

RESEARCH ARTICLE | JUNE 07 2021

## Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU



Special Collection: [Building the Bridge to Exascale Computing: Applications and Opportunities for Plasma Science](#)

K. Germaschewski ; B. Allen ; T. Dannert; M. Hrywniak ; J. Donaghy ; G. Merlo ; S. Ethier ; E. D'Azevedo ; F. Jenko ; A. Bhattacharjee



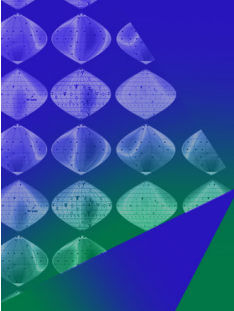
*Phys. Plasmas* 28, 062501 (2021)

<https://doi.org/10.1063/5.0046327>

CHORUS




CrossMark



## Physics of Plasmas

### Features in Plasma Physics Webinars

Register Today!



# Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU

Cite as: Phys. Plasmas **28**, 062501 (2021); doi: [10.1063/5.0046327](https://doi.org/10.1063/5.0046327)

Submitted: 3 February 2021 · Accepted: 7 May 2021 ·

Published Online: 7 June 2021



View Online



Export Citation



CrossMark

K. Geraschewski,<sup>1,a)</sup> B. Allen,<sup>2,3</sup> T. Dannert,<sup>4</sup> M. Hrywniak,<sup>5</sup> J. Donaghy,<sup>1</sup> G. Merlo,<sup>6</sup> S. Ethier,<sup>7</sup> E. D'Azevedo,<sup>8</sup> F. Jenko,<sup>6,9</sup> and A. Bhattacharjee<sup>7</sup>

## AFFILIATIONS

<sup>1</sup>Department of Physics and Space Science Center, University of New Hampshire, Durham, New Hampshire 03824, USA

<sup>2</sup>Department of Computer Science, The University of Chicago, Chicago, Illinois 60637, USA

<sup>3</sup>Computational Science Division, Argonne National Laboratory, Lemont, Illinois 60439, USA

<sup>4</sup>Max Planck Computing and Data Facility, 85748 Garching, Germany

<sup>5</sup>NVIDIA GmbH, 52146 Würselen, Germany

<sup>6</sup>Oden Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, Texas 78712, USA

<sup>7</sup>Theory Department, Princeton Plasma Physics Laboratory, Princeton, New Jersey 08540, USA

<sup>8</sup>Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37830, USA

<sup>9</sup>Max Planck Institute for Plasma Physics, 85748 Garching, Germany

**Note:** This paper is part of the Special Collection: Building the Bridge to Exascale Computing: Applications and Opportunities for Plasma Science.

<sup>a)</sup> Author to whom correspondence should be addressed: [kai.geraschewski@unh.edu](mailto:kai.geraschewski@unh.edu)

## ABSTRACT

GENE solves the five-dimensional gyrokinetic equations to simulate the development and evolution of plasma microturbulence in magnetic fusion devices. The plasma model used is close to first principles and computationally very expensive to solve in the relevant physical regimes. In order to use the emerging computational capabilities to gain new physics insights, several new numerical and computational developments are required. Here, we focus on the fact that it is crucial to efficiently utilize GPUs (graphics processing units) that provide the vast majority of the computational power on such systems. In this paper, we describe the various porting approaches considered and given the constraints of the GENE code and its development model, justify the decisions made, and describe the path taken in porting GENE to GPUs. We introduce a novel library called *GTENSOR* that was developed along the way to support the process. Performance results are presented for the ported code, which in a single node of the Summit supercomputer achieves a speed-up of almost  $15\times$  compared to running on central processing unit (CPU) only. Typical GPU kernels are memory-bound, achieving about 90% of peak. Our analysis shows that there is still room for improvement if we can refactor/fuse kernels to achieve higher arithmetic intensity. We also performed a weak parallel scalability study, which shows that the code runs well on a massively parallel system, but communication costs start becoming a significant bottleneck.

Published under an exclusive license by AIP Publishing. <https://doi.org/10.1063/5.0046327>

## I. INTRODUCTION

The US Department of Energy (DOE) Exascale Computing Project (ECP), which is arguably one of the most ambitious scientific computing activities ever undertaken, aims at delivering an exascale computing ecosystem that integrates hardware, software infrastructure, and applications. Teams of domain scientists, applied mathematicians, software engineers, and computer scientists work together on

the next generation of DOE-relevant codes that will make efficient use of the upcoming exascale computers. One of those codes is the high-fidelity whole device modeling application (WDMAPP),<sup>1</sup> which is being developed to tackle the grand challenge problem of magnetic confinement fusion from first principles. The physics involved spans a very large range of space and time scales, rendering the simulation of an entire fusion device with all its physics essentially infeasible, at least

within a single-algorithm code. This is why fusion scientists have historically split the challenge by focusing on specific regions of scales using models and numerical algorithms appropriate for the physics being simulated. The gyrokinetic model has been particularly successful at covering a wide range of scales while kinetically describing the complex effect of plasma turbulence and its impact on energy and particle transport across the confining magnetic field in tokamak fusion devices. Gyrokinetics is a 5D description of a magnetized plasma, reduced from the most fundamental fully kinetic 6D description of the evolution of the plasma distribution function, which will remain computationally prohibitively expensive even on exascale platforms for simulating fusion devices at realistic scale.

Gyrokinetic codes are computationally intensive but are highly scalable and have successfully used the full power of modern day petascale computers. This makes them ideal as the core application of an exascale whole device model to fit the ultimate goal of the WDMAPP project, namely, to simulate an entire tokamak discharge. Existing gyrokinetic codes are highly tuned for a specific tokamak region. Tokamak core codes like GENE<sup>2,3</sup> are optimized to capture small-amplitude turbulence in the closed flux-surface “core” region, whereas an edge gyrokinetic code like XGC<sup>4</sup> operates best with large-amplitude fluctuations in the near-separatrix and open-field-line regions. Extending the capability of existing codes to simulate the entire plasma volume is a major challenge; therefore, the approach pursued by WDMAPP is to couple together existing microturbulence codes. To achieve the final goal of WDMAPP, a scalable, flexible, highly efficient core-edge coupled whole device model requires several new developments, e.g., developing coupling technology, including additional physics, but also to port the constituent codes to the latest hardware architectures. Taken altogether, these developments are going to allow us to break into new multi-scale physics regimes that are not currently accessible. It is important to note that this paper focuses on just one particular aspect of this endeavor, that is, porting GENE to graphics processing units (GPUs) as used on the Summit supercomputer and on the future exascale systems Frontier and Aurora. GENE is one of the options to be used to simulate the core plasma in the WDMAPP coupled application,<sup>5</sup> with the other option being developed GEM.<sup>6</sup> The edge region will be handled by XGC.<sup>7</sup> All of these codes need to be able to run efficiently on GPU based systems, and complementary porting efforts are under way. It is expected that the majority of the computational resources will be used by XGC as it is using a much more expensive, particle-in-cell based method and has to be able to handle the more complicated magnetic topology beyond the last closed surface. In preliminary work performed so far, the core codes have only used 10% or less of the total computational resources. It is therefore at this point not necessary for GENE itself to scale to a full exascale machine. The results we present on GENE scalability in this paper show that we are close to the scalability required for a coupled WDMAPP, but still quite far from being able to scale GENE by itself to a full machine.

To run efficiently on exascale and existing accelerator-based pre-exascale systems like Summit at the Oak Ridge Leadership Computing Facility, codes must adopt new programming models. GENE is one of the leading simulation codes of micro-scale plasma turbulence and has a history of being adapted to run efficiently on many high-performance computing (HPC) systems. In this paper, we will describe our approach to porting GENE to run on Summit. We chose a hybrid

approach, keeping the existing Fortran code as the framework that controls program flow, and also preserving the original Fortran kernels, while adding the option to offload computationally intensive kernels to the GPU using a performance-portable C++14-based implementation. It is important to point out that, as explained in detail later, this was a choice that the GENE development team made with the goal of finding a solution that works for the existing team of developers and users as well as with the existing structure of the code. There is no one well established approach for porting codes to GPUs (or more generally to modern multi-core/many-core systems) that works well for everyone. Instead, there are a number of options with their own advantages and drawbacks. Early on, GPU ports were often done using low-level programming models for specific GPUs, e.g., CUDA called from Fortran in GENE<sup>8</sup> or CUDA Fortran in XGC. However, in recent years, the large coding efforts typically use either performance portability libraries or directive-based programming models. To list some more examples from plasma physics, PIconGPU<sup>9</sup> took an approach in some sense similar to what we describe here for GENE, developing their own performance portability library Alpaka<sup>10</sup> to suit their needs. Warp-X<sup>11</sup> also relies on a separate library, AMReX.<sup>12</sup> The gyrofluid code FELTOR<sup>13</sup> implements a layered approach, basing its numerical methods on matrix-free solvers that are implemented in terms of a hardware-independence layer of vector and matrix operations. This hardware independence layer can have multiple implementation, optimized, e.g., for GPUs (using CUDA) or Intel Xeon Phi (using C++/OpenMP), with a particular focus on bitwise reproducibility across architectures, which is not usually guaranteed in other codes/libraries. ORB5 (Ref. 14, see also references therein), GEM,<sup>6</sup> and the collision operator in XGC are examples of using a directive-based approach (OpenACC), while XGC has now moved to using the Cabana<sup>15</sup> particle simulation library, which is built on top of the general performance portability library, Kokkos.<sup>16</sup>

While Summit with its NVIDIA GPUs was our initial porting target, the approach as it has evolved is now designed to be GPU vendor independent in anticipation of the goal to run efficiently on the first exascale machines Frontier and Aurora, which will employ AMD and Intel GPUs rather than the NVIDIA GPUs used on Summit.

It is important to note that the challenges that come with the exascale area go far beyond GPU porting, though it is a crucial ingredient. Heldens *et al.*<sup>17</sup> provide a very nice literature review of the anticipated challenges on the path to exascale. We do not want to reiterate all of those challenges here, and many of them are beyond what can be addressed in terms of application code development, but rather need to be handled by vendors (e.g., system software and network interconnect). The work above finds two primary challenges: energy consumption and fault tolerance. Increasing the performance per Watt is crucial to attain exascale performance at a reasonable (if still high) power consumption on the order of 20 MW. Energy consumption is, of course, not something that can be primarily addressed in software, but it is the reason why many of the current leading supercomputers and the first exascale machines are going to rely on GPUs to achieve much higher energy efficiency than possible with a traditional central processing unit (CPU) architecture. Hence, our GPU port of GENE is a critical prerequisite for being able to run effectively on such systems.

Fault tolerance does not yet pose as big a challenge as originally anticipated for the exascale. A major reason for that is the “fat node” architecture of recent and upcoming supercomputers. If the scaling up

to exascale had happened by just scaling nodes, one would have expected to have to run on the order of millions of nodes by now. Instead, while the Titan supercomputer (2012) had almost 19 000 nodes, its successor Summit (2018) has less than 5000 “fat” nodes, that is, six powerful GPUs and two CPUs per node compared to Titan’s 1 GPU + 1 CPU design. This slow growth in node counts (or even decline, as in the example above) helps to performantly scale-up codes for extreme node-parallelism and also means that the chance of a node going down during a simulation run is still small enough that fault tolerance can be handled with the traditional checkpoint/restart approach.

Heldens *et al.*<sup>17</sup> identify various challenges with respect to algorithms, parallel scalability, and the software environment. We believe that the WDMAPP project overall does address some of these issues by its development of new strong and weak coupling technologies, though the GENE code itself has not undergone a major new algorithmic development in the work described in this paper. We agree with the reference that the software environments for the exascale remain unsettled at this point, with one aspect being the parallel programming models. We have designed a custom solution for GENE (which does generalize to certain class of problems, in particular, stencil-based computations, but that has not been our primary goal). We consider our work as a contribution to research in appropriate programming models and also think it is flexible enough that it can be adapted to lower-level approaches, be it, for example, SYCL, OpenMP, or Kokkos as clear best practices emerge.

## II. THE GENE CODE

GENE is one of the most widely used codes for simulating the plasma microturbulence. Recent results range from investigations of turbulence in the pedestal,<sup>18</sup> plasma shaping effects,<sup>19</sup> and fast ions<sup>20</sup> to isotope effect<sup>21</sup> and validation studies,<sup>22</sup> as well as space applications.<sup>23–25</sup> GENE employs the gyrokinetic model that is based on the assumption that the particle gyration about a mean magnetic field is much faster than all the dynamics of interest.<sup>26</sup> This allows one to average over the rapid gyromotion and eliminate the gyration angle as a phase space variable, thus reducing the dimensionality from six (three spatial and three velocity space coordinates of particles) to five coordinates (three spatial and two velocity space coordinates of gyrocenters).

### A. The gyrokinetic equations

The dynamics of the system are obtained by solving the time evolution of the gyrocenter distribution  $F_s(\mathbf{X}, v_{\parallel}, \mu, t)$  for each plasma species  $s$  according to the gyrokinetic Vlasov equation

$$\frac{\partial F_s}{\partial t} + \dot{\mathbf{X}} \cdot \nabla F_s + \dot{v}_{\parallel} \frac{\partial F_s}{\partial v_{\parallel}} = 0, \quad (1)$$

with  $\mathbf{X}$  being the gyrocenter position,  $\mu$  being the magnetic moment, and  $v_{\parallel}$  being the velocity component parallel to the background magnetic field  $\mathbf{B} = B\mathbf{b}$ .<sup>27</sup> The equations of motion of a gyrocenter with mass  $m$  and charge  $q$  read

$$\dot{\mathbf{X}} = v_{\parallel} \mathbf{b} + \frac{B}{B_{\parallel}^*} (\mathbf{v}_{\nabla B} + \mathbf{v}_{\kappa} + \mathbf{v}_{\mathbf{E}}), \quad (2)$$

$$\dot{v}_{\parallel} = -\frac{\dot{\mathbf{X}}}{mv_{\parallel}} \cdot (\mu \nabla B + q \nabla \bar{\phi}) - \frac{q}{m} \dot{A}_{\parallel}, \quad (3)$$

where  $\mathbf{v}_{\nabla B} = (\mu/(m\Omega B)) \mathbf{B} \times \nabla B$  is the grad-B drift velocity,  $\mathbf{v}_{\kappa} = (v_{\parallel}^2/\Omega) (\nabla \times \mathbf{b})_{\perp}$  is the curvature drift velocity, and  $\mathbf{v}_{\mathbf{E}} = (1/B^2) \mathbf{B} \times \nabla(\bar{\phi} - v_{\parallel} \bar{A}_{\parallel})$  is the generalized  $\mathbf{E} \times \mathbf{B}$  drift velocity. Here,  $\Omega = qB/m$  is the gyrofrequency, and  $B_{\parallel}^*$  is the parallel component of the effective magnetic field  $\mathbf{B}^* = \mathbf{B} + \frac{B}{\Omega} v_{\parallel} \nabla \times \mathbf{b} + \nabla \times (\mathbf{b} \bar{A}_{\parallel})$ . Finally,  $\bar{\phi}$  and  $\bar{A}_{\parallel}$  are the gyroaveraged electrostatic potential and the parallel component of the vector potential, respectively. In solving Eq. (1), GENE uses the delta-f method, splitting the distribution function  $F$  in a fixed background  $F_0$  and a fluctuating part  $f$ .

The system is closed self-consistently, computing the values of  $\phi$  and  $A_{\parallel}$  by solving the Poisson equation and the parallel component of Ampère’s law. The perturbed electrostatic potential is related to the perturbed charge density by means of the Poisson equation

$$\begin{aligned} \nabla_{\perp}^2 \phi(\mathbf{x}) &= -\frac{1}{\epsilon_0} \sum_s q_s n(\mathbf{x}) \\ &= -\frac{1}{\epsilon_0} \sum_s \frac{2\pi q_s}{m_s} \int B_{\parallel}^* f(\mathbf{x}) dv_{\parallel} d\mu, \end{aligned} \quad (4)$$

where  $n(\mathbf{x})$  denotes the density perturbation of the  $s$  species, expressed as the zeroth moment of the perturbed distribution function  $f$  in a particle space. Note that the velocity space moments need to be evaluated at fixed particle positions  $\mathbf{x}$ , whereas GENE evolves the gyrokinetic equation in gyrocenter variables  $\mathbf{X}$ . A pullback operator involving gyroaverages is therefore used to perform the necessary coordinate transformation. GENE uses a linearized pullback operator, see, e.g., Ref. 3 for more details, yielding

$$\begin{aligned} -\nabla_{\perp}^2 \phi(\mathbf{x}) &= \frac{2\pi}{\epsilon_0} \sum_s \left[ q_s \int f_s(\mathbf{X}) \delta(\mathbf{X} + \boldsymbol{\rho} - \mathbf{x}) \frac{B_{\parallel}^*}{m_s} d^3 \mathbf{X} dv_{\parallel} d\mu \right. \\ &\quad \left. - q_s^2 \int \tilde{\phi}(\mathbf{X}) \frac{F_0(\mathbf{X})}{T_0} \delta(\mathbf{X} + \boldsymbol{\rho} - \mathbf{x}) \frac{B_0}{m_s} d^3 \mathbf{X} dv_{\parallel} d\mu \right], \end{aligned} \quad (5)$$

where a tilde indicates the difference between a field and its gyroaverage. For the sake of readability, we have indicated only the spatial dependencies of distribution functions and fields.

Similarly, the  $A_{\parallel}$  potential is obtained by solving the parallel component of Ampère’s law for the fluctuation fields

$$-\nabla_{\perp}^2 A_{\parallel}(\mathbf{x}) = \mu_0 \sum_s j_{\parallel,s}(\mathbf{x}), \quad (6)$$

having neglected the displacement current. The perturbed parallel current  $j_{\parallel,s}$  is given by the first  $v_{\parallel}$  moment of the distribution function, obtaining

$$-\nabla_{\perp}^2 A_{\parallel}(\mathbf{x}) = \mu_0 \sum_s \frac{2\pi q_s}{m_s} \int B_{\parallel}^* v_{\parallel} f(\mathbf{x}) dv_{\parallel} d\mu, \quad (7)$$

which when expressed as a function of the gyrocenter distribution becomes

$$\begin{aligned} -\nabla_{\perp}^2 A_{\parallel}(\mathbf{x}) &= 2\pi\mu_0 \sum_s q_s \left[ \int v_{\parallel} f_s(\mathbf{X}) \delta(\mathbf{X} + \boldsymbol{\rho} - \mathbf{x}) \frac{B_{\parallel}^*}{m_s} d^3 \mathbf{X} dv_{\parallel} d\mu \right. \\ &\quad \left. - q_s \int \frac{\mathbf{b} \cdot \nabla \times \mathbf{b}}{B_0} \frac{mv_{\parallel}^2}{T_0} \tilde{\phi}(\mathbf{X}) F_{0,s}(\mathbf{X}) \right. \\ &\quad \left. \times \delta(\mathbf{X} + \boldsymbol{\rho} - \mathbf{x}) B_0 d^3 \mathbf{X} dv_{\parallel} d\mu \right]. \end{aligned} \quad (8)$$

GENE provides a comprehensive set of operational modes: flux-tube, global axisymmetric, and global non-axisymmetric, each suited for specific experimental conditions. Our current effort targets the *global axisymmetric* mode, as used by the WDMAPP project.

## B. Numerical approach

It proves convenient to introduce a modified distribution function  $g = f - \frac{q}{m} \bar{A}_{\parallel} \frac{\partial F_0}{\partial v_{\parallel}}$ , such that Eq. (1) can be formally rewritten as

$$\frac{\partial g}{\partial t} = \mathcal{L}(g) + \mathcal{N}(g), \quad (9)$$

where  $\mathcal{L}$  is a linear operator acting on  $g$  while  $\mathcal{N}$  represents the perpendicular nonlinearity. The right hand side of Eq. (9) will subsequently be referred to as just “the rhs” GENE employs a method-of-lines approach to solve Eq. (9):  $g$  is evolved on a fixed structured grid in a five-dimensional phase space that consists of three configurations (labeled  $x$ ,  $y$ , and  $z$ ) and two velocity space (labeled  $v_{\parallel}$  and  $\mu$ ) dimensions. Spatial coordinates are field-aligned. The underlying axisymmetry of a tokamak corresponds to an invariance of the unperturbed system with respect to the toroidal angle, which translates to an invariance with respect to  $y$ . Consequently, fluctuating fields related to a *linear* eigenmode correspond to a single  $k_y$  Fourier mode. This allows for a direct implementation of Eq. (9) in Fourier space for  $y$  and using pseudospectral methods. For a detailed description of the equations and numerical methods used, we refer to Ref. 3.

The main flow of program execution is shown in Fig. 1. Almost all of the computational work in a typical run is performed within the time loop. The standard time integration scheme used is an explicit fourth order Runge-Kutta (RK) method, which itself consists of four near identical stages. Each RK stage needs to calculate the rhs of the gyrokinetic Vlasov equation for all species. In order to do so, the electromagnetic potentials need to be known; hence, each rhs calculation is preceded by a computation of those potentials from the current 5D+species state.

The distribution functions  $g_s(x, y, z, v_{\parallel}, \mu)$  are stored in a 6D array (with species being the sixth dimension). The  $y$  dependence is represented in the Fourier space; all other directions are kept in a configuration space. The various terms that makeup the rhs involve derivatives in  $x$ ,  $y$ ,  $z$ ,  $v_{\parallel}$ , and also  $\mu$  if collisions are enabled.<sup>3</sup> Other than the  $y$  derivative, which is calculated by multiplication with  $ik_y$ , derivatives are approximated by finite differences, typically using a fourth order approximation requiring a five-point stencil. The rhs calculation hence largely consists of stencil computations. The nonlinearity is handled in a pseudo-spectral manner to avoid a computationally intensive convolution, i.e., the various terms are calculated based on the original Fourier-in- $y$  representation; those terms are then transformed to real space, the nonlinearity itself is computed, and the result is Fourier-transformed back in  $y$ .

The potentials, which are needed for the rhs evaluation, are computed from the current state  $g$  in a separate calculation of auxiliary quantities. Fundamentally, this requires the calculation of the 3D charge density and parallel current and then solving a Poisson equation to find the corresponding potential. In the gyrokinetic model, this is complicated by the fact that the distribution function is stored at gyro-centers, but the 3D fields need to be known at actual particle position. The computation of the moments therefore involves integrating out the  $v_{\parallel}$  dependence, then gyro-averaging to actual positions,

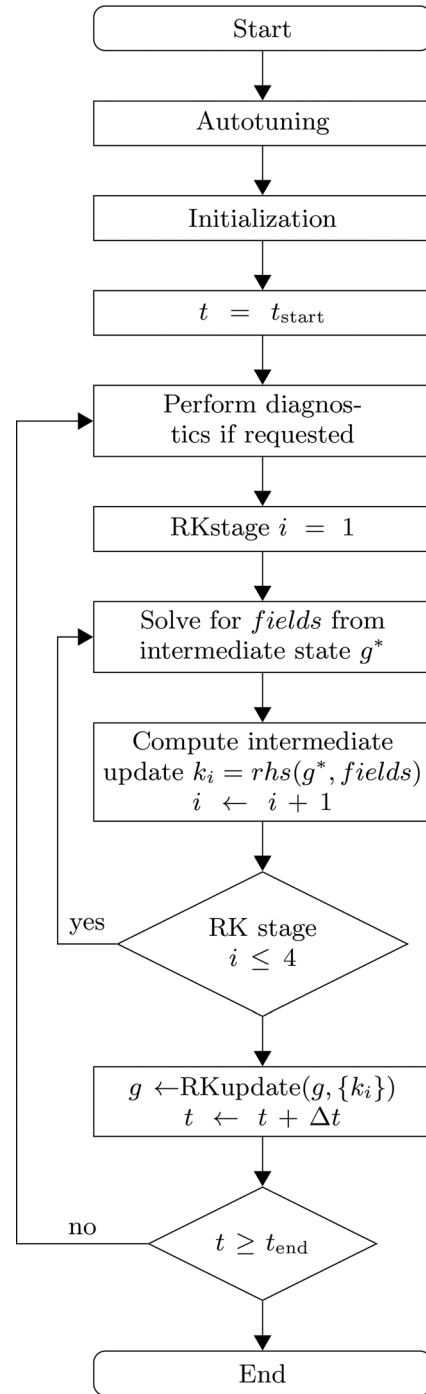


FIG. 1. Main program flow of GENE.

and integrating out the  $\mu$  and species dependence. The resulting potentials that are obtained from the field solves need to be known later at the gyro-centers, which requires another gyro-averaging procedure. The gyro-averaging procedure is written as a linear operator and computed by a banded matrix–vector multiplication.



The field solve itself separates in the  $y$  direction due to the Fourier representation, and in  $z$  due to the negligible dependence on the field-aligned direction. Hence, the field solve requires solving a 1D linear problem for every  $(k_y, z)$ .

### C. Parallelization and domain decomposition

The computations that go into the rhs are all local in the 5D+*species* distribution function space [with the exception of the Fast Fourier Transforms (FFTs) in the nonlinearity]. Hence, the distribution function can be straight-forwardly domain decomposed in all directions, and this is used to parallelize the computation using message passing interface (MPI). Decomposing in the  $y$  direction is usually avoided in nonlinear calculation since otherwise data have to be transposed to allow for the 1D FFTs in  $y$  to be calculated based on locally available data. The only obvious bottleneck to parallel scalability for the rhs calculation is the need to fill ghost points in all decomposed directions, other than species and  $\mu$  if collisions are not used.

The parallelization of the calculation of the auxiliary quantities is inherently more complex, as it starts out with the 6D distribution functions that are step-by-step reduced and gyro-averaged to 5D, 4D, 3D, the actual field solve is performed, and the results are gyro-averaged back to 5D. In the existing CPU code, those operations had typically accounted only for a smaller fraction of the run time due to their reduced dimensionality, and hence they had not been a focus of optimization. The reductions were performed using MPI\_Allreduce, and the calculation would be continued redundantly on multiple processes. When the domain is decomposed in the  $x$  direction, the linear operations for gyro-averaging and actual field solve are parallelized across processors, aided by the fact that many such operations would have to be completed in a batched manner (e.g., a 1D field solve along the  $x$ -direction would occur for every  $(k_y, z)$  point), which allows for overlapping of communication and computation.

### D. Implementation

The current implementation of GENE, preceding the GPU porting effort, makes heavy use of object-oriented programming (OOP) features of modern Fortran 2003/2008. Computational operations are encapsulated in classes, with an abstract base class defining the interface to a given operation, for example, the calculation of the nonlinearity or a single term in the linear part of the rhs computation. Derived classes then provide a specific implementation. Depending on the input parameters, an instance of a specific implementation class is created at run-time using the factory method pattern.<sup>28</sup> This scheme supports multiple goals: (a) It allows the selection of the right computational kernel given the overall characteristics of the simulation [e.g., depending on global vs local (flux-tube) mode, some derivatives are discretized differently]. (b) Different implementations can represent different physics or numerics, e.g., the Arakawa-implementation of the nonlinearity vs a direct discretization. (c) It allows for multiple implementations of the same calculation to exploit certain performance characteristics (e.g., computation of derivatives for a 2D  $x-y$  slice at a time vs a larger block in 6D space, which has implications for cache usage and the size of necessary temporary fields). In addition, these classes provide a natural place to store persistent data needed

(e.g., pre-calculated prefactors) as well as temporary fields for the given calculation.

### E. Auto-tuning

Due to the complexity of modern supercomputers, it is generally not easy to predict the run-time behavior of a complex code, as many factors are not well known. This includes the memory system that may include multiple levels of caches and non-uniform memory access (NUMA) characteristics, the characterization of the interconnect including different performance for on-node vs across-nodes communication, the amount of optimization the compiler achieves, etc.

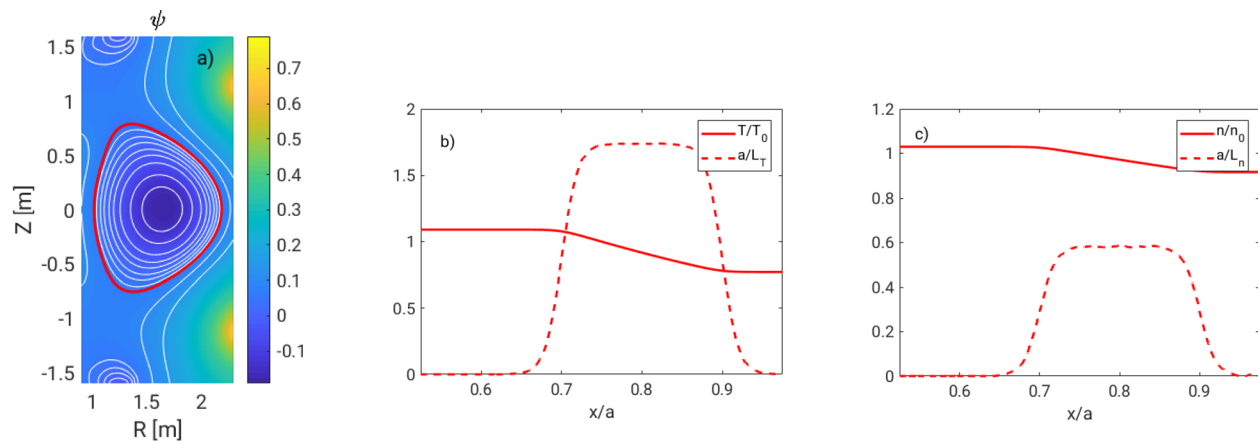
GENE therefore implements an auto-tuning process (see Fig. 1). In the most general setting, it iterates over all possible parallelizations—therefore testing for different domain decompositions—and all available implementations of algorithms (e.g., proceeding one 2D slice at a time for possibly better cache usage vs working on larger blocks at a time), estimates the memory need and if sufficient initializes them, runs a small number of timesteps, and finds the fastest of all of these combinations. For larger processor counts, this can take a long time, for which the search space can be reduced by either fixing some parallelization dimensions or at least reducing the possible range of numbers.

### F. Example production run

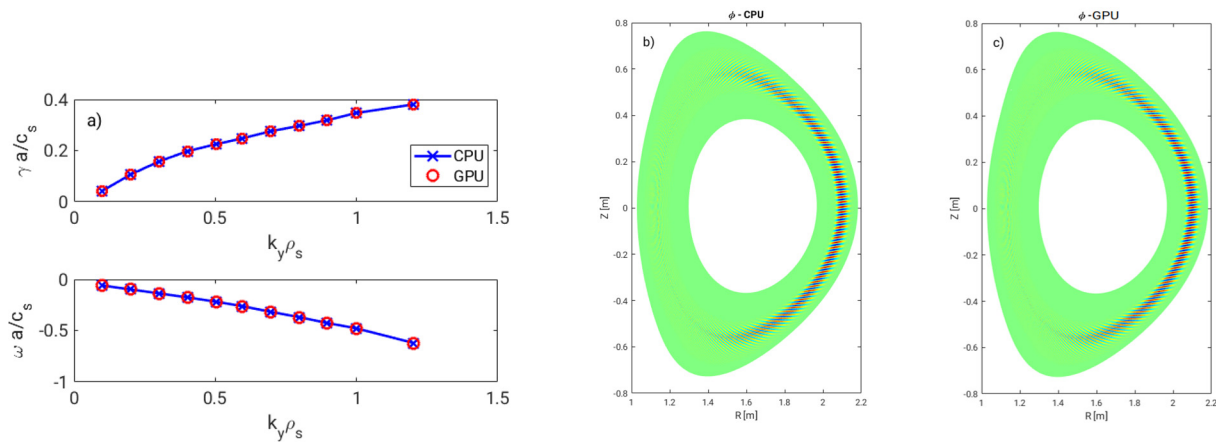
In order to exemplify the correctness and capability of the newly ported code, as detailed throughout the rest of this paper, we present a comparison with the results obtained using the regular CPU version. GENE is a thoroughly verified and validated code, see, e.g., Refs. 29–32. Because of their computational cost, most of the available benchmarks are however limited to the linear regime, or whenever nonlinear effects are also considered, an adiabatic electron response is often assumed.<sup>33</sup> For our purposes, it is necessary to include kinetic electrons, since they significantly increase the computational expense of the simulations, motivating the GPU port. Therefore, we have defined a new test-case, modeling a DIII-D plasma. Simulations consider the real MHD equilibrium and include ion and electrons with deuterium mass ratio and electromagnetic fluctuations, although the resulting turbulence is essentially electrostatic. Plasma profiles, shown in Fig. 2, have nonetheless been simplified with respect to an actual experiment by localizing the gradients driving turbulence only in the outer fraction of the radial domain. This allows us to significantly reduce the computational effort, while still considering a realistic production case.

We have performed both a linear and nonlinear comparisons, with the main results summarized in Figs. 3 and 4. In all cases, we have used a grid with  $n_x \times n_z \times n_{\parallel} \times n_{\mu} = 512 \times 56 \times 64 \times 30$  points per species, covering the domain  $0.52 \leq x/a \leq 0.97$ . Here,  $a = 0.68\text{m}$  is the DIII-D minor radius which, with our choice of temperature profile, corresponds to a value of  $\rho^* = \rho/a = 0.21 \cdot 10^{-2}$ . The nonlinear run retains 64  $k_y$  modes (with  $k_{y,\min} = 0.29$ , corresponding to simulating one fourth of the torus).

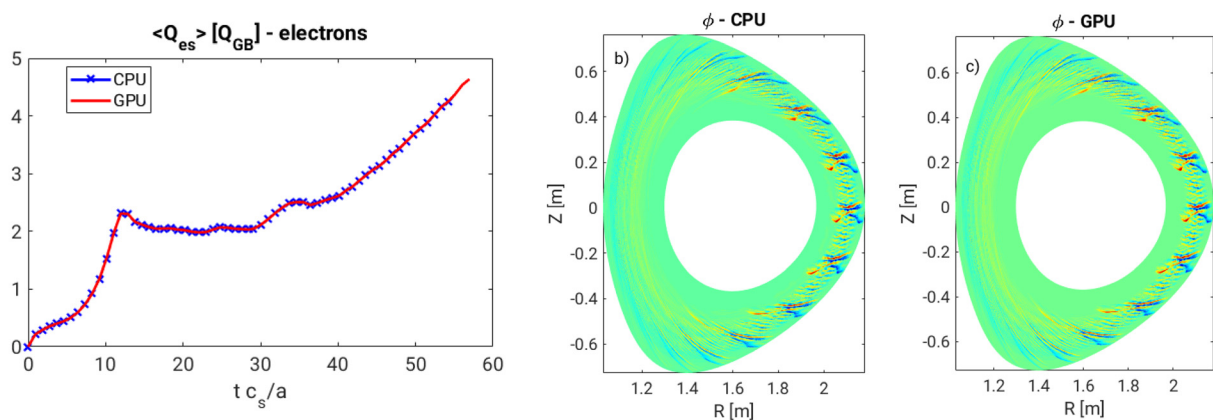
As one can see, the GPU code recovers essentially the same results as the CPU code. The linear analysis shows the same growth rate and mode real frequency, as well as the same eigenfunctions. Negative frequency corresponds here to trapped electron modes (TEMs). The nonlinear simulations have been carried out starting



**FIG. 2.** (a) Magnetic geometry (contours of the poloidal flux function  $\psi$ , in red the last closed flux surface) and (b) and (c) profiles used for the benchmark. The same temperature for ion and electrons is assumed.



**FIG. 3.** Linear comparison between CPU and GPU code results. Subplot (a) shows the converged frequency and growth rates (blue for CPU only and red using GPU), whereas (b) and (c) show a comparison of the eigenfunction obtained for  $k_y \rho_s = 0.3$ .



**FIG. 4.** Nonlinear comparison between CPU and GPU code results. Subplot (a) shows the time traces of the electron heat flux (blue for CPU only and red using GPU), whereas (b) and (c) show a comparison of a snapshot of the electrostatic potential.

with the exact same initial perturbation, which induces the same time evolution of the system, as can be seen by comparing the time traces of the volume averaged heat flux shown in Fig. 4. In terms of computational speedup, considering Summit nodes, the GPU code version turns out to be roughly  $8.2\times$  faster than the CPU. This is evaluated comparing the time per time step when using 80 nodes and averaged over 100 time steps. The GPU run uses therefore  $6 \times 80 = 480$  GPUs (and the same number of CPUs), whereas the CPU-only run uses  $42 \times 80 = 3360$  cores. The domain decomposition is obviously different between the two. For the CPU run, we have allowed the code to perform its autotuning procedure, identifying the best possible choice; similarly for the GPU, we have selected a domain decomposition that provides the fastest solution. A detailed discussion of scalability is deferred in Sec. IV C.

It is important to point out that the results when running on the GPU are not exactly the same. GENE, in general, has made a choice not to ensure bitwise reproducibility of its results when run on different machines, with different domain decompositions, etc., and consequently, we also have not attempted to achieve this for switching between CPU and GPU runs. We believe this is a common choice, though other codes like FELTOR<sup>13</sup> go to extra lengths (e.g., Ref. 34) to maintain bitwise reproducibility. In particular for a turbulence code, small changes in the initial perturbation or small machine-precision errors can grow quickly and can potentially cause a turbulent distribution function snapshot at a given late point in time to look very different (errors of order 1 in the  $l_\infty$  norm). However, the same can (and will) happen, for example, if running at different resolutions (truncation errors usually eclipse machine precision errors) or varying other parameters, which are often chosen to match experimental conditions and are hence not exactly known. Therefore, GENE users make sure that their physically relevant results (e.g., transport quantities) are converged but do not aim for bitwise reproducibility.

### III. GPU PORTING APPROACH

#### A. Challenges and constraints

GENE is a widely used code in the fusion community with quite diverse applications. Some users run relatively small and cheap flux-tube computations for their specific physics questions, while others couple GENE with external codes to perform the whole device modeling of fusion devices. It is therefore important that the code can run on a large range of computer systems, from a user's laptop to an upcoming exascale supercomputer. Using available resources is obviously very important, in particular, in the case of large, resource-heavy computations. GENE has a history of being ported to new systems.<sup>35,36</sup>

On the other hand, it is also necessary to keep the code maintainable and easily extendable. This goes down to the level of a graduate student working on a specific physics problem being able to modify the code to implement a new term in the equations or a new diagnostic. In these cases, performance is not a primary consideration but readability and ease of understanding are.

In order to support the wide range of use cases, it is important to rely on an open, standardized infrastructure. One example is the message passing interface (MPI), which is very mature, stable, and available virtually everywhere. GENE employs MPI for coarse-level parallelism. Our preference is not to rely on vendor-specific solutions

to ensure that the code can be used on a wide range of machines and compilers.

It is also crucial that the GENE code remains maintainable with reasonable resources. In particular, there are no guarantees that there is continued support for a large team of computational experts, as usually funding is obtained based on physics applications. That means that code duplication should be avoided in as far as possible, and the code should remain accessible to domain (physics) scientists, not just performance engineers or computer scientists.

#### B. Choice of programming model

Given the challenges and constraints just presented, it was decided to add the GPU capability within the existing codebase. The alternative of forking the code into two separate versions, a GPU and a traditional CPU one, was rejected. While a “Mini-GENE” had been created to investigate some aspects of the GPU port, having to maintain two separate versions of GENE that would inevitably diverge not only in their (performance) implementation but also in terms of the physics supported or the numerics implemented would not have been sustainable.

It may seem that the easy answer would be to create just a single “performance-portable” version of GENE. While this is the primary goal that was adopted, in practice, compromises were made. It is worth noting that our exact approach has evolved over time, and we will outline it here including the lessons learned.

A primary goal in the porting process was to maintain the large existing Fortran code base. The main reason for that is the relative simplicity and stability of Fortran, which means that physicists can more easily understand the code and make changes than if they have to learn, e.g., the depths of modern C++ programming. (We acknowledge that the use of modern OOP Fortran also raises the bar in being able to work with the code.)

We would like to note that the porting path taken in this work was designed to fit the needs and circumstances of the GENE code, and we therefore do not claim that our approach is directly applicable to other codes in different circumstances—and in fact, our approach was adjusted over time as the port progressed. We hope, though, that it is useful to describe our thought process and experiences. One particular constraint in our case is that GENE is part of the ECP WDMAPP project, which means that one goal is to efficiently run on upcoming exascale systems that, while all GPU-based, employed significantly distinct hardware and programming environments. The initial three Exascale supercomputers that are planned to become available in the United States are all heterogeneous systems where multi-core CPUs are augmented with GPU-based accelerators: Aurora at Argonne National Lab (using Intel GPUs), Frontier at Oak Ridge National Lab, and El Capitan at LLNL (using AMD GPUs). Nearly all of the computing power of these systems is expected to be provided by the GPUs. The main challenge is to bring most of the computation and data to the GPUs to make use of these powerful resources.

The vast majority of the runtime of production runs is spent in the timeloop, specifically the four near identical RK stages that have therefore been the main focus for the GPU port.

Our initial target for the port was chosen to be the Summit supercomputer, which uses NVIDIA GPU hardware. We made sure, though, that our decisions did not lock us into a software environment



that would prevent us from successfully and efficiently using other hardware architectures.

The spectrum of possible porting approaches for the initial port can be explored by considering the different options to generate GPU kernels:

*Use of existing GPU libraries.* In this case, optimized GPU code already exists and just needs to be employed. This approach in fact has been partially used in the GENE GPU port. In particular, GENE can use NVIDIA-provided libraries like cuFFT for Fast Fourier Transforms (FFTs) and cuBLAS for linear algebra routines, as well as, optionally, cuSPARSE for linear solves (similar libraries exist or are under development, for other GPU architectures). However, while this covers an important subset of GENE's computational kernels, it is far from enough, as the vast majority of GENE consists of custom code, mostly stencil computations, which are not available from existing libraries.

*Directive-based approach.* To generate custom GPU kernels for GENE, we considered a directive-based approach first, like OpenACC<sup>37</sup> or OpenMP  $\geq 4.5$ .<sup>38</sup> This compiler-driven option generally has the lowest barriers to entry, as the existing Fortran code can remain unchanged—at least initially—and simply be augmented by adding directives. Specifically, a given piece of Fortran code can be compiled into CPU code (possibly parallelized using CPU threads) or GPU code as needed. No duplication of the code or rewriting in another language is needed.

In practice, however, problems were encountered with compiler support. In particular, stable OpenMP-offload compilers were not available at the start of the project, and the main option for OpenACC was the PGI compiler, which had issues compiling GENE correctly due to its use of unsupported Fortran 2003/2008 features. Therefore, while we explored this venue first and obtained promising results on selected kernels, in the end we shelved this approach, with the option of revisiting it again in the future.

*CUDA Fortran.* The only other option to generate GPU kernels directly from Fortran is CUDA Fortran as provided by the PGI and IBM's XL compilers. While it can require some code duplication to maintain CPU and GPU versions of kernels, it is a productive porting approach that keeps the entire code base in a single language and avoids the complications of interfacing Fortran with C/C++/CUDA. However, since other vendors do not provide native Fortran GPU support in their toolchains or conversion tools such as “hipify” for CUDA C/C++, this option does not provide a clear transition path to non-NVIDIA GPUs and was therefore not appropriate for GENE.

*CUDA C++.* The vast majority of NVIDIA GPU kernels are compiled from the CUDA language, which is the original programming model for general purpose programming of NVIDIA GPUs. CUDA is essentially C++ (and therefore a superset of C) with extensions to designate the device code and invoke it from the host. CUDA is the most mature GPU programming approach, and usually the language of choice when GPU kernels or libraries are developed from scratch.

In the case of GENE, CUDA was initially not the developers' preferred approach. Due to the reasons laid out earlier, the main code base was to remain in Fortran, and hence generating a kernel from C/C++/CUDA code would inevitably mean duplication of an existing algorithm in a new language. Bare CUDA grants more fine-grained access to the hardware, which makes it possible to extract near-

optimal performance, but it requires good understanding of how GPUs work, and writing CUDA code manually is not highly productive. This can be improved by making use of advanced C++ features that in fact also make it possible to write GPU architecture independent code, for example, by using frameworks like Kokkos,<sup>16</sup> RAJA,<sup>39</sup> ALPAKA,<sup>10</sup> or AMReX.<sup>12</sup> Additionally, as part of C++17, the parallel STL and its execution policies provide a language-integrated, standardized option for parallelizing algorithms, which also allows GPU off-loading through a supporting compiler.

In the end, CUDA C++ was chosen to make progress quickly on the immediately available NVIDIA hardware and software, despite its portability drawbacks. This decision was made with the intention of abstracting the CUDA implementation to support other GPU vendors in the future—in essence CUDA was used to prototype the general approach needed to have GENE run efficiently on GPUs.

### C. Restructuring of the code for efficient parallelization

Following the decision to use CUDA to implement kernels, the first step—other than managing memory, which will be detailed later—was to define a set of low-level operations that could optionally be off-loaded to the GPU. Following GENE's general design, the interface was defined by an abstract base class, with type-bound procedures (TBPs) (virtual function calls in usual OOP parlance) for, e.g., a 1D complex-to-real FFT, a BLAS axpy type operation, or a certain derivative computation by means of a stencil computation.

Two derived classes of the interface class were provided—a CPU implementation, where the current GENE implementation was moved to, and a GPU implementation which called CUDA/C++ code via Fortran/C interoperability. When feasible, the CUDA implementation would just use an existing library like cuFFT, cuBLAS, and otherwise custom CUDA kernels were implemented by hand as a first step.

This approach was workable and allowed us to make initial progress toward a version of GENE that could be used on the CPU as before, but would take advantage of GPUs when available. There were, however some drawbacks, which led to a modification of the approach as the port progressed:

1. Performance gains were often rather limited, because the low-level operations mostly worked on small problems at a time, which are well suited for a CPU architecture where they use the caches efficiently, but not ideal for a modern, highly parallel GPU.
2. Writing custom CUDA kernels was labor-intensive and also error-prone, despite unit tests being added. Two factors contributed: (a) C/C++/CUDA natively do not have a flexible way of supporting multi-dimensional array accesses. Initially, we just passed the pointers and the multi-d to 1D index mapping was performed manually. (b) Using CUDA directly is somewhat verbose, as kernels are written separately from their invocation, and both have to deal with the mapping of the computational work to threads and threadblocks. A custom CUDA implementation of a kernel would often be 10× more lines of code than the usually quite short original Fortran implementation.
3. The resulting CUDA code was obviously vendor specific for NVIDIA. Given existing CUDA code, porting to HIP (AMD GPUs) is mostly automated, but Intel's programming model

(SYCL/OneAPI) is quite distinct—though automated conversion tools are also being developed.

Efficiently parallelizing across Fortran's TBPs, which are used extensively in GENE, can be challenging. The existing design was motivated by achieving effective cache usage on CPUs, where the kernels inside a TBP work on a relatively small sub-problem at a time. However, GPUs need to be able to work on large problems so that thousands of threads can be active at a time in order to work efficiently and many of the TBP interfaces were designed in a way that limited the exposed parallelism. Therefore, significant refactoring of the existing Fortran code was required to address this first issue in order to use GPUs to their full potential. The solution was to move away from the newly introduced low-level operations approach and instead provide the transition to GPU control at the existing level of Fortran classes previously described. This level of abstraction already exists and allows for multiple implementations of a given numerical term, like the non-linearity, so another implementation that runs on the GPU could easily be added. While the CPU could still proceed in a cache-friendly fashion on smaller pieces of the problem at a time, the GPU could now work on the entire problem, typically covering the whole local 6D subdomain, at once. We also made use of this layer of abstraction for testing—in order to make sure that the GPU implementation is correct, we made sure that the CPU and GPU implementations of a given algorithm give the same result to machine precision. We verify this both by using the existing GENE testsuite, which works at a higher level, and in most cases we also added unit tests to specifically test implementations of a single algorithm at a time.

The remaining issues were addressed by our development of the `GTENSOR`<sup>40</sup> library, which will be described in more detail below.

## D. Memory management

Given that GENE is an Eulerian code, with the distribution function discretized on a mesh, multi-dimensional arrays are used pervasively throughout the code. Fortran has an excellent native language support for multi-dimensional arrays, especially since Fortran 90 where arrays can be passed to subroutines via interfaces, which will not just pass the data, but also information about the dimensionality and data layout, including bounds.

With Fortran supporting multi-dimensional arrays as part of the language, though, its support is not easily extendable. In our work, we encountered two limitations: GENE frequently wants to subdivide and reshape data arrays. While selecting a subset is natively supported well, the built-in `reshape` function sometimes does make a copy of the data, which is not what is needed. For example, GENE often recasts a 6D distribution function to a 3D ( $x, y$ , block number) array that can be processed in a batched fashion. That is, dimensions 3–6 are collapsed into a single index, while the underlying data storage remains the same.

The second limitation is that Fortran has no support for custom memory allocators. This is particularly important for the design of our GPU port, where we need to allocate GPU accessible memory from the main Fortran code base. It is not feasible to allocate and copy data to/from device memory at each Fortran–CUDA boundary for computations that are performed on the GPU. The performance penalty of these frequent host-device memory transfers would be too high, since most of the actual computations are short.

We introduced a new data type `reg_storage_t`, which is a class encapsulating a defined region of contiguous memory, and its interpretation as multi-dimensional arrays of varying dimensionality. It replaces standard allocatable arrays throughout the code, and allocations and deallocations are handled through this type rather than directly calling `allocate` and `deallocate`. Views of the data in a given dimensionality are obtained as pointers, using, e.g., `regsto_f%get_6d_ptr(bounds...)`.

The issue of allocating GPU-accessible memory was resolved by providing two derived classes of `reg_storage_t`, one that allocates regular CPU memory, while the other allocates, typically, CUDA managed memory (though it can also allocate GPU device memory directly). Managed memory is a feature of CUDA that automatically migrates a memory region between the host (CPU) and device (GPU) memory depending on where it is accessed from. It is very helpful in porting, since during development, some kernels are still on the CPU, while others have already been ported to the GPU, so the developer does not have to explicitly transfer memory back and forth. (The transfer costs are still incurred, though they just happen automatically.) Managed memory has worked well for this porting effort. Current generation AMD GPUs, however, slow down significantly when using AMD's version of managed memory, which is supposed to be improved for the upcoming Frontier system. It is not clear yet, whether Intel's GPU programming model supports a similar feature. In any case, since now the entire time step, other than diagnostics, has been successfully ported, we now have the option to switch the data to GPU device memory in the future, with copies only required when performing diagnostics.

## E. The `GTENSOR` productivity and portability library

The development of the `GTENSOR` library happened somewhat organically—it evolved as we were making progress on the port and learned lessons of what was needed to keep GENE maintainable and the porting process productive. `GTENSOR` started as an effort to simplify passing of multi-dimensional arrays from Fortran to C/C++ and CUDA. As opposed to Fortran, C/C++ do not have sophisticated multi-d array handling built into the language. However, C++'s OOP and generic programming features allow for implementing multi-dimensional arrays in a library, which provides a large amount of flexibility, but unfortunately lack standardization. Many such multi-d array classes exist as libraries, e.g., `BOOST`,<sup>41</sup> `EIGEN`,<sup>42</sup> `XTENSOR`,<sup>43</sup> etc.

It was decided to closely follow `XTENSOR`'s<sup>43</sup> approach, a header-only library that in turn provides array handling in C++ largely following Python's popular NumPy library.<sup>44</sup> Much of the semantics of `XTENSOR` and `GTENSOR` will be familiar to computational Python developers. `XTENSOR` is a complex piece of software with many features, some of which are not needed for GENE and is not originally designed to support GPUs. For this reason, writing a simpler new implementation was seen as a faster approach rather than adapting the complex CPU optimized library to run on GPUs. `XTENSOR` and `GTENSOR` provide owning and non-owning arrays (i.e., views that do not own the memory but provide an array-like view into data actually owned by a separate entity, including allocated from Fortran). The arrays have standard C++ value semantics and lifetime rules. The underlying data storage can be flexibly chosen, but remains transparent to an application using `GTENSOR`. We are using `THRUST`'s<sup>45</sup> `host_vector`

**ALGORITHM 1.** Calling a CUDA/C++ kernel from Fortran.

---

```
call add_i_sten_2d_wrapper(dist, shape(dist),
                          rhs, shape(rhs),
                          sten, shape(sten))
```

---

**ALGORITHM 2.** Interfacing arrays passed from Fortran.

---

```
extern "C" void add_i_sten_2d_wrapper (
    thrust::complex<double>* _dist,
    const int dist_shape,
    thrust::complex<double>* _rhs,
    const int *rhs,
    double *_sten,
    const int *sten_shape
)
{
    auto dist = gt::adapt_device<2>(_dist,
    dist_shape);
    auto rhs = gt::adapt_device<2>(_rhs,
    rhs_shape);
    auto sten = gt::adapt<1>(_sten,
    sten_shape);
    //...
}
```

---

and `device_vector` to provide arrays that are accessible from the CPU and the GPU, respectively.

As an example of how this simplifies Fortran interfacing, [Algorithm 1](#) shows the calling Fortran code and [Algorithm 2](#) gives the C++/CUDA implementation of passing multi-d arrays:

In this example, `dist` and `rhs` are allocated in managed memory, which will be directly accessed in GPU kernel code. `sten` is an example of an array in CPU memory. The arrays can be directly accessed on the C++/CUDA side using familiar notation, e.g., `rhs(i,j) = fac * (dist(i+1,j)-dist(i-1,j))`, which directly accesses the memory allocated from Fortran, both in host code and in device code.

**1. CUDA kernels via lambda functions**

These array objects significantly simplified implementation of custom CUDA kernels. Kernels, which were previously implemented from scratch, mapping multi-dimensional indices to a 1D index into simple pointers passed from Fortran, could now directly access the data using multi-dimensional indexing. Still, those hand-written CUDA kernels had a lot of repeated “boilerplate code,” from setting up the CUDA thread grid to invoking the kernels separately from their implementation.

The next step was adding generation of CUDA kernels using C++ lambda functions. This is a modern C++ feature that allows us

**ALGORITHM 3.** Example of a lambda function generated kernel.

---

```
Gt::launch<2> (
    rhs.shape(), GT_LAMBDA(int i, int j) mutable {
        rhs(i, j) = (sten(0) * dist(i - 2, j) +
                    sten(1) * dist(i - 1, j) +
                    sten(2) * dist(i, j) +
                    sten(3) * dist(i + 1, j) +
                    sten(4) * dist(i + 2, j));
    })
```

---

to define and invoke functions right where they are needed and has recently become popular as an approach to programming GPU.

A full implementation of a function that calculates a fourth order finite difference approximation to the  $x$  derivative could now be written as seen in [Algorithm 3](#).

This is in fact quite close to how this kernel is expressed in Fortran directly. `gt::launch` takes a range of indices (2D, in this case) to iterate over—it then calls the provided function for every  $(i, j)$  in that range. More specifically, it actually generates a CUDA kernel where each thread is performing the updated for  $(i, j)$ , so that all threads work together to achieve the calculation in parallel.

Note that this approach, while relatively recent, is not novel and is the basis of how productivity and performance portability libraries like Kokkos,<sup>16</sup> RAJA,<sup>39</sup> AMReX,<sup>12</sup> etc., as well as SYCL/OneAPI work. All of these libraries go way beyond this relatively simple feature and extend to reductions, multi-level parallelism, etc. In the beginning of the GPU porting effort Kokkos and RAJA had been considered; however at the time we were still hoping to keep all the code in Fortran, and those libraries depend on C++ language features like templates and lambda functions that are not available in Fortran. The intention of Kokkos and RAJA is to provide a framework for performance portability, where kernels have to be written only once. Following that approach would have meant transitioning large parts of GENE to C++, which, as stated before, conflicted with the goal to keep a working Fortran codebase. Nevertheless, the approach we ended up pursuing, which trades some code duplication for the preservation of the Fortran codebase, could also have employed one of the existing frameworks, just using a small subset of their features, limiting their use to generate GPU kernels. However, `GTENSOR` already existed at this point and it was quite straightforward to add the needed lambda-function kernel features to `GTENSOR`, so pulling in a large framework, which comes with its own maintenance costs, appeared to not be worth it at the time. Indeed, the subsequent extension of `GTENSOR` to support lazily evaluated expressions via template metaprogramming is something that Kokkos or RAJA do not provide—the idea, based on pioneering work on expression templates in BLITZ++,<sup>46</sup> is taken from `XTENSOR`, which however does not support GPU kernel generation. Nevertheless, it would certainly be possible to use Kokkos or RAJA on the backend side of `GTENSOR`, which would give us access to more sophisticated reductions, for example, rather than having to re-implement such a feature within `GTENSOR`. We will reconsider using one of those libraries under the hood in the future, as that would also mean that we do not have to support backends for various hardware



(e.g., NVIDIA vs AMD vs Intel) ourselves, though so far that has required surprisingly little effort.

## 2. Expression templates for GPU kernel generation

We have now described how `GTENSOR` simplified the interfacing between Fortran and C++/CUDA, and its lambda function-based approach also removes any CUDA-specific pieces in the application code. Although it is also possible to compile the `GTENSOR` kernels into CPU code, this feature is used in GENE only for debugging purposes, since we already have explicit Fortran implementations of such kernels for the CPU.

It turned out that the vast majority of computations in GENE could be expressed even more simply in C++, reusing the idea of `XTENSOR`'s lazily evaluated expression templates. While the implementation involves non-trivial template metaprogramming in C++, which is beyond the scope of this paper to detail, this allows recasting the example derivative calculation above as shown in [Algorithm 4](#).

This expression generates the same GPU kernel as shown previously. We acknowledge that the view function may make it somewhat opaque what is going on, though this is primarily an issue that in C++ we cannot use the same compact colon-based slice notation that Fortran and Python/NumPy have; we therefore added the NumPy equivalent as comments, which is really only a difference in notation.

A big, if invisible, difference is that in Python and Fortran such an expression would be evaluated operation by operation. That is, the first line would be evaluated and then stored into a temporary. Next, the second line would be evaluated and then added to the temporary, etc., until the computation is complete, when finally the result is moved into rhs. We would expect a good optimizing Fortran compiler to fuse the loops and avoid the temporaries, though. It should be noted that the original Fortran kernel code does not use array expressions in such cases in the first place, it uses an explicit loop that fuses the computations for each grid point. However, a conventional implementation of multi-dimensional arrays in C++ would lead to the aforementioned behavior, which is not a particular efficient implementation on the CPU, and it would definitely lead to very poor performance on the GPU, since it would involve nine kernel invocations, one for each multiplication and addition, so each kernel would only do minimal work (one flop per array element, but having to repeatedly transfer the arrays from and to memory). Such low arithmetic intensity (AI) severely limits GPU performance. The conventional approach is therefore not a feasible option, and instead `GTENSOR` uses lazily evaluated expressions to ensure that only a single kernel is generated to evaluate expressions like the one shown in [Algorithm 4](#).

**ALGORITHM 4.** Example of a stencil computation using lazily evaluated slicing expressions.

---

```

rhs =                                     //Python equivalent
  sten(0) * f.view(_s(_, -4)) + //sten[0] * f[:-4] +
  sten(1) * f.view(_s(1, -3)) + //sten[1] * f[1:-3] +
  sten(2) * f.view(_s(2, -2)) + //sten[2] * f[2:-2] +
  sten(3) * f.view(_s(3, -1)) + //sten[3] * f[3:-1] +
  sten(4) * f.view(_s(4, _)) ; //sten[4] * f[4:]

```

---

The `GTENSOR` approach (like `XTENSOR` and other libraries—the idea of expression templates is not new) returns un-evaluated expressions from the `+` and `*` operators. Only once the resulting expression is assigned to `rhs`, a single kernel is generated that do all of the arithmetic operations in one go. This feature of `GTENSOR` also makes it straightforward to fuse computational kernels in order to achieve higher arithmetic intensity (AI), as will be shown in a more detailed example later. `GTENSOR` in its current state has been custom designed for the needs of GENE's GPU port. It should, however, be applicable to other codes, particularly those that primarily perform stencil computations. In fact, one of the authors has started to use it for the field calculations in a particle-in-cell code. `GTENSOR` does not, however, try to achieve the generality of frameworks like Kokkos and RAJA. On the other hand, its relative simplicity facilitated porting to different accelerators. `GTENSOR` already supports both AMD HIP/ROCm for running on AMD GPUs and SYCL for running on Intel GPUs, which will be used for expanding our GENE port to run on Frontier and Aurora.

## IV. PERFORMANCE

The primary goal of this porting effort was, naturally, to move the computationally intensive parts of GENE to be executed on the GPU. We will present performance results obtained on OLCF's Summit supercomputer, which has "fat nodes" that contain two IBM Power9 CPUs with 512 GB of memory and six NVIDIA V100 GPUs, each with 16 GB HBM memory. Nodes are connected by a dual-port Mellanox EDR InfiniBand network.

We consider a typical plasma core turbulence scenario. Plasma profiles are inspired by the regular cyclone base case, such that ion temperature gradient (ITG) and trapped electron mode (TEM) turbulence develop at all radii. We consider a circular geometry and include electromagnetic effects. The global grid used for the one node run is  $N_x \times N_y \times N_z \times N_{v_{\parallel}} \times N_{\mu} \times N_{\sigma} = 70 \times 16 \times 24 \times 48 \times 32 \times 2$  and is scaled up to  $1120 \times 32 \times 48 \times 96 \times 128$  on 512 nodes as detailed in [Table III](#). In parallel to the increase in  $N_x$ , we increased the box size in this direction proportionally to keep the resolution constant. This corresponds to simulating increasingly larger devices and keeps the same sparsity of the gyro-matrices and the field matrices and therefore the same numerical effort. To complete the description of the scaling, we note that the product  $L_x \rho^*$  is also kept constant by changing the value of  $\rho^*$ . The parameter  $\rho^*$  describes the ratio of the ion gyro radius to the minor radius of the simulated device.

We will first present an overview of how the code runs on the GPUs, compared to running on CPUs only. We then perform a detailed roofline analysis of select GPU kernels that were generated using `GTENSOR`. The data and post processing scripts used to generate the plots in this paper are available on zenodo and github.<sup>47</sup> The repository also includes information on how GENE was built on summit and which branch and commit were used—the GENE source can be obtained separately from the GENE website at <https://genecode.org>.

Finally, we will present and analyze the parallel scalability of GENE, comparing CPU and GPU performance for large runs.



## A. GPU performance overview

Figure 5 shows a trace of GENE as visualized by NVIDIA's Nsight Systems<sup>48</sup> profiler. We have zoomed into a single RK stage, as this presents essentially a complete picture of the computations that GENE performs. A full time step is comprised of four of these RK stages, all nearly identical, and an entire production run consists of tens of thousands of such timesteps, after an initialization phase, neglecting occasional diagnostics and I/O.

The top section shows profile ranges that have been added into GENE. Here, they are used via NVTX;<sup>49</sup> they can be used with various other profiling tools, too. At the top-level is the timeloop (`t_loop`), which for these benchmarks only consists of a smaller number of time-steps. Underneath is a single time step (`eRK_standard`), of which only part is visible, that is, the third RK stage (`eRK_stage_3`). Other than short RK-related computations (in the beginning, the computation of the intermediate state vector for this stage, at the end, saving the result into a temporary vector  $k_3$ ), each RK stage consists of two main parts: `calcaux` computes auxiliary quantities (referred to as just "AUX" in this paper), which are then used in `CalFRhs0` ("RHS") to calculate the rhs of the gyrokinetic equation.

### 1. Computation of the rhs

The computational kernels making up the computation of the rhs were the initial target of the GPU port, as the CPU code spends most (~80%) of its time in these kernels. This part is made up of computing some modified distribution functions and then calculating the terms that makeup the linear part of the rhs and finally computing the nonlinearity. These kernels mostly work on 6D arrays (5D distribution function + *species*), though some quantities are lower-dimensional (e.g., the gyro-averaged potentials do not depend on  $v_{||}$ ).

The bottom section of Fig. 5 shows the GPU utilization, containing the timeline of active kernels in the upper line and memory transfers in the lower line. It shows clearly that the GPU is well utilized, running kernel after kernel with no significant gaps. The only memory transfers between device and host occur as part of MPI communications. (We are using CUDA-aware MPI, but the Spectrum-MPI implementation on Summit goes through host memory under certain circumstances.) The computation itself occurs entirely in device memory. During development, when not all kernels had been moved to the GPU, we observed heavy migration between the host and device, which was very detrimental to performance (the runtime ended up being slower than leaving the computation on the CPU). This confirmed that in order to obtain good

performance on the GPU, we could not afford large memory transfers at the single kernel granularity.

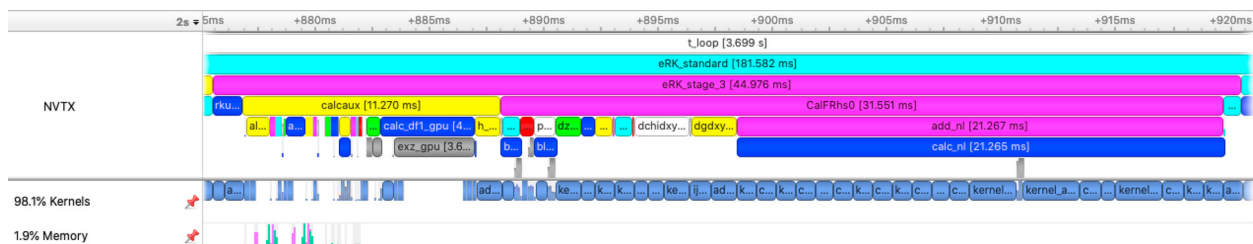
### 2. Computation of the potentials

Once the port of the RHS computation was complete and quite good speed-ups had been achieved, unsurprisingly the remainder of the code, specifically AUX, was now a major bottleneck. This was exacerbated by the fact that the GPU runs are usually performed using just six MPI processes per node on Summit, each using one designated GPU. This generally gives the best GPU performance, having the GPUs work on large kernels at a time, and reduces the amount of communication needed, e.g., for ghost points due to the larger surface/volume ratio. But it also means that any remaining CPU code now runs on just six cores rather than 42 cores. This reduction of resources could have been alleviated by using OpenMP on the host, but since that would still imply frequent transfers between the host and device, the better solution is to port the remaining computations to the GPU.

It was mostly straightforward to port the kernels making up the auxiliary field computation using `GTENSOR`, except for some operations that were not simple stencil computations or reductions. These operations required special handling; two such cases are described below.

AUX computes the electrostatic potential  $\phi$  and the parallel vector potential  $A_{||}$  from velocity space moments of the current distribution function, and it also needs to translate from gyro-centers to actual particle locations and back in the process. These so-called gyro-averaging processes are expressed as banded matrix-vector multiplications, which were also ported using `GTENSOR`, but employing custom kernels using the lambda function approach.

In addition, once the density and parallel current are known, finding the potentials involves solving 3D linear problems. Due to the nature of the discretization and coordinate system, these 3D problems separate into a sequence of 1D problems, which can be solved using a batched LU solve, provided by `cuBLAS`. The LU factorization can be pre-calculated and is therefore not a significant performance issue, only forward and backward substitutions need to be performed for every solve. Due to the finite Larmor radius of the particles, the matrices involved are banded, though with varying bandwidth. Using a full dense solve is not scalable—the work goes like  $N_x^2$ , where  $N_x$  is the number of grid points in the  $x$  direction. While this is not usually an issue in practice due to the fact that the linear solves are only 3D while the main computations are 5D+*species*, an alternate solver based on `cuSPARSE` has been implemented that takes advantage of the sparsity of the matrices.



**FIG. 5.** Trace of the computation of a single Runge-Kutta stage performed by the GPU ported GENE (screenshot taken from the NVIDIA Nsight Systems profiler). The top section shows performance markers delineating different parts of the calculation. The bottom section shows GPU usage in terms of kernels as well as memory transfers between host and device memory.

With AUX completely ported, the original runtime balance of computation of auxiliary quantities vs the rhs on the CPU is roughly restored on the GPU. Due to the lower-dimensional, hence smaller, computations, the GPU is not always fully utilized, but the bigger bottlenecks are the communications involved, both in reductions which use `MPI_Allreduce` as well as ghost point exchanges in preparation for the rhs calculation.

Table I shows a performance comparison between CPU and GPU, broken down by kernel (only the most important kernels are listed). Overall, one time step runs  $14.78\times$  faster on the six GPUs of a single Summit node vs its 42 CPU cores. Broken down by calculation of the auxiliary fields vs the main rhs the speedups are  $12.38\times$  and  $16.07\times$ , respectively. Broken down further, individual terms get accelerated typically by somewhere between  $15\times$  and  $25\times$ , with the non-linearity, which is an expensive, hence important term for performance, being somewhat lower, but still achieving  $12.88\times$ .

Not all of the speedup shown in this table is exclusively due to higher GPU performance. For a CPU run  $7\times$  more MPI processes are used, hence the domain is additionally decomposed sevenfold in the  $x$  direction on the CPU, while this direction is not decomposed on the GPU. Hence, the CPU version has some additional communication costs that are missing from the GPU version. This affects the regions `prdi_h`, `prdi_g` and `prdi fld` as well as the  $x$  boundary exchange in the nonlinearity `add_nl`.

## B. Roofline model of a single Summit node with selected code parts

The advent of multi-core/many-core heterogeneous architectures has significantly increased the barrier to performing objective performance analysis. Quantifying processor performance in a heterogeneous environment while considering the constraining resource of

**TABLE I.** Tabular results from Summit single node run. Shows time per time step in milliseconds for CPU and GPU run, and the speedup achieved on GPU.

Region	CPU	GPU	Speedup
eRK_stage	262.51	17.76	$14.78\times$
eRK_stage.calcaux	63.85	5.16	$12.38\times$
eRK_stage.calcaux.h_from_f	19.78	0.42	$47.55\times$
eRK_stage.calcaux.fldsolve	0.46	0.32	$1.41\times$
eRK_stage.calcaux.ccdens	17.38	1.08	$16.11\times$
eRK_stage.calcaux.calc_dfl	13.27	2.55	$5.21\times$
eRK_stage.calcaux.bar_emf	12.97	0.70	$18.42\times$
eRK_stage.CalFRhs0	186.27	11.59	$16.07\times$
eRK_stage.CalFRhs0.prdi_h	6.72	0.29	$23.33\times$
eRK_stage.CalFRhs0.prdi_g	6.60	0.29	$22.93\times$
eRK_stage.CalFRhs0.prdi fld	6.58	0.24	$27.43\times$
eRK_stage.CalFRhs0.hypz_cmp	7.12	0.24	$29.69\times$
eRK_stage.CalFRhs0.dzv_ak	18.68	0.41	$45.33\times$
eRK_stage.CalFRhs0.dgdx	12.23	0.77	$15.84\times$
eRK_stage.CalFRhs0.dfdxy_h	10.79	0.31	$35.03\times$
eRK_stage.CalFRhs0.dchidxy	11.06	0.77	$14.33\times$
eRK_stage.CalFRhs0.add_nl	102.65	7.97	$12.88\times$

such systems, off-chip memory bandwidth, results in non-trivial performance analysis. In this work, we leverage the roofline model<sup>50</sup> as a powerful yet simple performance analysis framework.

The roofline model visualizes the attainable floating point operations (FLOPs) per second of a program as a function of arithmetic intensity (AI), where AI is defined as the number of FLOPs per byte moved to/from RAM. For large AI (“compute-bound”), peak performance is bounded by a horizontal line representing the theoretical maximum FLOP/s of the processor. For low AI (“memory-bound”), the performance limit is a diagonal line indicating the peak memory bandwidth (byte/sec). The intersection of the two lines designates the balance point of the architecture, where memory and compute bound performance regions meet. Within the memory bound region, peak performance is equal to peak memory bandwidth multiplied by the AI. The roofline model can be extended to capture characteristics of cache hierarchies and the memory subsystem, but we limit our analysis to the basic version that works well for GENE.

In the following, we will analyze the performance of select GENE-generated kernels in detail. This analysis confirms, as expected, that our stencil computations are memory bandwidth limited, but that we get reasonably close to the maximum performance that these kernels can be expected to run at. While not yet widely tested, we anticipate that just knowing the achievable memory bandwidth on a given computer system is going to allow us to predict GENE kernel performance quite well—however, parallel scalability is strongly affected by the latency, bandwidth, and topology of the interconnect network and much harder to model.

### 1. The $dgdx$ term

In a schematic view, the addition of this term to the right-hand side can be written as

$$r_{\sigma}(x, k_y, z, v_{||}, \mu) = r_{\sigma}(x, k_y, z, v_{||}, \mu) + p_x(x, z, v_{||}, \mu, \sigma) \frac{\partial g_{\sigma}(x, k_y, z, v_{||}, \mu)}{\partial x} + p_y(x, z, v_{||}, \mu, \sigma) i k_y g_{\sigma}(x, k_y, z, v_{||}, \mu), \quad (10)$$

with  $r$  being the right-hand side vector,  $g$  being the modified distribution function,  $x$  and  $y$  being the radial and binormal directions, and  $p_x$  and  $p_y$  being the respective prefactors. GENE treats the binormal direction in Fourier space and hence the derivative with respect to  $y$  becomes just a multiplication with  $i k_y$ . The derivative in the  $x$  direction is discretized with centered differences of even order  $p$ , leading to a stencil of

$$\left. \frac{\partial g}{\partial x} \right|_{x_i} \approx \sum_{s=-p/2}^{p/2} c_s g_{i+s}, \quad (11)$$

with the function value  $g_i = g_{\sigma}(x_i, y, z, v_{||}, \mu)$  at the  $x$  grid points  $x_i = -\frac{L_x}{2} + i\Delta x$  with  $i = 0, \dots, n_x - 1$  and the stencil coefficients  $c_s$ . The function values needed at the boundary are set according to appropriate boundary conditions.

The implementation of this term in GENE is split into two steps: In the first kernel, the derivatives are computed and stored and in a second step, those derivatives are multiplied with the prefactors and added to the right-hand side.

This approach makes it easier from a software engineering point of view, as the computation of the derivatives is separated from the right-hand side update, but fusing the two kernels might give better performance. We will derive the arithmetic intensity (AI) of both approaches to compare them.

*a. Two-kernel approach.* To find the arithmetic intensity of the two kernels, we count the number of floating point operations (FLOPs) and the number of data bytes to be read from GPU memory.

For the  $dgdx$  term, the derivative has to be computed for all grid points of the distribution function  $g$  which are  $n_x \cdot n_{k_y} \cdot n_{zv\mu\sigma}$  with  $n_{zv\mu\sigma} \equiv n_z n_{v\parallel} n_{\mu} n_{\sigma}$ . At each grid point, we need  $(4 \cdot p + 2)$  FLOPs for the  $x$  derivative ( $p$  additions of complex numbers and  $p + 1$  multiplications of a real with a complex number) and six FLOPs for the  $y$  derivative (multiplication of two complex numbers).

The data needed are the whole distribution function with two additional boundary points on each side in the  $x$  direction for reading. With 16 bytes per double precision complex number, this gives  $((n_x + 4)n_{k_y} n_{zv\mu\sigma} \cdot 16)$  bytes. For writing there are no boundary points:  $(n_x n_{k_y} n_{zv\mu\sigma} \cdot 16)$  bytes for the  $x$  derivative and the same number of bytes for the  $y$  derivative. We assume double precision for all variables. Putting this all together and canceling common terms gives an arithmetic intensity (AI) for the derivatives kernel of

$$AI_{xyderiv} = \frac{(4p + 2)n_x + 6n_x}{16 \cdot (3n_x + 4)} \frac{\text{FLOP}}{\text{Byte}} \xrightarrow{n_x \text{ large}} \frac{p + 2}{12} \frac{\text{FLOP}}{\text{Byte}}. \quad (12)$$

For the standard case of a centered difference of fourth order ( $p = 4$ ), we get an AI of  $0.5 \frac{\text{FLOP}}{\text{Byte}}$ .

For the update term, we have two multiplications of the real prefactors with complex numbers and two additions of complex numbers leading in total to eight FLOPs per grid points. The data needed for this operations per grid point is  $2n_x n_{k_y} n_{zv\mu\sigma} \cdot 16$  ( $r$  read and write),  $2n_x n_{k_y} n_{zv\mu\sigma} \cdot 16$  (derivatives computed by the previous kernel, read only) and  $2n_x n_{zv\mu\sigma} \cdot 8$  (real prefactors). This leads to an AI of

$$AI_{update} = \frac{8n_{k_y}}{64n_{k_y} + 16} \frac{\text{FLOPs}}{\text{Byte}} \xrightarrow{n_{k_y} \text{ large}} 0.125 \frac{\text{FLOPs}}{\text{Byte}}. \quad (13)$$

*b. Single fused kernel approach.* Fusing both kernels into one gives  $4p + 16$  flops per grid point and needed memory data of  $2n_x n_{k_y} n_{zv\mu\sigma} \cdot 16$  ( $R$  read and write),  $(n_x + 4)n_{k_y} n_{zv\mu\sigma} \cdot 16$  ( $g$ , read only), and  $2n_x n_{zv\mu\sigma} \cdot 8$  (real prefactors) bytes, leading to an AI of

$$AI_{fused} = \frac{(4p + 16)n_x n_{k_y}}{32n_x n_{k_y} + 16(n_x + 4)n_{k_y} + 16n_x} \frac{\text{Flops}}{\text{Byte}} \xrightarrow{n_{k_y}, n_x \text{ large}} \frac{p + 4}{12} \frac{\text{Flops}}{\text{Byte}}. \quad (14)$$

For the default case with  $p = 4$ , we have an AI of  $0.67 \frac{\text{Flops}}{\text{Byte}}$ . As expected for stencil computations, all these AIs are well below the balance point for NVIDIA V100 GPUs, but the fused kernel does help increase it.

*c. Runtime comparison.* To get an idea of which approach is faster in the end, we compute the ratio of the runtimes of the two approaches. All kernels are at an AI that indicates a memory bandwidth bound kernel on the V100, since our data are too large to fit into caches. Therefore, the runtime of a kernel can be computed as

$$T = \frac{D}{BW}, \quad (15)$$

with  $T$  being runtime of the kernel,  $D$  being the data in bytes needed for the kernel, and  $BW$  being the relevant memory bandwidth of the device. The ratio of the kernel runtimes is then

$$\frac{T_{fused}}{T_{deriv} + T_{update}} = \frac{32n_x n_{k_y} + 16(n_x + 4)n_{k_y} + 16n_x}{16 \cdot (3n_x + 4)n_{k_y} + 64n_x n_{k_y} + 16n_x} \xrightarrow{n_{k_y}, n_x \text{ large}} \frac{48}{112} = \frac{3}{7}. \quad (16)$$

Hence, the runtime of the fused kernel is expected to be only 42% of the runtime of the two separate kernels. This finding is independent of the actual bandwidth and holds as long as all kernels are bandwidth limited and the kernel fusion does not change the caching behavior.

*d. Fusing kernels with GTENSOR.* GTENSOR lazily evaluated expressions facilitate fusing kernels. Algorithm 5 shows a non-fused and the fused implementation of  $dgdx$ . Instead of evaluating the  $x$  and  $y$  derivatives into a temporary field, they just get directly used in the final rhs update expression. (Note: it is also possible to assign the unevaluated expressions to intermediate variables without any loss in performance.)

## 2. The dzv term

The  $dzv$  term in GENE involves derivatives of the modified distribution function  $g$  with respect to the field line parallel coordinates,  $z$

**ALGORITHM 5.** Separate and fused GPU implementation of  $dgdx$  using GTENSOR.

```

if 0
//separate calculation of the derivatives
dij.view(_all, _all, 0) =
  x_deriv_5(f, sten, bnd);
dij.view(_all, _all, 1) =
  y_deriv(f, ikj, bnd);

//then update rhs
rhs = rhs +
  p1.view(_all, _newaxis) * dij.view(_all, _all, 0) +
  p2.view(_all, _newaxis) * dij.view(_all, _all, 1);
#else
//generate a single fused kernel
rhs = rhs +
  p1.view(_all, _newaxis) *
  x_deriv_5(f, sten, bnd) +
  p2.view(_all, _newaxis) *
  y_deriv(f, ikj, bnd);
#endif

```

(spatial) and the parallel velocity  $v_{\parallel}$ . The GPU port of GENE uses Arakawa's fourth order method<sup>51</sup> for solving the derivatives, which involves 13-point 2D finite difference stencils. This kernel was chosen for analysis since it has a comparatively high AI for a stencil computation and also because the stencils are in the  $z$  and  $v_{\parallel}$  directions, which are far from stride-1, so it is not *a priori* clear that the GPU caches can handle this situation as well as the  $x$  derivatives.

For the 13-point stencil, we need to compute 13 complex multiplications and 12 complex additions at each grid point, for a total of 50 FLOPs per point. The total number of points where a stencil calculation occurs is the entire problem grid.

The stencil coefficients depend on all coordinates except for  $k_y$ , so these stencil calculations involve more data movement than the dgdx kernels. The extra data movement is offset by higher FLOP count from the larger stencil. The entire distribution function with four ghost points in the  $z$  and  $v_{\parallel}$  dimensions must be read, and this is written to the right hand side vector without ghost points. Grid points are complex double, so 16 bytes per point. The stencil coefficients are real double—8 bytes each. The number of bytes read is therefore  $16(n_{xy\mu\sigma}(n_z + 4)(n_v + 4)) + 8(13n_{xyz\mu\sigma})$ , using the notation  $n_{abc} = n_a n_b n_c$ ,  $n_y \equiv n_{k_y}$ ,  $n_v \equiv n_{v_{\parallel}}$ . The number of bytes written is simply 16 times the total grid size  $n_{xyzv\mu\sigma}$ . After canceling common  $n$  terms, this yields an AI of

$$AI_{zv} = \frac{25n_{yzv}}{16n_{yzv} + 32n_{yz} + 32n_{yv} + 128n_{ky} + 52n_{zv}}, \quad (17)$$

where  $n_{yzv}$ ,  $n_{yz}$ , etc., are defined using the same notation as above.

For the single node run in our scaling study (see Table III), the local domain is  $n_{k_y} = 16$ ,  $n_z = 8$ ,  $n_{v_{\parallel}} = 48$  (the global  $N_z = 24$  is distributed across  $P_z = 3$  MPI processes), yielding an AI for this kernel of around 1.03.

### 3. Measured AI and roofline plots

Using NVIDIA's Nsight Compute<sup>52</sup> profiler and methodology developed by Yang *et al.*,<sup>50,53</sup> we measured run time, FLOPs, and GPU DRAM bytes transferred for the kernels described above on Summit. The parameters used for the scaling test were adapted for a single Summit node and used for this analysis. Figure 6 shows the results.

The measurements confirm that the kernels are bandwidth bound on NVIDIA V100 GPUs, and that the FLOP rate achieved by the kernels is close to the theoretical ceiling. The measured values together with the estimated values can be seen in Table II. Estimated values for FLOP rates and memory transfer rates were calculated using the analysis above for FLOPs and bytes together with the measured time. Measured FLOPs match our estimated values exactly, demonstrating that the NVIDIA hardware and tools provide very accurate flop counters. The memory transferred was slightly higher than estimated for dgdx kernels, and slightly lower for the dzv kernel. It is possible that the dzv kernel is better able to take advantage of caching, so less data must be moved from main memory. Further investigation is required to understand these bandwidth differences.

The fused dgdx kernel runtime is  $\sim 45\%$  of the derivative and update runtimes together, which is nearly as good as the prediction of 42%.

In summary, the roofline analysis shows that our GTENSOR-based approach is getting quite close to hardware limits, specifically, to the

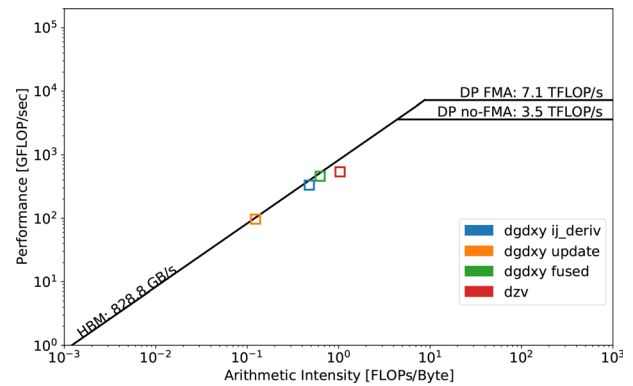


FIG. 6. Double precision, HBM roofline for NVIDIA Tesla V100<sup>50</sup> with kernels described in the text.

memory bandwidth available to GPU HBM memory. This shows that further performance improvement requires an increased AI, or specifically, a reduction in memory transfers which can, for example, be achieved by fusing kernels, or by redundantly recalculating certain quantities, rather than storing and reading them back from memory. It also shows that we cannot expect any significant speed-up by using GPU shared memory to provide a software-managed cache. The latter had been a consideration, and it would be possible to implement a cache blocking algorithm within the generated GTENSOR-kernels, since this is a non-trivial undertaking which does not address a current bottleneck, we left this for future work.

### 4. The nonlinearity

GENE being a pseudo-spectral code, the nonlinearity is computed in real space to avoid an expensive convolution in Fourier space. Usually, the Arakawa discretization<sup>51</sup> is employed. For the details of this method, we refer to the reference, here we just outline the computational steps required to implement it. Six (5D+species) quantities are transformed to real space in  $y$  using batched 1D FFTs, which in the GPU implementation are provided by cuFFT. To avoid introduction of unphysical modes, dealiasing by the two-thirds rule is employed, which requires padding the input fields with an additional 50% zeros and then truncating the final result accordingly. Some relatively simple calculations are performed on the transformed quantities, and the most computationally involved one being a calculation of an

TABLE II. Tabular results from Summit single node roofline analysis. Compares calculated (calc.) and measured (meas.) values for various metrics. The calculated flop and bandwidth rates are based on the absolute values for FLOPs and bytes transferred derived in the text and the measured times, see Fig. 6.

Kernel	Time (ms)	GFLOP/s		GB/s		AI	
		Calc.	Meas.	Calc.	Meas.	Calc.	Meas.
dgdx ij_deriv	0.923	333.2	333.2	679.2	697.1	0.491	0.478
dgdx update	1.059	96.8	96.8	786.8	784.4	0.123	0.123
dgdx fused	0.891	460.2	460.2	717.8	732.5	0.641	0.628
dzv	1.187	539.9	539.9	523.7	520.8	1.03	1.04



$x$ -derivative using finite differences similar to the one shown earlier for the  $\text{dgdxy}$  term. Two partial results are computed, then Fourier-transformed back, combined, and used to update the rhs

The entire computation consists of 20 GPU kernels, of which eight makeup the FFTs. With the exception of the FFT, most of the kernels comprising the computation are of quite low AI. Some kernels even have an AI of 0, that is, they do not perform any floating point operations—these are the kernels that zero-pad the complex arrays before transforming into real space or truncate the results. We have analyzed the achieved memory bandwidth for all of these kernels, eight of which are FFTs or inverse FFTs. The lowest achieved bandwidth is 650 GB/s (out of 830 GB/s), most kernels achieve around 800 GB/s. This means that there is little room for improvement at the individual kernel level and confirms that GTENSOR does a good job of generating kernels that operate close to hardware limits. However, the absolute number of FLOP/s achieved for the whole nonlinearity computation is disappointing at 60 GFLOP/s, which is only 2% of peak. The explanation is the use of numerous temporary fields that cause a lot of memory traffic. They also occupy large amounts of available GPU memory, preventing us from making the local problem size bigger to improve scalability.

If one considers the entire computation of the nonlinearity as a unit, the AI would increase substantially, getting near the crossover from memory-bound to compute-bound that indicates the theoretical opportunity to attain much higher compute performance. However, in practice, it is not easily possible to actually implement the algorithms without temporaries, since the FFTs require input and output arrays to exist in memory as temporary fields. Besides those arrays, we already used GTENSOR to fuse computations where possible. Nevertheless, we may consider writing a custom implementation of the nonlinearity in the future which would reduce the use of temporaries and/or enable the use of blocking. It should be noted, though, that while the nonlinearity is a comparatively expensive term, it is not a big bottleneck, particularly in a large simulation, so even improving the nonlinearity by an additional  $10\times$  would only give us somewhere around 10%–20% speed-up overall due to Amdahl's law.

### C. Parallel scalability

To measure the scalability of our GPU port, we performed a weak parallel scaling test on Summit from 8 to 512 nodes. Weak scaling was chosen because most of the FLOPs on Summit are provided by the GPU, and using them well requires a problem size per GPU that uses enough threads to keep the GPU busy, due to the general low arithmetic intensity of stencil computations. The choice in scaling dimensions, in particular, high number of points in the  $x$  dimension, was motivated by the physical requirements for modeling the type of problem of interest for the WDMAPP project.

GENE decomposes the grid evenly between the processors in all three space and two velocity space plus species dimensions. Hence, the problem size must be chosen such that the grids are divisible by the number of processes in the respective dimensions. While each dimension on its own is not extremely large, the total number of grid points ranges from  $82 \times 10^6$  to  $84 \times 10^9$ . The increase in the problem size and the decomposition is shown in Table III.

For the CPU runs, only the Fortran parts of GENE were compiled and used. We employed 42 MPI processes per node, that is, one MPI process per CPU core while for the GPU runs six MPI processes

**TABLE III.** Problem size and decomposition of the weak scaling runs without  $x$ -parallelization; performance results are shown in Fig. 7. Changes from the previous line are indicated in boldface. The MPI decomposition shown is for the GPU runs with six MPI ranks per node. For pure CPU runs, the  $x$  direction has been additionally decomposed sevenfold to use 42 MPI ranks per node.

#Nodes	$N_x/P_x$	$N_{k_y}/P_{k_y}$	$N_z/P_z$	$N_{v_{\parallel}}/P_{v_{\parallel}}$	$N_{\mu}/P_{\mu}$	$N_{\sigma}/P_{\sigma}$
1	70/1	16/1	24/3	48/1	32/2	2/1
8	<b>280/1</b>	<b>32/1</b>	24/3	<b>48/2</b>	32/8	2/1
16	<b>560/1</b>	32/1	24/3	48/2	32/8	2/2
32	560/1	32/1	24/3	<b>96/4</b>	32/8	2/2
64	560/1	32/1	24/3	96/4	<b>64/16</b>	2/2
128	<b>1120/1</b>	32/1	24/3	96/4	64/32	2/2
256	1120/1	32/1	<b>48/6</b>	96/4	64/32	2/2
512	1120/1	32/1	48/6	96/4	<b>128/64</b>	2/2

per node (one per GPU) were used. GENE does not support multi-threaded operation, so we did not leverage the four-way hyperthreading of each Power9 core.

### 1. Results

For our test problem, the GPU port scales extremely well up to 64 nodes, with a speedup of over  $9\times$  in total time per time step compared to the CPU runs. For 128 nodes and higher, the MPI communication required for the field calculation and filling ghost points starts to dominate. While AUX is still faster on GPU, the total speedup drops to around  $6\times$ . The RHS computation on GPU scales very consistently throughout all the runs, with a speedup over  $12\times$  compared to CPU. This is consistent with our findings from single node performance tracing (see Fig. 5), which shows very good GPU utilization for the RHS section, and MPI communication gaps in the AUX section. Full results can be seen in Fig. 7 and Table IV.

GENE and continuum gyrokinetic solvers, in general, face multi-scale scalability challenges that contribute to the non-ideal weak scalability that we observe.

Due to the high dimensionality, using the surface-to-volume ratio to relatively reduce communication is constrained. In a 3D computational fluid dynamics (CFD) code, for example, one might decompose the problem so that the local subdomain has  $1000^3$  grid points. Each face of the cube would have to communicate data with neighboring processes, that is  $6 \times 1000^2 = 6 \times 10^6$  points, which is a 0.6% ratio of communication to computation. Doing the same computation in 5D, one has ten faces and the ratio of communication to computation is obtained as  $10 \times (10^9)^{4/5}/10^9 \approx 16\%$ .

This shows the problematic general trend, and specifically for GENE, the situation is even more complicated. Two ghostpoints have to be transferred per face. The  $y$ -direction is handled in Fourier space, which means one does not need ghost points there, but for the FFTs in the nonlinearity, one would have to do all-to-all communication to transpose the problem, which is why domain decomposition in the  $y$  direction is generally avoided—but finer decompositions in the actual decomposed directions is the consequence. In the  $\mu$  direction, ghost points are not required unless collisions are included, which helps. Decomposing in the  $x$  direction creates additional complications for gyro-averaging and field solves, as detailed below. The surface-to-

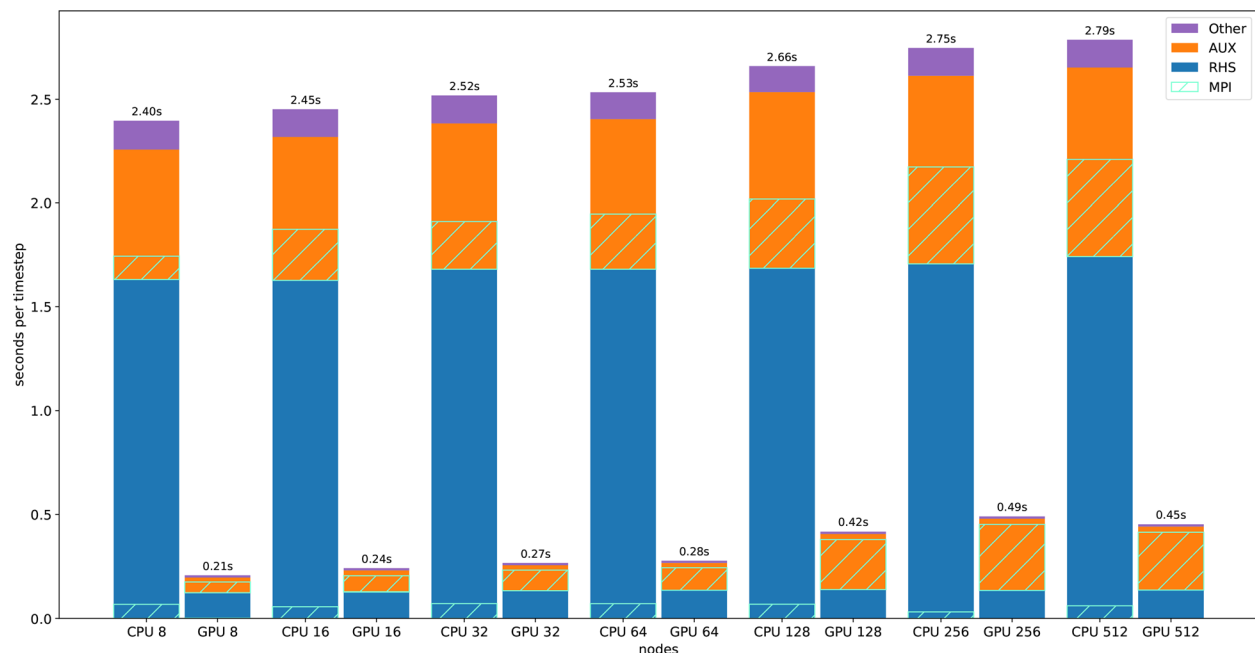


FIG. 7. Weak scaling to 512 Summit nodes, CPU vs GPU.

volume ratio can be increased by increasing the local problem size, however, we are limited by the available GPU memory (16 GB on Summit), and the code requires a number of temporary 5D+*species*-sized fields for the RK4 and the various FFTs in the nonlinearity.

Summit has a powerful interconnect, but it is not enough to avoid off-node boundary exchanges from becoming a bottleneck. One reason for that is while exchanges in the *x* direction involve mostly communication between GPUs in the same node, for an exchange in, e.g., the *z* direction, all six GPU on one node talk to six GPUs on another node, so they all share the same (though dual-port) Infiniband network. One possible approach to improve the ratio of communication to computation is to use the node's large CPU memory in a streaming fashion, overlapping host-device transfers with computation using double buffering, and at the same time overlap off-node communication. This is something we plan to investigate in future work.

The second complication is that a gyrokinetic code is not purely a 5D+*species* code. Before the rhs can be calculated, potentials have to be computed, which requires moment calculations, i.e., reductions and gyro-averaging. Gyro-averaging and the field solve itself are matrix operations. Both of them naively scale like  $N_x^2$ ; however, GENE can take advantage of the sparsity, which makes them algorithmically scalable. The reduced dimensionality of these operations helps to keep their cost down; however, it also means that a naive approach to deal with the dimensional reduction was not efficient on the GPU. On the GPU, the same 5D+*species* domain decomposition would be maintained for the lower-dimensional computations, performing them redundantly across processes in now invariant directions. This worked well on the CPU, but became a bottleneck on the GPU. We therefore implemented a remapping process that handles lower-dimensional operations on a different, non-redundant domain decomposition.

TABLE IV. Tabular results from Summit scaling runs in Fig. 7. Shows time per time step in seconds for CPU and GPU runs, and the speedup achieved on GPU.

# Nodes	RHS			AUX			Other			Total		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
8	1.631	0.124	13.2×	0.626	0.072	8.7×	0.139	0.011	12.6×	2.396	0.207	11.6×
16	1.627	0.127	12.8×	0.691	0.103	6.7×	0.134	0.011	12.2×	2.452	0.241	10.2×
32	1.681	0.132	12.7×	0.702	0.123	5.7×	0.135	0.011	12.3×	2.518	0.266	9.5×
64	1.681	0.134	12.5×	0.723	0.132	5.5×	0.130	0.011	11.8×	2.534	0.277	9.1×
128	1.686	0.138	12.3×	0.849	0.268	3.2×	0.124	0.011	11.1×	2.658	0.417	6.4×
256	1.707	0.134	12.8×	0.905	0.346	2.6×	0.135	0.011	11.9×	2.746	0.491	5.6×
512	1.742	0.135	12.9×	0.910	0.306	3.0×	0.134	0.011	12.1×	2.786	0.453	6.2×

While it comes with its own communication cost, this significantly reduces the computational work in the AUX calculation. It also solved another problem, that is, domain decomposed matrix operations (which are only relevant in the  $x$  direction) can be avoided by remapping to a domain that is not decomposed in the  $x$  direction, allowing us to use non-distributed algorithms that are much better suited for the GPU. Nevertheless, as seen above, scalability still has significant room for improvement, which we will revisit as new machines become available, since hardware characteristics, in particular, the interconnect network have a large impact on GENE's performance.

## V. CONCLUSIONS

In this work, we have described the design of the GPU port of GENE. While the path toward our solution was somewhat convoluted, the eventual approach of preserving Fortran kernels but adding GPU implementations separately in CUDA/C++ using the `GTENSOR` library has in the end worked well for us. It clearly comes with a certain amount of code duplication, but it also preserves simplicity. In our opinion, it is worth noting that it does not seem likely that there will ever be a perfect "performance portability" solution. While it is possible to generate implementations for very different hardware from a common source using approaches like OpenMP or Kokkos/RAJA/GTENSOR, as we have found in our work, different architectures can require different formulations of the algorithm to really perform well. CPUs, which execute a relatively small number of threads at once, can work well with a small sub-problem that can be chosen to fit in cache, while GPUs on low-AI problems need to work on large problem sizes to keep all execution units busy. Similarly, the batched LU solve on the CPU performed quite well even while decomposed with MPI due to the fact that communication and computation of the batched problems could be overlapped, while on the GPU it is better to remap the problem in a way that the batched solves can be executed concurrently on a single GPU. The point here is that it is important not only to retain flexibility in how algorithms are implemented and make common parts reusable, but also to accept the fact that not all hardware is the same and does need custom solutions.

The main porting work, once our approach had been crystallized, was fairly rapid (most of it was done within a few months). Long term maintainability is something we cannot finally judge yet, but we believe that we are on a sustainable path.

Single node speed-ups are very good, on the order of  $15\times$ , which is roughly consistent with the  $16\times$  memory bandwidth ratio between GPUs and CPUs on Summit. Our roofline analysis shows that we are getting quite close to hardware limits but are clearly on the memory-bound branch. This means that there is room for improvement if the AI can be increased, and there are certainly opportunities to do so, in particular, by fusing more kernels, which, however, also would require more refactoring of the existing Fortran code structure.

In terms of parallel scalability, our results show that while the scaling performs reasonably well, communication does become a clear bottleneck for large node counts. While that behavior can be understood given the complexity of the system of equations we are solving, there remain a number of avenues to further improve the scalability in future work. One is to explore more complex strategies in remapping the lower-dimensional computations that are part of the AUX calculation, including also the mapping of the 6D domain decomposition to MPI processes in a hardware/topology aware fashion. In addition,

more overlap of communication and computation may be achievable by subdividing the computations on the GPU into a sequence of moderately sized blocks, a strategy already supported on the CPU. Finally, a worthwhile goal is to increase the maximum local problem size that can fit on each GPU. This is generally helpful as available memory can be a severe constraint for large multi-scale physics simulations. In addition, it improves scalability by increasing the ratio of computation to communication. One avenue is to further reduce the use of temporary fields on the GPU; the aforementioned blocking approach also helps there. Finally, one could take advantage of the large available CPU memory by performing an entire RK stage in a streaming fashion, which would likely be enough computational work to hide the large host-device memory transfers that it entails.

The limitations that we encountered are closely related to hardware characteristics like on- and off-node communication latencies and bandwidths, available GPU memory, balance points between memory bandwidth and computational capacity, etc. It is therefore imperative that, as new systems come online, we analyze performance and adapt the code accordingly.

## ACKNOWLEDGMENTS

We would like to thank Charlene Yang for help in understanding the subtleties of the NVIDIA GPU metrics needed for roofline analysis.

This research was supported by the Exascale Computing Project (No. 17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. 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. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

## DATA AVAILABILITY

The data that support the findings of this study are openly available in zenodo and github at <https://doi.org/10.5281/zenodo.4390089>, Ref. 47. Detailed simulation data are available from the corresponding author upon reasonable request.

## REFERENCES

- <sup>1</sup>See [wdmapp.pppl.gov](http://wdmapp.pppl.gov) for a description of the WDMapp Exascale Computing Project.
- <sup>2</sup>F. Jenko, W. Dorland, M. Kotschenreuther, and B. N. Rogers, "Electron temperature gradient driven turbulence," *Phys. Plasmas* **7**, 1904–1910 (2000).
- <sup>3</sup>T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, and D. Told, *J. Comput. Phys.* **230**, 7053–7071 (2011).
- <sup>4</sup>S. Ku, C. S. Chang, R. Hager, R. M. Churchill, G. R. Tynan, I. Cziegler, M. Greenwald, J. Hughes, S. E. Parker, M. F. Adams, E. D'Azevedo, and P. Worley, "A fast low-to-high confinement mode bifurcation dynamics in the boundary-plasma gyrokinetic code `xgcl`," *Phys. Plasmas* **25**, 056107 (2018).
- <sup>5</sup>G. Merlo, S. Janhunen, F. Jenko, A. Bhattacharjee, C. S. Chang, J. Cheng, P. Davis, J. Dominski, K. Germaschewski, R. Hager, S. Klasky, S. Parker, and E. Suchyta, "First coupled gene-xgc microturbulence simulations," *Phys. Plasmas* **28**, 012303 (2021).

- <sup>6</sup>J. Cheng, J. Dominski, Y. Chen, H. Chen, G. Merlo, S.-H. Ku, R. Hager, C.-S. Chang, E. Suchyta, E. D'Azevedo, S. Ethier, S. Sreepathi, S. Klasky, F. Jenko, A. Bhattarjee, and S. Parker, "Spatial core-edge coupling of the particle-in-cell gyrokinetic codes gem and xgc," *Phys. Plasmas* **27**, 122510 (2020).
- <sup>7</sup>J. Dominski, J. Cheng, G. Merlo, V. Carey, R. Hager, L. Ricketson, J. Choi, S. Ethier, K. Germaschewski, S. Ku, A. Mollen, N. Podhorski, D. Pugmire, E. Suchyta, P. Trivedi, R. Wang, C. S. Chang, J. Hittinger, F. Jenko, S. Klasky, S. E. Parker, and A. Bhattarjee, "Spatial coupling of gyrokinetic simulations, a generalized scheme based on first-principles," *Phys. Plasmas* **28**, 022301 (2021).
- <sup>8</sup>T. Dannert, A. Marek, and M. Rapp, "Porting large hpc applications to gpu clusters: The codes gene and vertex," *Parallel Comput.: Accel. Comput. Sci. Eng.* **25**, 305–314 (2013).
- <sup>9</sup>H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussmann, "Picongpu: A fully relativistic particle-in-cell code for a gpu cluster," *IEEE Trans. Plasma Sci.* **38**, 2831–2839 (2010).
- <sup>10</sup>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," in *High Performance Computing*, edited by J. M. Kunkel, R. Yokota, M. Tafer, and J. Shalf (Springer International Publishing, Cham, 2017), pp. 496–514.
- <sup>11</sup>J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thévenet, and W. Zhang, "Warp-x: A new exascale computing platform for beam-plasma simulations," *Nucl. Instrum. Methods Phys. Res. Sect. A* **909**, 476–479 (2018).
- <sup>12</sup>W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. P. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "Amrex: A framework for block-structured adaptive mesh refinement," *J. Open Source Software* **4**, 1370 (2019).
- <sup>13</sup>M. Wiesenberger, L. Einkemmer, M. Held, A. Gutierrez-Milla, X. Sáez, and R. Iakymchuk, "Reproducibility, accuracy and performance of the feltor code and library on parallel computer architectures," *Comput. Phys. Commun.* **238**, 145–156 (2019).
- <sup>14</sup>N. Ohana, C. Gheller, E. Lanti, A. Jocksch, S. Brunner, and L. Villard, "Gyrokinetic simulations on many- and multi-core architectures with the global electromagnetic particle-in-cell code orb5," *Comput. Phys. Commun.* **262**, 107208 (2021).
- <sup>15</sup>ECP-CoPA, "Cabana - a co-designed library for exascale particle simulations," <https://github.com/ECP-copa/Cabana>.
- <sup>16</sup>H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.* **74**, 3202–3216 (2014).
- <sup>17</sup>S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Z. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: A data-driven literature analysis," *ACM Comput. Surv.* **53**, 23 (2020).
- <sup>18</sup>D. Hatch, M. Kotschenreuther, S. Mahajan, G. Merlo, A. Field, C. Giroud, J. Hillesheim, C. Maggi, C. P. von Thun, C. Roach, S. Saarelma, and JET Contributors, "Direct gyrokinetic comparison of pedestal transport in JET with carbon and ITER-like walls," *Nucl. Fusion* **59**, 086056 (2019).
- <sup>19</sup>G. Merlo, M. Fontana, S. Coda, D. Hatch, S. Janhunen, L. Porte, and F. Jenko, "Turbulent transport in tcv plasmas with positive and negative triangularity," *Phys. Plasmas* **26**, 102302 (2019).
- <sup>20</sup>A. D. Siena, T. Görler, E. Poli, A. B. Navarro, A. Biancalani, and F. Jenko, "Electromagnetic turbulence suppression by energetic particle driven modes," *Nucl. Fusion* **59**, 124001 (2019).
- <sup>21</sup>J. Garcia, T. Görler, F. Jenko, and G. Giruzzi, "Gyrokinetic nonlinear isotope effects in tokamak plasmas," *Nucl. Fusion* **57**, 014007 (2017).
- <sup>22</sup>T. Görler, A. E. White, D. Told, F. Jenko, C. Holland, and T. L. Rhodes, "A flux-matched gyrokinetic analysis of diii-d l-mode turbulence," *Phys. Plasmas* **21**, 122307 (2014).
- <sup>23</sup>D. Grošelj, S. S. Cerri, A. B. Navarro, C. Willmott, D. Told, N. F. Loureiro, F. Califano, and F. Jenko, "Fully kinetic versus reduced-kinetic modeling of collisionless plasma turbulence," *Astrophys. J.* **847**, 28 (2017).
- <sup>24</sup>D. Told, F. Jenko, J. M. TenBerge, G. G. Howes, and G. W. Hammett, "Multiscale nature of the dissipation range in gyrokinetic simulations of alfvénic turbulence," *Phys. Rev. Lett.* **115**, 025003 (2015).
- <sup>25</sup>M. J. Pueschel, D. Told, P. W. Terry, F. Jenko, E. G. Zweibel, V. Zhdankin, and H. Lesch, "Magnetic reconnection turbulence in strong guide fields: basic properties and application to coronal heating," *Astrophys. J. Suppl. Ser.* **213**, 30 (2014).
- <sup>26</sup>A. Brizard and T. Hahm, "Foundations of nonlinear gyrokinetic theory," *Rev. Mod. Phys.* **79**, 421 (2007).
- <sup>27</sup>Various collisions' operators can be used in GENE; however, they are not indicated for clarity as they have not yet been optimized for GPU usage.
- <sup>28</sup>E. Gamma, R. Helm, R. Johnson, and J. Vliissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series (Pearson Education, 1994).
- <sup>29</sup>T. Görler, N. Tronko, W. A. Hornsby, A. Bottino, R. Kleiber, C. Norscini, V. Grandgirard, F. Jenko, and E. Sonnendrücker, "Intercode comparison of gyrokinetic global electromagnetic modes," *Phys. Plasmas* **23**, 072503 (2016).
- <sup>30</sup>G. Merlo, J. Dominski, A. Bhattarjee, C. S. Chang, F. Jenko, S. Ku, E. Lanti, and S. Parker, "Cross-verification of the global gyrokinetic codes gene and xgc," *Phys. Plasmas* **25**, 062308 (2018).
- <sup>31</sup>C. Lechte, G. D. Conway, T. Görler, T. Happel, and ASDEX Upgrade Team, "Fullwave doppler reflectometry simulations for density turbulence spectra in ASDEX upgrade using GENE and IPF-FD3d," *Plasma Sci. Technol.* **22**, 064006 (2020).
- <sup>32</sup>G. Merlo, S. Brunner, Z. Huang, S. Coda, T. Görler, L. Villard, A. B. Navarro, J. Dominski, M. Fontana, F. Jenko, L. Porte, and D. Told, "Investigating the radial structure of axisymmetric fluctuations in the TCV tokamak with local and global gyrokinetic GENE simulations," *Plasma Phys. Controlled Fusion* **60**, 034003 (2018).
- <sup>33</sup>X. Lapillonne, B. F. McMillan, T. Görler, S. Brunner, T. Dannert, F. Jenko, F. Merz, and L. Villard, "Nonlinear quasisteady state benchmark of global gyrokinetic codes," *Phys. Plasmas* **17**, 112321 (2010).
- <sup>34</sup>S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Numerical reproducibility for the parallel reduction on multi- and many-core architectures," *Parallel Comput.* **49**, 83–97 (2015).
- <sup>35</sup>H. Lederer, R. Tisma, R. Hatzky, A. Bottino, and F. Jenko, "Application enabling in DEISA: Petascale of plasma turbulence codes," *Adv. Parallel Comput.* **15**, 713–720 (2007); available at <https://www.worldcat.org/title/parallel-computing-architectures-algorithms-and-applications/oclc/1050624789>.
- <sup>36</sup>T. Dannert, T. Görler, F. Jenko, and F. Merz, "Gyrokinetic turbulence simulation with GENE," in *Jülich Blue Gene/P Extreme Scaling Workshop 2009*, No. FZJ-JSC-IB-2010-02, edited by B. Mohr and W. Frings (2009).
- <sup>37</sup>OpenACC-Standard.org, "The OpenACC application programming interface version 3.0," <https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf> (2019).
- <sup>38</sup>OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf> (2015).
- <sup>39</sup>D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland, "Raja: Portable performance for large-scale scientific applications," in 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (2019).
- <sup>40</sup>K. Germaschewski and B. Allen, "wdmapp/tensor: Physics of plasmas special issue gene version," <https://doi.org/10.5281/zenodo.4385420> (2020).
- <sup>41</sup>See [https://www.boost.org/doc/libs/1\\_63\\_0/libs/multi\\_array/doc/user.html](https://www.boost.org/doc/libs/1_63_0/libs/multi_array/doc/user.html) for "The boost multidimensional array library" (2020).
- <sup>42</sup>G. Guennebaud, B. Jacob, et al., "Eigen v3," <http://eigen.tuxfamily.org> (2010).
- <sup>43</sup>J. Mabile, S. Corlay, W. Vollprecht, and QuantStack, "C++ tensors with broadcasting and lazy computing," <https://github.com/xtensor-stack/xtensor> (2021).
- <sup>44</sup>T. E. Oliphant, *Guide to NumPy* (CreateSpace Independent Publishing Platform, 2015).
- <sup>45</sup>N. Bell and J. Hoberock, "Thrust: Productivity-oriented library for cuda," *Astrophys. Source Code Lib.* **7**, 12014 (2012); available at <https://ui.adsabs.harvard.edu/abs/2012ascl.soft12014B/abstract>.
- <sup>46</sup>T. Veldhuizen, "Using c++ template metaprograms," *C++ Rep.* **7**, 36–43 (1995); available at <https://scholar.google.com/citations?user=9vJnOMMAAAJ&hl=en>.



<sup>47</sup>B. Allen (2020). “wdmapp/gene-paper-artifacts-pop2020,” [Zenodo and github](https://doi.org/10.5281/zenodo.4390089). <https://doi.org/10.5281/zenodo.4390089>.

<sup>48</sup>See <https://developer.nvidia.com/nsight-systems> for “NVIDIA Nsight Systems.”

<sup>49</sup>See <https://github.com/NVIDIA/NVTX> for “NVIDIA Tools Extension.”

<sup>50</sup>C. Yang, T. Kurth, and S. Williams, “Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc- 9 perlmutter system,” *Concurrency Comput.: Practice Experience* **32**, e5547 (2019).

<sup>51</sup>A. Arakawa, “Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. Part I,” *J. Comput. Phys.* **1**, 119–143 (1966).

<sup>52</sup>See <https://developer.nvidia.com/nsight-compute> for “NVIDIA Nsight Compute.”

<sup>53</sup>See <https://gitlab.com/NERSC/roofline-on-nvidia-gpus/> for “Roofline on nvidia gpus.”