CPU libraries to factorize the matrix and copy the result to the device. We notice that this simple strategy is efficient enough to consider this step as negligible compared to the solving step. In contrast, all the operations in algorithm 1 must be performed for each time step, and thus these must be implemented in a performance portable manner. In addition, the biggest sub-matrix Q can be a banded matrix which can be solved efficiently with specialized solvers. Table I shows the type of sub-matrix Q with different spline degree and uniformity. The sub-matrix δ' is a general matrix. To solve these matrices efficiently, we need **getrs** (General matrix), **gbtrs** (General banded matrix), **pbtrs** (Positive definite symmetric (PDS) banded matrix), and **pttrs** (PDS tridiagonal matrix) which are implemented in LAPACK.

TABLE I
TYPE OF SUB-MATRIX Q WITH DIFFERENT SPLINE DEGREES AND UNIFORMITY. DEDICATED LAPACK FUNCTIONS ARE PRESENTED IN PARENTHESES.

Degree	Uniform	Non-uniform
3	PDS tridiagonal (**pttrs**)	General banded (**gbtrs**)
4	PDS banded (**pbtrs**)	General banded (**gbtrs**)
5	PDS banded (**pbtrs**)	General banded (**gbtrs**)

2) Iterative method: Alternatively, we can also solve Eq. (2) with iterative methods such as **BiCG**, **BiCGStab**, **CG**, and **GMRES**. Contrary to the direct methods which need multiple specialized solvers for the choice of spline, we only need to prepare a single matrix solver. In addition, the interpolation matrix A is well conditioned [33] which is attractive in terms of accuracy and performance.

C. Relevant libraries

In this work, we employ two different linear algebra libraries Kokkos-kernels [21] and Ginkgo [24]. Both of these support multiple backends, and thus satisfy our needs for performance portability.

1) Kokkos-kernels: The Kokkos-kernels library [21] offers a performance portable linear algebra library on top of Kokkos [5]. It consists of 4 functionalities.

- Sparse Linear Algebra
- Dense Linear Algebra
- Graph Algorithms
- Batched Data Structures and Utilities

Among these, batched data structures and utilities attract our attention. The batched functionality offers BLAS and/or LAPACK that can be used inside a Kokkos parallel region. The advantage of this approach is that a user can compose various batched dense linear algebra kernels in a single parallel region, exploiting temporal locality. This feature is exactly what we want for the spline solver where we need to solve a small matrix for multiple right-hand sides.

- Pros
  If we have appropriate solvers, all the operations can be performed in-place which is highly memory efficient. In addition, this approach is quite flexible which makes room for optimization. In principle, we can integrate arbitrary Kokkos functions into matrix solvers, and thus adding some missing solvers is a relatively easy task.
- Cons
  The algorithms to solve the linear systems are intrinsically sequential. Thus, we can parallelize only along the batch direction. In addition, we need to prepare multiple solvers for each type of matrix, which makes the implementation complicated and costly.

2) Ginkgo: Ginkgo [24] is one of the most famous performance portable linear algebra libraries focusing on sparse computations. It features iterative solvers (BiCG, BiCGStab, CG, GMRES) and sophisticated preconditioners. Contrary to Kokkos-kernels, Ginkgo takes a hybrid approach for performance portability relying on a generic layer for simple core kernels and a specialized layer for performance-critical kernels. The generic layer has a similar concept to Kokkos, which can work on any platform. For key algorithms, they rely on the vendor specific languages to achieve the best performance. For our use case, pros and cons are as follows:

- Pros One of the most important advantages of Ginkgo is that it works for any arbitrary matrix A as long as it is solvable. We do not need to implement a solver for each matrix type. In addition, the solver is fully parallelized internally and Ginkgo can achieve competitive performance across different GPU architectures.
- Cons The memory consumption cannot be controlled by users. We encountered a memory shortage for some batch sizes, and thus we needed to pipeline along the batch direction (see subsection III-B). In addition, the maximum number of batches are limited to 65535 due to the hardware constraints for CUDA and HIP backends. It should be emphasized that this maximum number is insufficient for our needs (See subsection II-B).

D. Implementation strategies

After our investigation of Kokkos-kernels and Ginkgo, it turns out that both libraries do not fully satisfy our needs as is. Neither Kokkos-kernels nor Ginkgo expects a linear system of a small single matrix with a large number of batches. The Kokkos-kernels library uses parallelization to solve the matrix equation, indicating that it assumes a bigger matrix with a small number of batches. In addition, The Kokkos-kernels library does not include the functions we require, that is, **getrs**, **gbtrs**, **pbtrs**, and **pttrs**.

As Ginkgo can handle arbitrary matrices, we started with a Ginkgo implementation and used it as a reference. As a long term solution, we choose Kokkos-kernels for two reasons: First, we are developing the new version of GYSELA on top of Kokkos [5], which natively supports Kokkos-kernels. Second, the operations can be in-place and thus the implementation in the Kokkos-kernels library is more memory efficient, which is critical to our application. Accordingly, we decided to implement **getrs**, **gbtrs**, **pbtrs**, and **pttrs** as part of Kokkos-kernels batched functionalities and then optimize the code. Our implementations of these solvers are currently placed in our fork [25] but we plan to upstream them to Kokkos-kernels in the future. It should be noted that we optimize only the Kokkos-kernels implementation and do not optimize the Ginkgo implementation at all. Accordingly, there is a lot of room to improve the performance of the Ginkgo implementation.

III. Baseline implementation and benchmark application

In this section, we describe the implementation of spline solvers using Kokkos-kernels and Ginkgo.

A. Kokkos-kernels implementation

For Kokkos-kernels, we have started by implementing multiple solvers in **KokkosBatched** Serial format. Listing 1 shows the API (lines 1-8) and implementation details (lines 10-26) of **pttrs** which solves the positive symmetric tridiagonal matrix equation. The template parameters **ArgUplo** and **ArgAlgo** of the API are used to specify the matrix format (upper or lower tridiagnoal) and the usage of a cache blocking (line 1). The views "d" and "e" are diagonal and upper diagonal components of a matrix. Most importantly, this operation is in-place, that is, "b" stores the right-hand side vectors on entry and stores the solution vectors on exit. As with other solvers like **getrs**, this algorithm is strictly sequential against a matrix (lines 18-24). Accordingly, the parallelization can only be made against a batch dimension and thus a serial implementation is sufficient. Listing 2 shows the baseline implementation of algorithm 1 with Kokkos-kernels. Matrix-matrix operations like $\mathbf{b}_1 - \lambda \mathbf{x}'_0$ are performed with **KokkosBlas::gemm** (line 9). Although both **pttrs** (lines 1-8) and **getrs** (lines 10-19) are sequentially solved in a Kokkos parallel region, the batch size is about 10^6 which is sufficient to saturate GPUs. Although **KokkosBlas::gemm** can execute vendor libraries like cublas [34] and rocblas [35], we use the native Kokkos-kernels version in order to avoid the initialization costs of vendor libraries.

B. Ginkgo implementation

In the Ginkgo implementation, we rely on **BiCGStab** solver for GPU and **GMRES** for CPU because of a Ginkgo issue (#1563 [36]) with **BiCGStab** for OpenMP backend. The matrix is stored in CSR (Compressed Sparse Row) format. A block-Jacobi preconditioner is used with the **max_block_size** being tunable between 1 and 32. The tolerance is set to a reduction factor $\frac{\|Ax-b\|}{\|b\|} < 10^{-15}$. We first tried to apply it to all the right-hand sides, but it failed due to the large amount of memory usage. Thus, we pipelined along the batch direction to perform operations on smaller chunks as shown in listing 3 (lines 1-9). **m_cols_per_chunk** defines the chunk size which is set as 8192 for CPUs and 65535 for GPUs (line 1-2). The chunk of right-hand sides is copied to buffers (lines 24-25)

Listing 1. Internal **Pttrs** implementation

```
template <typename ArgUplo, typename ArgAlgo>
struct SerialPttrs {
  template <typename DViewType, typename EViewType,
            typename BViewType>
  KOKKOS_INLINE_FUNCTION static int invoke(const DViewType &d,
                                           const EViewType &e,
                                           const BViewType &b);
};

template <>
template <typename ValueType>
KOKKOS_INLINE_FUNCTION int
SerialPttrsInternal<Uplo::Lower, Algo::Pttrs::Unblocked>::invoke(
  const int n, const ValueType *KOKKOS_RESTRICT d, const int ds0,
  const ValueType *KOKKOS_RESTRICT e, const int es0,
  ValueType *KOKKOS_RESTRICT b, const int bs0, const int ldb) {
  // Solve A * X = B using the factorization L * D * L**T
  for (int i = 1; i < n; i++) {
    b[i * bs0] -= e[(i - 1) * es0] * b[(i - 1) * bs0];
  }
  b[(n - 1) * bs0] /= d[(n - 1) * ds0];
  for (int i = n - 2; i >= 0; i--) {
    b[i * bs0] = b[i * bs0] / d[i * ds0] - b[(i + 1) * bs0] * e[i * es0];
  }
  return 0;
}
```

Listing 2. Kokkos-kernels implementation

```
Kokkos::parallel_for(
  "KokkosBatched::SerialPttrs",
  batch,
  KOKKOS_LAMBDA(const int i) {
    auto sub_b0 = Kokkos::subview(b0, Kokkos::ALL, i);
    KokkosBatched::SerialPttrs<
      KokkosBatched::Algo::Pttrs::Unblocked>::invoke(d, e, sub_b0);
});
KokkosBlas::gemm(ExecSpace(), "N", "N", -1., bottom_left, b0, 1., b1);
Kokkos::parallel_for(
  "KokkosBatched::SerialGetrs",
  batch,
  KOKKOS_LAMBDA(const int i) {
    auto sub_b1 = Kokkos::subview(b1, Kokkos::ALL, i);
    KokkosBatched::SerialGetrs<
      KokkosBatched::Trans::NoTranspose,
      KokkosBatched::Algo::Getrs::Unblocked>::
      invoke(delta, ipiv_device, sub_b1);
});
```

which are packed into a contiguous 2D array before being passed to the Ginkgo solver (lines 28-29). After solving, the chunk of solutions are copied back (line 32) to the original right-hand sides.

C. 1D advection solver for benchmarks

For the benchmark, we employ a 1D advection solver using the Semi-Lagrangian scheme. This corresponds to solving the advection term along the x direction while using batching along the v_x direction in Eq. (1). This includes the entire procedures of spline interpolation: building splines and interpolation. Algorithm 2 describes the 1D batched advection solver with spline interpolation.

Here, f^n and f^{n+1} are respectively the distribution functions at time step t_n and t_{n+1}, $\eta(x_{i=*}, v_j)$ is the spline coefficient. In this algorithm, we need two transpose operations before and after the spline solver (line 3-5). Because the Ginkgo solver is only compatible with a contiguous row-major

Listing 3. Ginkgo implementation

```
std::size_t const main_chunk_size
    = std::min(m_cols_per_chunk, b.extent(1));
std::size_t const iend
    = (b.extent(1) + main_chunk_size - 1) / main_chunk_size;
for (std::size_t i = 0; i < iend; ++i) {
    std::size_t const subview_begin = i * main_chunk_size;
    std::size_t const subview_end
        = (i + 1 == iend) ? b.extent(1) :
        (subview_begin + main_chunk_size);

    auto const b_chunk
        = Kokkos::subview(b,
          Kokkos::ALL,
          Kokkos::pair(subview_begin, subview_end));
    auto const b_buffer_chunk = Kokkos::
        subview(b_buffer,
                Kokkos::ALL,
                Kokkos::pair(std::size_t(0),
                             subview_end - subview_begin));
    auto const x_chunk = Kokkos::subview(x,
        Kokkos::ALL,
        Kokkos::pair(std::size_t(0), subview_end - subview_begin));

    Kokkos::deep_copy(b_buffer_chunk, b_chunk);
    Kokkos::deep_copy(x_chunk, b_chunk);

    m_solver->add_logger(convergence_logger);
    m_solver->apply(to_gko_dense(gko_exec, b_buffer_chunk),
                    to_gko_dense(gko_exec, x_chunk));
    m_solver->remove_logger(convergence_logger);

    Kokkos::deep_copy(b_chunk, x_chunk);
}
```

Algorithm 2 1D batched advection with spline interpolation

1: **Input:** $f^n(x_i, v_j)$, **Output:** $f^{n+1}(x_i, v_j)$
2: **for all** All grid points (v_j) **do**
3: Transpose $f^n(x_{i=*}, v_j)$ to be contiguous $f^T(x_{i=*}, v_j)$
4: Solve $A\eta^T(x_{i=*}, v_j) = f^T(x_{i=*}, v_j)$ for $\eta^T(x_{i=*}, v_j)$
5: Transpose $\eta^T(x_{i=*}, v_j)$ to be original layout $\eta(x_{i=*}, v_j)$
6: **for all** All grid points (x_i) **do**
7: $(x_i)^* \leftarrow$ Foot of characteristic for one time step Δt that ends at (x_i, v_j)
8: Interpolate$f^n(x_i, v_j)$ at location $(x_i)^*$ using spline coefficients $\eta(x_{i=*}, v_j)$
9: $f^{n+1}(x_i, v_j) \leftarrow$ the interpolated value
10: **end for**
11: **end for**

layout, we need to pack the distribution function f into this layout before solving. Accordingly, we keep the data in a contiguous row-major layout for both CPUs and GPUs. Except for the matrix solver part implemented in Kokkos-kernels or Ginkgo, other operations are parallelized with Kokkos.

IV. Optimization for Kokkos-kernels Implementation

In this section, we have optimized the Kokkos-kernels implementation of spline building kernels. We have tried multiple optimization approaches while keeping performance portability. We have first a applied kernel fusion approach to improve

the data locality. We have then replaced the **gemv** function by **spmv** (Sparse matrix vector multiplication) based on the sparsity of sub-matrices. Finally, we evaluate performance improvements with optimizations. We have optimized the code on NVIDIA A100 GPU which is profiled with "NVIDIA Nsight systems" and "NVIDIA Nsight compute".

A. Settings for performance measurements

The source codes used in this study are publicly available on GitHub [37]. The benchmark of the entire application has been performed with benchmark [38] library, while the code is profiled by Kokkos-tools for optimizations. Performance has been measured on Intel Icelake [39], NVIDIA A100 [40] and AMD MI250X [41]. For AMD MI250X, we regard each Graphic Compute Die (GCD) as a single GPU. Table II summarizes the devices and compilers used in the present work. To measure performance on different architectures, we use Kokkos-tools [42], whose measurements are cross-checked with "NVIDIA Nsight compute" on A100. We use a single GPU for all the performance measurements in this work.

B. Baseline performance

Firstly, we have profiled the baseline implementation of spline building kernel in listing 2. The problem size is fixed as $(N_x, N_v) = (1000, 100000)$ with 10 iterations. This corresponds to the matrix size of 1000 and the batch size of 100000, which is our typical use case. It consists of 4 different kernels: **pttrs**, **gemm**, **getrs**, and **gemm**.

Analyzing the Gantt chart obtained from "NVIDIA Nsight systems", it turns out that the **pttrs** (taking 2.941 ms), and two **gemm** kernels (taking 3.795 ms and 4.423 ms) take roughly the same time. As expected, **getrs** (taking 6.496 µs) has negligible impact to the entire performance. We also investigate the detailed performance of **pttrs** using "NVIDIA Nsight compute". Considering that our problem size is $(n, batch) = (1000, 100000)$, we can neglect the memory footprint of the symmetric tridiagonal matrix kept in $((n-1) \times 2)$ format (where 2 comes from the diagonal and upper diagonal components). If load/store operations of multiple right-hand sides are fully cached, we expect the total memory access to be $0.8\text{GB} = (8 \times n \times batch)$ for double precision. Unfortunately, "Nsight compute" reports 1.58 GB load and 1.56 GB store operations from this kernel, indicating redundant memory accesses. At the same time, it reports the L1/TEX Hit Rate of 59.14 % and L2 Hit Rate of 57.37%, indicating that some data are actually cached. It is thus expected that the shared tridiagonal matrix is kept in cache, whereas the right-hand side vectors are not cached perfectly.

C. Improve data locality with kernel fusions

One of the most powerful features of Kokkos-kernels is the batched functionality [21]. It allows us to parallelize over the batch direction and perform multiple operations in a single parallel region. Thanks to this feature, we can define a single kernel to solve the entire matrix using **pttrs**, **gemv** and **getrs**. Listing 4 shows the spline building kernel with kernel fusion. Inside the parallel region, all the operations are performed on a single right-hand side (lines 5-6). With this change, **gemm** operations are now replaced with (batched) **gemv** as shown in lines 11-14 and 21-24. We expect improved hardware cache effects of the right-hand sides b0 and b1 in this implementation.

Listing 4. Kokkos-kernels (kernel fusion)

```
1  Kokkos::parallel_for(
2    "KokkosBatched::SerialPttrs−Gemv",
3    batch,
4    KOKKOS_LAMBDA(const int i) {
5      auto sub_b0 = Kokkos::subview(b0, Kokkos::ALL, i);
6      auto sub_b1 = Kokkos::subview(b1, Kokkos::ALL, i);
7
8      KokkosBatched::SerialPttrs<
9        KokkosBatched::Algo::Pttrs::Unblocked>::invoke(d, e, sub_b0);
10
11     KokkosBatched::SerialGemv<
12       KokkosBatched::Trans::NoTranspose,
13       KokkosBatched::Algo::Gemv::Unblocked>::
14         invoke(−1.0, bottom_left_block, sub_b1, 1.0, sub_b1);
15
16     KokkosBatched::SerialGetrs<
17       KokkosBatched::Trans::NoTranspose,
18       KokkosBatched::Algo::Getrs::Unblocked>::
19         invoke(bottom_right_block, bottom_right_piv, sub_b1);
20
21     KokkosBatched::SerialGemv<
22       KokkosBatched::Trans::NoTranspose,
23       KokkosBatched::Algo::Gemv::Unblocked>::
24         invoke(−1.0, top_right_block, sub_b1, 1.0, sub_b0);
25   });
```

"Nsight compute" reports 3.16 GB load and 2.37 GB store operations from this merged kernel. This indicates that the newly introduced **gemv** and **getrs** induce additional data load and store. At the same time, it reports almost the same L1/TEX Hit Rate and L2 Hit Rate of 56.32 % and 52.28%, indicating that tridiagonal matrix is still kept in cache.

D. Benefit from sparse sub-matrices

Although **gemm** operations are applied to the bottom-left and top-right sub-matrices, these matrices are largely sparse. For example, the top-right corner matrix with the shape of $(999, 1)$ contains 48 non-zeros and the bottom-left corner matrix with the shape of $(1, 999)$ contains 2 non-zeros. The matrix sparsity can differ for different spline orders and uniformity, but they are still highly sparse. In order to avoid implementing kernels for both CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats, we store sparse matrices in a COO (COOrdinate list) format. Listing 5 shows the data structure to handle COO storage inside a Kokkos parallel region. This class stores the non-zero elements in **m_values** (line 6) whose coordinates are stored in **m_rows_idx** and **m_cols_idx** (line 5). All the methods in this class are available inside Kokkos parallel region with KOKKOS_FUNCTION macro (lines 13-18). Using COO storage, the sequential loop to compute the multiplication can be reduced to the size of non-zeros (nnz), which drastically reduces the number of operations.

Listing 6 shows the Kokkos implementation of **spmv** using COO storage (lines 11-15 and 22-26). Although we need to

TABLE II
HARDWARE DESCRIPTION FOR ONE PROCESSOR. THERMAL DESIGN POWER (TDP) IS EXTRACTED FROM VENDORS DATA-SHEETS [39]–[41].

Processor	Intel Xeon Gold 6346 (Icelake)	NVIDIA A100 (A100)	AMD MI250X (MI250X)
Number of cores (FP64)	32	3456	-
Shared Cache [MB]	36	40	16/2
Peak performance [GFlops]	3174.4	9700	26500
Peak B/W [GB/s]	204.8	1555	1600
B/F ratio	0.064	0.160	0.060
SIMD width	512 bit	-	-
Warp/wavefront size	-	32	64
TDP [W]	205	400	500 / 2
Manufacturing process [nm]	10	7	6
Year	2021	2020	2021
Compilers	gcc 11.0	CUDA/12.2.128	rocm 5.7.0

Listing 5. COO storage class

```
struct Coo {
  std::size_t m_nrows = 0, m_ncols = 0;
  using IdxType = Kokkos::View<int*, Kokkos::LayoutRight>;
  using ValueType = Kokkos::View<double*, Kokkos::LayoutRight>;
  IdxType m_rows_idx, m_cols_idx;
  ValueType m_values;

  Coo(std::size_t const nrows, std::size_t const ncols,
      IdxType rows_idx, IdxType cols_idx, ValueType values)
    : m_nrows(nrows), m_ncols(ncols), m_rows_idx(std::move(rows_idx)),
      m_cols_idx(std::move(cols_idx)), m_values(std::move(values)) {}

  KOKKOS_FUNCTION std::size_t nnz() { return m_values.size(); }
  KOKKOS_FUNCTION std::size_t nrows() { return m_nrows; }
  KOKKOS_FUNCTION std::size_t ncols() { return m_ncols; }
  KOKKOS_FUNCTION IdxType rows_idx() { return m_rows_idx; }
  KOKKOS_FUNCTION IdxType cols_idx() { return m_cols_idx; }
  KOKKOS_FUNCTION ValueType values() { return m_values; }
};
```

iterate over all the elements of right-hand sides, we only need to perform matrix-matrix multiplication over the non-zero elements, leading to the order of reduction in operations. With this optimization, "Nsight compute" reports 1.60 GB load and 1.59 GB store operations. The additional memory accesses from **gemv** are almost suppressed. In addition, the L1/TEX Hit Rate and L2 Hit Rate have been increased to 59.79 % and 57.71%.

E. Impact of optimizations

The impacts of optimizations on Icelake, A100 and MI250X are summarized in Table III. With the kernel fusion, we got the speed-ups of $1.30\times$, $2.25\times$ and $1.42\times$ are obtained on Icelake, A100 and MI250X, respectively. The larger speed up on A100 than MI250X with kernel fusion can be explained by a larger cache of A100. Focusing on the sparsity of sub-matrices, we replace the dense **gemv** kernel with the **spmv** kernel, and the speed-ups of $1.37\times$, $1.70\times$ and $3.52\times$ are obtained on Icelake, A100 and MI250X, respectively. The larger speed up on MI250X indicates that the **gemv** kernel is a bottleneck on MI250X. We have also tried to store the tridiagonal matrix in read-only memory using **Kokkos::RandomAccess** trait.

Listing 6. Kokkos-kernels (**spmv**)

```
Kokkos::parallel_for(
  "KokkosBatched::SerialPttrs-Spmv",
  batch,
  KOKKOS_LAMBDA(const int i) {
    auto sub_b0 = Kokkos::subview(b0, Kokkos::ALL, i);
    auto sub_b1 = Kokkos::subview(b1, Kokkos::ALL, i);

    KokkosBatched::SerialPttrs<
      KokkosBatched::Algo::Pttrs::Unblocked>::invoke(d, e, sub_b0);

    for (int nz_idx = 0; nz_idx < bottom_left_block.nnz(); ++nz_idx) {
      const int r = bottom_left_block.rows_idx()(nz_idx);
      const int c = bottom_left_block.cols_idx()(nz_idx);
      sub_b1(r) -= bottom_left_block.values()(nz_idx) * sub_b0(c);
    }

    KokkosBatched::SerialGetrs<
      KokkosBatched::Trans::NoTranspose,
      KokkosBatched::Algo::Getrs::Unblocked>::
        invoke(bottom_right_block, bottom_right_piv, sub_b1);

    for (int nz_idx = 0; nz_idx < top_right_block.nnz(); ++nz_idx) {
      const int r = top_right_block.rows_idx()(nz_idx);
      const int c = top_right_block.cols_idx()(nz_idx);
      sub_b0(r) -= top_right_block.values()(nz_idx) * sub_b1(c);
    }
  });
```

TABLE III
IMPACT OF OPTIMIZATION ON ICELAKE, NVIDIA A100 GPUS AND AMD MI250X.

	Icelake	A100	MI250X
Original	145.8 ms	11.39 ms	16.14 ms
Kernel fusion	112.1 ms	5.06 ms	11.34 ms
gemv $\rightarrow$ **spmv**	82.0 ms	2.98 ms	3.22 ms

Unfortunately, it has a negligible impact, so we do not report it in this paper.

V. BENCHMARK FOR SPLINE BUILDING KERNELS WITH ITERATIVE AND DIRECT METHODS

In this section, we evaluate the performance of the 1D batched advection code for different batch sizes. The perfor-

mance comparison between the Kokkos-kernels and Ginkgo approaches are made. Then, we investigate the performance portability of the developed spline building kernels. We also discuss the benefit of Kokkos-kernels and Ginkgo approaches.

A. Benchmark of the entire 1D advection solver

We have measured the performance with a benchmark code which performs one time step of 1D batched advection using the Semi-Lagrangian method. As described in Algorithm 2, this benchmark includes the entire procedure of spline interpolation: building splines and interpolation. For a performance metric, we use Giga Lattice Update per Seconds (GLUPS) defined by

$$\text{GLUPS} = N_x \times N_v \times 10^{-9}/t, \tag{7}$$

where $N_x \times N_v$ is the problem size, and t is the total elapsed time in seconds. We have measured the performance of uniform and non-uniform splines with degrees of 3, 4 and 5. For the problem size, we fixed $N_x = 1024$ and scanned with respect to N_v for $(100 - 100000)$.

Figure 2 shows the achieved GLUPS with Kokkos-kernels and Ginkgo on CPUs and GPUs. We have significantly better performance with Kokkosl-kernels than Ginkgo for almost all regimes. In the Kokkos-kernels implementation, we have slightly better performance with uniform splines than non-uniform splines. The best performance has been obtained with degree 3 splines for a uniform mesh. The relatively low performance on CPUs stems from the non-ideal data layout. In the current implementation, the parallelization is made over the contiguous dimension. For a better cache usage, it is ideal to parallelize over the non-contiguous dimension, i.e., the batch dimension should be the non-contiguous dimension. This requires a layout abstraction which remains as a future work.

In the Ginkgo implementation, we use **GMRES** for CPUs and **BiCGstab** for GPUs. As discussed in subsection III-B, we set the chunk size to be 8192 for CPUs and 65535 for GPUs. Since we solve the time evolution of a simple 1D advection, the solution of the previous time step should be a good initial guess for the subsequent solve. The achieved performance is roughly the same for uniform and non-uniform splines (red and blue curves overlap). The performance degrades with higher spline degrees on GPUs which stem from the increments in the number of iterations. Table IV shows the number of iterations for the problem size $(N_x, N_v) = (1000, 100000)$. The number of iterations for each chunk remains constant in this case. The performance degradation with respect to the number of iterations implies that the performance bottleneck is the spline building kernels in the Ginkgo implementation.

B. Performance portability of spline building kernels in Kokkos-kernels

To understand the efficiency of the developed spline building kernels in Kokkos-kernels, we evaluate the achieved memory bandwidth with the following equation

$$\text{Bandwidth} = N_x \times N_v \times 8/t,$$

TABLE IV
THE NUMBER OF ITERATIONS IN GINKGO SOLVER FOR THE PROBLEM SIZE $(N_x, N_v) = (1000, 100000)$.

	GMRES	**BiCGStab**
uniform (Degree 3)	17	10
uniform (Degree 4)	22	14
uniform (Degree 5)	30	21
non-uniform (Degree 3)	24	14
non-uniform (Degree 4)	32	21
non-uniform (Degree 5)	41	28

where $N_x \times N_v$ is the problem size fixed as 1000×100000, and t is the total elapsed time in seconds profiled with Kokkos-tools. The number 8 represents one load/store operation of double precision data of right-hand sides in bytes, assuming a perfect and unlimited cache. Ignoring the read operations of a single matrix is probably oversimplification for CPUs, since the current data layout disturbs caching on CPUs. This is, however, a reasonable assumption for GPUs, since the caching of the matrix is implied by profilers (See IV).

In the present work, we employ the performance portability metric [43], [44] defined as

$$\mathcal{P}(a, p, H) = \begin{cases} \dfrac{|H|}{\displaystyle\sum_{i \in H} \dfrac{1}{e_i(a, p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \tag{8}$$

$$e_i(a, p) = \frac{P_{a,p,i}}{R_{a,i}} \times 100\%. \tag{9}$$

Here, $e_i(a, p)$ denotes the architectural efficiency of an application a on the device i with simulation settings p based on the roofline model [45]. $P_{a,p,i}$ is the achieved GFlops of the kernel a on the device i based on the hand count metrics of kernels. The attainable performance $R_{a,i}$ is computed by the roofline model [45] as

$$R_{a,i} = \min(F_i, B_i \times f_a/b_a), \tag{10}$$

where F_i and B_i are the Peak Floating point Performance in GFlops and the Peak Memory Bandwidth in GBytes/s of the device i, which are found in Table II. f_a and b_a denote the total number of floating point operations and memory accesses per grid point of a kernel a, counted by hand. H represents a set of platforms including Icelake, A100, and MI250X. A low value of $\mathcal{P}(a, p, H)$ indicates the achieved performance on each architecture is insufficient. All the evaluated kernels here are memory bound and thus the achieved memory bandwidth represents the roofline.

Table V shows the obtained performance for different types of splines. On Icelake, the performance degrades with the higher spline degrees. Unexpectedly, we obtain a better performance with the non-uniform splines compared to the uniform splines of degrees 4 and 5. As we have discussed, the data layout is not ideal for CPUs and thus the performance

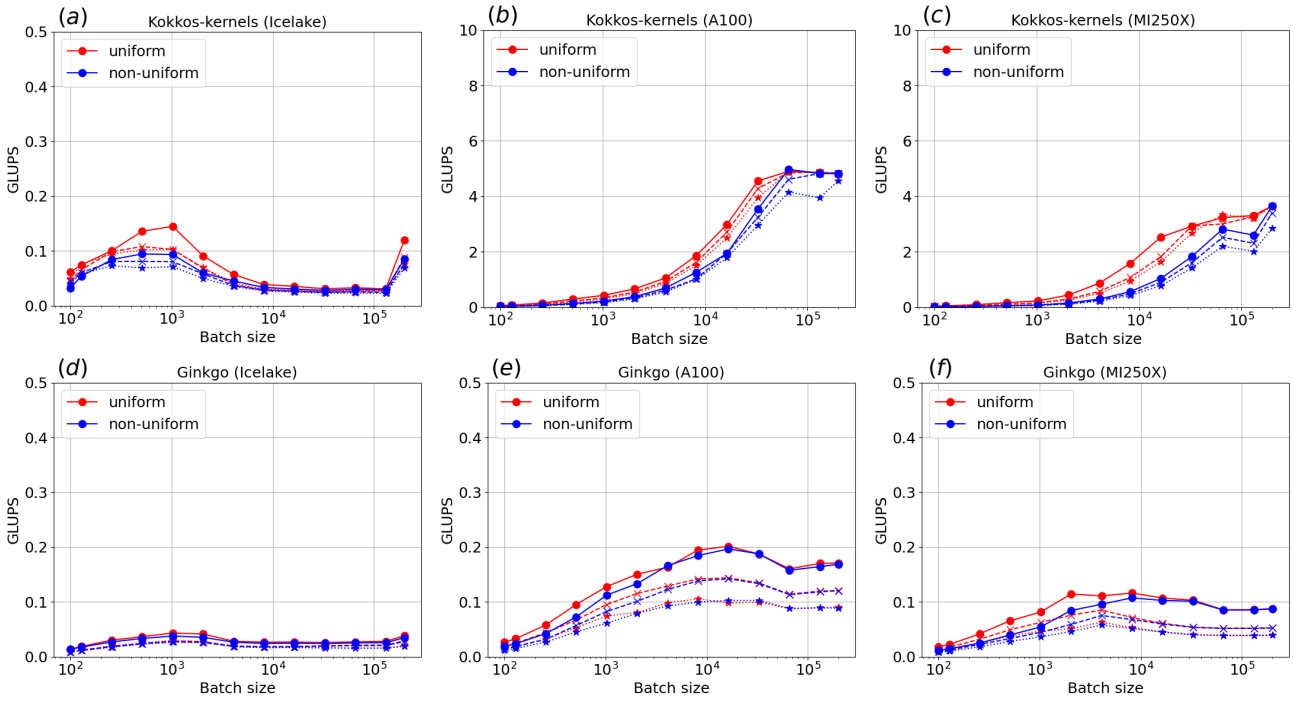

Fig. 2. Achieved performance of 1D batched advection with Kokkos-kernels (Top row) and Ginkgo (Bottom row) on Icelake (a, d), A100 (b, e) and MI250X (c, f). The red and blue colors correspond to the uniform and non-uniform mesh, respectively. The solid (with circle), dashed (with cross), and dotted (with star) lines represent spline degrees of 3, 4 and 5.

TABLE V
OBTAINED PERFORMANCE AND PERFORMANCE PORTABILITY OF EACH SPLINE BUILDING KERNEL. THE ACHIEVED BANDWIDTH ON EACH ARCHITECTURE AND RATIO TO THE PEAK BANDWIDTH IS SHOWN.

	Icelake	A100	MI250X	$\mathcal{P}(a,p,H)$
uniform (Degree 3)	9.75 GB/s (4.38 %)	268.6 GB/s (17.3 %)	247.8 GB/s (15.5 %)	0.086
uniform (Degree 4)	3.83 GB/s (1.87 %)	252.6 GB/s (16.2 %)	154.6 GB/s (9.7 %)	0.043
uniform (Degree 5)	3.83 GB/s (1.87 %)	251.3 GB/s (16.1 %)	153.5 GB/s (9.6 %)	0.043
non-uniform (Degree 3)	5.37 GB/s (2.62 %)	208.4 GB/s (13.4 %)	123.5 GB/s (7.7 %)	0.051
non-uniform (Degree 4)	5.15 GB/s (2.52 %)	169.9 GB/s (10.9 %)	81.8 GB/s (5.1 %)	0.044
non-uniform (Degree 5)	4.96 GB/s (2.42 %)	142.2 GB/s (9.15 %)	59.2 GB/s (3.7 %)	0.038

is not optimal. Further investigations are needed for CPU performance which remains as future work. On A100, the uniform splines can achieve high memory bandwidth. In contrast, the performance degrades in non-uniform splines. We may introduce a cache blocking version of **gbtrs** to improve performance. A similar trend has been found for MI250X, but the performance degradation compared to the uniform degree 3 spline is more critical than on A100.

C. Discussions for each approach

- Kokkos-kernels If performance is the top priority, we recommend the Kokkos-kernels approach. This approach is memory efficient since the spline building kernels can be in-place. From the code maintainability, we need to implement multiple solvers for each type of matrix. C++ polymorphism allows us to use a common interface in the host code, but polymorphism is not fully available for device kernels. Further optimizations may be possible by fusing transpose kernels with spline building kernels.
- Ginkgo The most important benefit of Ginkgo is code maintainability and readability. A dedicated solver for a single matrix and multiple right-hand sides should exhibit better performance with lower memory consumption.

VI. SUMMARY

In this paper, we demonstrate performance portable spline solvers based on Kokkos-kernels. For this purpose, we have developed batched serial versions of matrix solvers (corresponding to **getrs**, **gbtrs**, **pbtrs** and **pttrs** in LAPACK) as Kokkos-kernels batched solvers. Using these solvers, we have developed performance portable spline building kernels for uniform and non-uniform splines. We have optimized the spline building kernels based on kernel fusion and matrix sparsity to achieve reasonable performance on both NVIDIA A100

and AMD MI250X GPUs. This effort significantly reduces the development and maintenance cost of spline solvers dedicated to exa-scale supercomputing systems.

ACKNOWLEDGMENT

We acknowledge the financial support of the Cross-Disciplinary Program on "Numerical simulation" of CEA, the French Alternative Energies and Atomic Energy Commission. This work has also been supported by the CExA project of CEA. This work was carried out using FUJITSU PRIMERGY GX2570 (Wisteria/BDEC-01) at The University of Tokyo. This work was partly supported by JHPCN project jh230037.This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was also granted access to the HPC resources of CINES under the allocation 2023-cin4492 made by GENCI. The author Y.A thanks Dr. Luc Berger-Vergiat for helping us to upstream developed solvers to Kokkos-kernels.

REFERENCES

[1] P. Grete, F. W. Glines, and B. W. O'Shea, "K-athena: a performance portable structured grid finite volume magnetohydrodynamics code," *CoRR*, vol. abs/1905.04341, pp. 211 – 231, 2019. [Online]. Available: http://arxiv.org/abs/1905.04341

[2] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An overview of performance portability in the uintah runtime system through the use of kokkos," in *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, New York, NY, USA, Nov 2016, pp. 44–47. [Online]. Available: http://doi.acm.org/10.1109/ESPM2.2016.012

[3] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 11, p. e5640, 2020, e5640 cpe.5640. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5640

[4] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and S. A. Jarvis, "Performance portability of an unstructured hydrodynamics mini-application," in *Proceedings of 2018 International Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*. New York, NY, USA: ACM, 2018.

[5] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[6] E. M. Rangel, S. J. Pennycook, A. Pope, N. Frontiere, Z. Ma, and V. Madananth, "A performance-portable sycl implementation of crk-hacc for exascale," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1114–1125. [Online]. Available: https://doi.org/10.1145/3624062.3624187

[7] I. Z. Reguly, "Evaluating the performance portability of sycl across cpus and gpus on bandwidth-bound applications," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1038–1047. [Online]. Available: https://doi.org/10.1145/3624062.3624180

[8] J. Latt, C. Coreixas, and J. Beny, "Cross-platform programming model for many-core lattice Boltzmann simulations," *PLOS ONE*, vol. 16, no. 4, pp. 1–29, 04 2021. [Online]. Available: https://doi.org/10.1371/journal.pone.0250306

[9] Y. Asahi, T. Padioleau, G. Latu, J. Bigot, V. Grandgirard, and K. Obrejan, "Performance portable vlasov code with c++ parallel algorithm," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 68–80.

[10] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier *et al.*, "A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations," *Computer Physics Communications*, vol. 207, pp. 35 – 68, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465516301230

[11] Y. Asahi, G. Latu, V. Grandgirard, and J. Bigot, "Performance portable implementation of a kinetic plasma simulation mini-app," in *Accelerator Programming Using Directives*, ser. series, S. Wienke and S. Bhalachandra, Eds. Cham: Springer International Publishing, 2020, pp. 117–139.

[12] Y. Asahi, G. Latu, J. Bigot, and V. Grandgirard, "Optimization strategy for a performance portable vlasov code," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 79–91.

[13] Hornung, Richard D. and Keasler, Jeffrey A., "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep., 9 2014.

[14] R. Reyes, G. Brown, R. Burns, and M. Wong, "Sycl 2020: More than meets the eye," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388649

[15] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, and M. Bussmann, "Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library," Jun 2017. [Online]. Available: https://arxiv.org/abs/1706.10086

[16] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[17] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[19] *A shared-memory FFT for the Kokkos ecosystem*, "https://github.com/kokkos/kokkos-fft", Last accessed on 2024-07-24.

[20] *A KOkkos based colLIsion OPerator (KoLiOp)*, "https://gitlab.com/cines/code.gysela/libkoliop", Last accessed on 2024-07-30.

[21] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," 2021. [Online]. Available: https://arxiv.org/abs/2103.11991

[22] F. Champagnat and Y. L. Sant, "Efficient cubic b-spline image interpolation on a gpu," *Journal of Graphics Tools*, vol. 16, no. 4, pp. 218–232, 2012. [Online]. Available: https://doi.org/10.1080/2165347X.2013.824736

[23] T. Briand and A. Davy, "Optimization of Image B-spline Interpolation for GPU Architectures," *Image Processing On Line*, vol. 9, pp. 183–204, 2019, https://doi.org/10.5201/ipol.2019.257.

[24] T. Cojean, Y.-H. M. Tsai, and H. Anzt, "Ginkgo—a math library designed for platform portability," *Parallel Computing*, vol. 111, p. 102902, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819122000096

[25] *kokkos-kernels*, "https://github.com/yasahi-hpc/kokkos-kernels", Last accessed on 2024-07-30.

[26] G. Latu, M. Haefele, J. Bigot, V. Grandgirard, T. Cartier-Michaud, and F. Rozar, "Evaluating kernels on xeon phi to accelerate gysela application," *ESAIM: Proc.*, vol. 53, pp. 211 – 231, 2016. [Online]. Available: https://doi.org/10.1051/proc/201653013

[27] Y. Asahi, G. Latu, T. Ina, Y. Idomura, V. Grandgirard, and X. Garbet, "Optimization of fusion kernels on accelerators with indirect or strided

memory access patterns," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1974–1988, 2017.
[28] G. Latu, Y. Asahi, J. Bigot, T. Feher, and V. Grandgirard, "Scaling and optimizing the gysela code on a cluster of many-core processors," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 466–473.
[29] E. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo, "The semi-lagrangian method for the numerical resolution of the vlasov equation," *Journal of Computational Physics*, vol. 149, no. 2, pp. 201–220, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999198961484
[30] E. Bourne, Y. Munschy, V. Grandgirard, M. Mehrenberger, and P. Ghendrih, "Non-uniform splines for semi-lagrangian kinetic simulations of the plasma sheath," *Journal of Computational Physics*, vol. 488, p. 112229, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999123003248
[31] W. J. Gordon and R. F. Riesenfeld, "B-spline curves and surfaces," in *Computer Aided Geometric Design*, R. E. BARNHILL and R. F. RIESENFELD, Eds. Academic Press, 1974, pp. 95–126. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780120790500500114
[32] F. Zhang, *The Schur Complement and its Applications*, ser. Numerical Methods and Algorithms. New York: Springer, 2005, vol. 4.
[33] E. W. Cheney and D. R. Kincaid, *Numerical Mathematics and Computing*, 6th ed. USA: Brooks/Cole Publishing Co., 2007.
[34] NVIDIA, "cublas library user's guide," July 2024, https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf, Last accessed on 2024-07-30.
[35] Advanced Micro Devices, "rocBLAS Documentation," June 2024, https://rocm.docs.amd.com/projects/rocBLAS/en/latest/, Last accessed on 2024-07-30.
[36] *BiCGStab OMP issue*, "https://github.com/ginkgo-project/ginkgo/issues/1563", Last accessed on 2024-08-08.
[37] *ddc (P3HPC 2024 Artifact)*, "https://github.com/yasahi-hpc/ddc", Last accessed on 2024-09-26.
[38] *A microbenchmark support library*, "https://github.com/google/benchmark", Last accessed on 2024-08-01.
[39] Intel, "IntelXeon Gold 6346 Processor (36 M Cache, 3.10 GHz)," https://www.intel.com/content/www/us/en/products/sku/212457/intel-xeon-gold-6346-processor-36m-cache-3-10-ghz/specifications.html.
[40] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.
[41] AMD, "AMD CDNA™ 2 ARCHITECTURE," https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf, 2021.
[42] *Kokkos C++ Performance Portability Programming Ecosystem: Profiling and Debugging Tools*, "https://github.com/kokkos/kokkos-tools", Last accessed on 2024-07-31.
[43] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947 – 958, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17300559
[44] S. J. Pennycook and J. D. Sewall, "Revisiting a metric for performance portability," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 1–9.
[45] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [DEVELOPMENT OF PERFORMANCE PORTABLE SPLINE SOLVER FOR EXA-SCALE PLASMA TURBULENCE SIMULATION]

A. Abstract

This appendix is the artifact description of the paper entitled "Development of performance portable spline solver for exa-scale plasma turbulence simulation". It includes the basic instruction for compilation and experiment workflows. The expected results and the data evaluation method in the paper are also demonstrated.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: Semi-Lagrangian method, Spline interpolation**
- **Program:** `ddc` **[https://github.com/yasahi-hpc/ddc] (SHA: ac4e774)**
- **Compilation: See "Compilation" section.**
- **Run-time environment: "Experiment workflow"**
- **Hardware: Intel Xeon Gold 6346 (Icelake), AMD MI250X, and A100 (PCIE 40GB).**
- **Execution: See "Experiment workflow" section.**
- **Output: See "Evaluation and expected result" section.**
- **Experiment workflow: See "Experiment workflow" section.**
- **Publicly available?: Yes**

2) How software can be obtained (if available): The source codes are publicly available on the GitHub page https://github.com/yasahi-hpc/ddc. (branch: kokkos-kernels-profile, SHA: e3ef33b). The source codes for profiling are "examples/characteristics_advection.cpp" and "benchmarks/s-plines.cpp".

3) Hardware dependencies: We have tested on Intel Xeon Gold 6346 (Icelake), NVIDIA A100, and AMD MI250X GPUs.

4) Software dependencies: This software relies on external libraries including Kokkos [https://github.com/kokkos/kokkos], `mdspan` [https://github.com/kokkos/mdspan], Kokkos-kernels [https://github.com/kokkos/kokkos-kernels], googletest [https://github.com/google/googletest], benchmark [https://github.com/google/benchmark], LAPACK [https://github.com/Reference-LAPACK/lapack], Kokkos-tools [https://github.com/kokkos/kokkos-tools] and Ginkgo [https://github.com/ginkgo-project/ginkgo]. Kokkos, `mdspan`, Kokkos-kernels, googletest, and benchmark are included as submodules. LAPACK, Ginkgo, and Kokkos-tools are assumed to be installed to the system.

C. Installation

First of all, we have to git clone DDC in the following manner.

```
git clone --recursive \
https://github.com/yasahi-hpc/ddc.git
cd ddc
git switch kokkos-kernels-profile
```

Then, we configure and build DDC with the following commands.

```
cmake -B build \
-DCMAKE_CXX_COMPILER=<compiler_name> \
-DCMAKE_BUILD_TYPE=Release \
-DDDC_BUILD_KERNELS_FFT=OFF \
-DDDC_BUILD_KERNELS_SPLINES=ON \
-DDDC_BUILD_PDI_WRAPPER=OFF \
-DDDC_BUILD_BENCHMARKS=ON \
-DDDC_SPLINES_SOLVER=GINKGO \ # or LAPACK
-DDDC_SPLINES_VERSION=0 # 0, 1, 2 or None
```

```
<COMMANDS_FOR_KOKKOS>
cmake --build build -j 16
```

The compilation commands for each architecture <COMMANDS_FOR_KOKKOS> for Icelake, A100 and MI250X are as follows.

```
# Icelake (compiler_name: g++)
-DKokkos_ENABLE_OPENMP=ON \
-DKokkos_ARCH_SKX=ON

# A100 (compiler_name: g++)
-DKokkos_ENABLE_CUDA=ON \
-DKokkos_ARCH_AMPERE80=ON

# MI250X (compiler_name: hipcc)
-DKokkos_ENABLE_HIP=ON \
-DKokkos_ARCH_AMD_GFX90A=ON
```

D. Experimental workflow for "Optimization for Kokkos-kernels implementation" (section IV)

1) Compilation: For the compilation of baseline, we set `-DDDC_SPLINES_SOLVER=LAPACK` and compile without `-DDDC_SPLINES_VERSION` for baseline. For kernel-fusion and **spmv**, we compile with `-DDDC_SPLINES_VERSION=1` and `2`, respectively.

2) Evaluation and expected results: We execute the benchmark `app` (`build/examples/characteristics_advection`) in the following way. The first and second arguments to the executable are the non-uniformity of mesh and degree of splines.

```
export tools_dir=<path-to-kokkos-tools>
#profile with nsys
nsys profile .app 0 3 \
--kokkos-tools-libs=\
${tools_dir}/libkp_nvtx_connector.so
ncu -f --set full -o profile .app 0 3 \
--kokkos-tools-libs=\
${tools_dir}/libkp_nvtx_connector.so

# Just run with Kokkos-tools
export PATH=${PATH}:${tools_dir}
export LD_LIBRARY_PATH=\
${LD_LIBRARY_PATH}:${tools_dir}

./app 0 3
${tools_dir}/../bin/kp_reader *.dat
```

Finally, we get the following performance results in standard output file in the ascii format. We use the average time for a measurement.

```
 (Type)   Total Time, Call Count, \
Avg. Time per Call, ...
---------------------------------

Regions:
- ddc_splines_solve
 (REGION)   0.029775 10 0.002978 \
            22.599015 16.500760

---------------------------------
```

E. Experimental workflow for "Benchmark for spline construction with iterative and direct methods" (section V)

1) Compilation: For the compilation of benchmark, we set `-DDDC_SPLINES_SOLVER=GINKGO` for Ginkgo and `-DDDC_SPLINES_SOLVER=LAPACK` with `-DDDC_SPLINES_VERSION=2` for Kokkos-kernels.

2) Evaluation and expected results: We execute the benchmark `app` (`build/benchmarks/ddc_benchmark_splines`) in the following way.

```
./app --benchmark_format=json \
--benchmark_out=<file>.json
```

`<file>` is `splines_bench_<lib>_<backend>.json`, where `<lib>` is `ginkgo` or `lapack` and `<backend>` is `omp`, `cuda` or `hip`.

After we gather all the profiles, we visualize the data with python in the following way, where all the json files must be present under `<dir>`. Figs. 2 (a)-(f) are found under `<dir>` in png format.

```
python comparison.py -dirname <dir>
```