A GPU-parallelization of the neXtSIM-DG dynamical core (v0.3.1)

Robert Jendersie[1,2], Christian Lessig[1,3], and Thomas Richter[2]
[1]Institute for Simulation and Graphics, Otto-von-Guericke University, Magdeburg, Germany
[2]Institute of Analysis und Numerics, Otto-von-Guericke University, Magdeburg, Germany
[3]European Centre for Medium-Range Weather Forecasts, Bonn, Germany
Correspondence: Robert Jendersie (robert.jendersie@ovgu.de)

Abstract. The cryosphere plays a crucial role in Earth's climate system, making accurate sea ice simulation essential for improving climate projections. To achieve higher resolution simulations, graphics processing units (GPUs) have become increasingly appealing due to their higher floating point peak performance and superior energy efficiency compared to CPUs. However, harnessing the full theoretical performance of GPUs often requires significant effort in redesigning algorithms and careful implementation. Recently, several frameworks have emerged, aiming to simplify general-purpose GPU programming. In this study, we evaluate multiple such frameworks, including CUDA, SYCL, Kokkos, and PyTorch, for the parallelization of neXtSIM-DG, a finite-element-based dynamical core for sea ice. Based on our assessment of usability and performance, CUDA demonstrates the best performance, while Kokkos is a suitable option for its robust heterogeneous computing capabilities. Our complete implementation of the momentum equation using Kokkos achieves a sixfold speedup on the GPU compared to our OpenMP-based CPU code, while maintaining competitiveness when run on the CPU. Additionally, we explore the impact of different discretization orders and the use of lower precision floating-point types on the GPU, showing that switching to single precision can further accelerate sea ice codes.

1 Introduction

Simulations are essential for understanding the effects of climate change and enabling stakeholders to mitigate its impact on societies and individuals (Jakob et al., 2023). The cyrosphere is a key component of Earth's climate system and it has a particular impact long-term processes. neXtSIM-DG is a novel sea ice code that is designed as part of The Scale-Aware Sea Ice Project (SASIP)[1] to improve both the representation of physical processes and the efficiency and accuracy of numerical implementation.

A crucial factor for the fidelity and reliability of climate simulations is horizontal resolution, with kilometer-scale, convection-resolving simulations being the target for the next generation of models (Stevens et al., 2019; Bauer et al., 2021b). These computations require exascale HPC systems with substantial GPU-based accelerators (Schär et al., 2020; Bauer et al., 2021a). As a result, significant efforts have been dedicated to porting components of existing climate models to GPUs (Ikuyajolu et al., 2023; Sauer and Muñoz-Esparza, 2020; Cao et al., 2023; Sun et al., 2023).

[1]https://sasip-climate.github.io

In line with these developments, SASIP aims to substantially increase the accuracy of the sea ice component of coupled climate models by using modern higher order discretizations of the governing equations. Kilometer-scale sea ice simulations are also of substantial theoretical interest, as initial investigations show a marked change in the behavior of the commonly used viscous plastic sea ice model when approaching km-scale resolutions (Bouchat et al., 2022; Hutter et al., 2022). Computationally much more efficient models are needed for detailed investigations of this behavior, its relationship with known statistical and physical principles (Marsan et al., 2004), and the role of possible alternative models (Dansereau et al., 2016; Ólason et al., 2022). Kilometer-scale or even higher resolved sea ice forecasts are also in high demand among sea ice forecast users (e.g. Kauker et al. (2021)), even though it is unclear how current systems should be used at such resolutions (Hunke et al., 2020). The role of small-scale sea ice features, such as ridges and leads, in atmosphere–ocean–ice interactions in weather and climate models can also only be speculated on with current 10 km scale resolution models (Esau, 2007; Marcq and Weiss, 2012; Ólason et al., 2021). Highly efficient kilometer-scale sea ice models, either as stand-alone sea ice models or as sea ice components in fully coupled climate models, are needed to address these questions.

The neXtSIM-DG dynamical core currently builds on the established viscous plastic sea ice model (Hibler, 1979) and uses a discretization with a higher order discontinuous/continuous Galerkin method (Richter et al., 2023). At the heart of the numerical implementation is a large number of identical operations to be carried out on each mesh element. GPUs, which are based on a data parallel processing model are ideally suited for this kind of tasks. The GPU parallelization of a finite-difference solver for sea ice is investigated by Rasmussen et al. (2024).

Besides performance, other factors are also important for the implementation of the neXtSIM-DG dynamical core: the code must be adaptable to future hardware, a long-term support of the software must be guaranteed and is should be easy to use. Such considerations are vital for the model's adoption and sustained use. With these aspects in mind, we examine the current landscape of general-purpose GPU programming frameworks and compare prototype implementations for the parallelization of finite-element/finite-volume codes like neXtSIM-DG.

The de facto standard is CUDA (NVIDIA, 2023a), a highly popular framework for general-purpose GPU programming, though it only supports NVIDIA hardware. Additionally, it often necessitates a dedicated GPU implementation, requiring significant effort for development and performance tuning. As a result, various alternatives have emerged, offering greater flexibility and usability. A focus on ease of use and minimal effort in port to GPUs is found in frameworks like OpenMP and OpenACC. These, however, offer less control and fewer performance optimization options compared to CUDA.

SYCL (The Khronos Group, 2023a) and Kokkos (Trott et al., 2022) are frameworks designed for heterogeneous computing, enabling targeting of different compute hardware. A major challenge with these frameworks is achieving near-optimal performance across diverse hardware platforms, particularly in efficiently leveraging GPU-specific features such as shared memory or tensor cores.

Another recent alternative is the use of libraries such as jax (Bradbury et al., 2018) and PyTorch (Paszke et al., 2019), which were primarily developed for machine learning. Their backends are built upon high-performance linear algebra libraries that support various hardware architectures and include compilers that efficiently map computations onto these platforms. Examples of such backends include XLA (xla, 2023), Triton (Tillet et al., 2019), and TensorRT (NVIDIA, 2023b).

With machine learning being a driving force in the development of new accelerator hardware, significant resources are invested in lower precision floating point support. Since these types offer orders of magnitude higher throughput than double precision, which is commonly used in scientific computing, mixed precision is a promising direction for further speedups. Analysis of weather simulations with a reduced-precision emulator indicates that the precision can be reduced for most variables, in some cases down to half precision, without degrading results (Hatfield et al., 2019; Tintó Prims et al., 2019). For example, the Integrated Forecasting System (IFS) has successfully switched to single precision without requiring major code changes (Lang et al., 2021). Climate simulations with a reduced-precision emulator suggest that single precision can be equally accurate even for long term runs (Paxton et al., 2022; Kimpson et al., 2023; Banderier et al., 2024).

To compare the GPU programming frameworks and to investigate the potential speedup of an optimized GPU implementation, we use the different frameworks to port an important part of the neXtSIM-DG dynamical core to run on GPU. Our results show that SYCL is still immature, suffering from an unreliable toolchain. Dedicated CUDA remains the best option for speed, while Kokkos provides comparable performance and greater flexibility. PyTorch is currently not a viable alternative to handwritten C++ code, but the new compiler TorchInductor shows promise. We also find that single precision is a strong option to further accelerate our sea-ice simulation. Based on our evaluation, we use Kokkos to implement the complete momentum equation on GPU, for which we achieve a speedup by a factor of 6 compared to a multi-core CPU implementation.

The structure of the paper is as follows. We start in Section Sect. 2 with an overview of the neXtSIM-DG dynamical core. The different GPU implementations of it are detailed in Sect. 3. In the subsequent Sect. 4, their performance is compared and the impact of mixed precision as well as higher order discretizations is analyzed. In Sect. 5, the Kokkos implementation is extended and practical results are presented. Finally, directions for future work and a summary is provided in Sect. 6.

2 Model description

For simplicity of the presentation we restrict ourselves to the sea ice model introduced by Hibler (Hibler, 1979). For a discussion on different material laws we refer to the literature (Feltham, 2008). This model consists of advection equations for the ice height H and ice concentration A

$$\partial_t H + \operatorname{div}(\mathbf{v}H) = S_H, \qquad \partial_t A + \operatorname{div}(\mathbf{v}A) = S_A, \tag{1}$$

where the right hand side terms S_H and S_A are describing thermodynamics. In addition, the 2d velocity field $\mathbf{v}$ is governed by the momentum equation

$$\rho_{\text{ice}} H \partial_t \mathbf{v} = \operatorname{div} \boldsymbol{\sigma}(\mathbf{v}, A, H) + F, \tag{2}$$

where ρ_{ice} is the ice density and σ the stress tensor. In F we collect all external forcings that come from wind and ocean current. Implicit solvers for the Hibler model suffer from the strong nonlinearities, are costly and call for advanced numerical solution strategies (Mehlmann and Richter, 2017). Hence, often often explicit iterations in the sense of a pseudo-time stepping is used, namely the mEVP solver (Bouillon et al., 2013)

$$\boldsymbol{\sigma}^{(p)} = \frac{\alpha}{1+\alpha}\boldsymbol{\sigma}^{(p-1)} + \frac{1}{1+\alpha}\boldsymbol{\sigma}_{vp}(\mathbf{v}^{(p-1)}, A, H), \qquad \boldsymbol{\sigma}_{vp}(\mathbf{v}, A, H) = \eta(\nabla\mathbf{v} + \nabla\mathbf{v}^T) + \zeta\,\mathrm{div}(\mathbf{v})I - \frac{P}{2}I. \tag{3}$$

Here, the viscosities η, ζ depend on velocity $\mathbf{v}$, ice height H and ice concentration A, while the ice strength P depends only on H and A. The parameter $\alpha > 0$ controls the stability and the speed of convergence. This approach is also the basis of the neXtSIM-DG implementation. While the advection problems Eq. (1) are solved using a large time step, the momentum equation Eq. (2) and the mEVP iteration Eq. (3) are subcycled with a smaller step size. Often more than 100 substeps are required in each advection step and the main effort lies in the repeated evaluation of the nonlinear material law, see $\sigma(\mathbf{v}, A, H)$ in Eq. (3). All details on the model and its discretization in neXtSIM-DG are given in Richter et al. (2023).

2.1 Discretization

We briefly sketch the discretization of the model in the neXtSIM-DG dynamical core and refer to (Richter et al., 2023) for details. Equations (1) to (3) are discretized on quadrilateral meshes in spherical coordinates. This mesh is topologically fully structured but each element is mapped from a uniform reference element onto the computational element to allow better alignment with coastlines and a more equal mesh spacing. For the advection equations higher order discontinuous Galerkin upwind methods and high order explicit Runge-Kutta schemes are used. The velocity $\mathbf{v}$ is instead discretized using a continuous Galerkin approach. As the velocity stress coupling has the form of a mixed formulation, a discontinuous Galerkin space is used to represent the stresses. This space must include the gradient of the velocity space for stability.

3 Implementation

Starting point for the GPU implementation of neXtSIM-DG dynamical core is the C++ CPU implementation that is described in (Richter et al., 2023). This OpenMP parallelized CPU code also serves as the baseline for the performance evaluation. The CPU implementation leverages the linear algebra library Eigen (Guennebaud et al., 2010), which is highly optimized and, e.g., exploits CPU vector units. Due to the explicit character of the discretization, and the parametric finite element setup, most computations are matrix-vector or matrix-matrix products with small matrices and vectors, e.g. vectors of size 4. As several vectors and matrices are already available at compile time, e.g. all quantities that refer to the reference element, neXtSIM-DG greatly benefits from Eigen's template-based design. These fixed size matrices do not require dynamic memory allocation and operations involving such matrices can be fully loop-unrolled. Also, the use of expression templates in Eigen eliminates unnecessary temporary variables in expressions involving multiple operations.

Table 1 indicates computational times for the different parts of the dynamical core in a typical sea ice dynamics simulation. The mEVP iteration (middle lines of the table from "strain" to "velocity") takes most of the time and the stress update is the single most expensive part. These computations are local on each mesh element and hence scale well with more cores. They are further well suited as computational unit for the evaluation of the different GPU programming frameworks. A pseudocode overview of the stress update computations is shown in Listing 1. The original C++ code is documented in Appendix B. Unless

	serial [s]	OpenMP [s]	speedup
advection	188.32	34.34	5.48
boundary	38.30	11.25	3.40
strain	918.84	185.28	4.96
stress	**1741.43**	**206.36**	8.44
divergence	1023.07	170.81	5.99
velocity	728.80	85.73	**8.50**
other	14.42	5.98	2.41
total	4653.2	699.75	6.65

Table 1. Runtime of 120 time-steps of the simulation with 2.6×10^5 elements on a 10 core CPU. Except for the advection, all major computations are part of the mEVP iteration which performs 100 sub-steps in each time-step. Remaining operations, e.g. i/o and external forcing, are summarized by "other".

otherwise stated all numerical testcases use double precision. The code ,however, is generic in this respect and Section 4.2 will study the effect of using lower or mixed precision arithmetics.

For readers unfamiliar with GPU architecture and the specifics of GPU programming, we provide a brief introduction in Appendix A to complement the following text.

3.1 CUDA

The standard for general purpose GPU programming is CUDA (NVIDIA, 2023a), a C++ based language and API developed by NVIDIA. CUDA has a mature ecosystem and gives low level access to the GPU, which allows one to develop highly optimized code. However, CUDA is limited to NVIDIA hardware and the development effort to obtain code with a high utilization of the available compute resources can be considerable.

Since version 3.3, Eigen has limited support for CUDA and allows one to use fixed sized matrices in CUDA kernels. Through this, we can use the code from Listing 1 largely unchanged. Eigen's manually vectorized code paths need to be disabled to have the code run on the GPU, we still benefit from Eigen's other features such as expression templates and optimizations when a size is known at compile time. For the use of CUDA with Eigen, we have to ensure that the required data is in GPU memory. For dynamic buffers like S^{11} in Listing 1, we allocate memory manually and copy it as needed before and after the kernel invocation. Inside the CUDA kernel, an Eigen::Map is constructed with

```
auto B = Map<Matrix<T, Dynamic,n>>(bufDevice, N, n) .
```

This provides the same interface as the original matrix. For compile time matrices such as PSI we use the GPU's *constant memory*. Advantages of constant memory are that no manual memory management is required, faster memory access through a dedicated cache, and that further compiler optimizations are possible since the values are available at compile time. In the

```
void StressUpdateHighOrder(Matrix<N,$n_S$>& $S^{11}$, Matrix<N,$n_S$>& $S^{12}$, Matrix<N,$n_S$>& $S^{22}$,
        const Matrix<N,$n_S$>& $E^{11}$, const Matrix<N,$n_S$>& $E^{12}$, const Matrix<N,$n_S$>& $E^{22}$,
        const Matrix<N,$n_A$>& $H$, const Matrix<N,$n_A$>& $A$, double $\alpha$) {
        for ($i=0$; $i<N$; $++i$) { // in parallel
                Vector<$n_G$> $h$ = $\max\{0, H_{i,*}\mathrm{PSI}\langle n_A\rangle\}$
                Vector<$n_G$> $a$ = $\min\{1,\max\{0, A_{i,*}\mathrm{PSI}\langle n_A\rangle\}\}$
                Vector<$n_G$> $e^{11}$ = $E^{11}_{i,*}\mathrm{PSI}\langle n_S\rangle$
                Vector<$n_G$> $e^{12}$ = $E^{12}_{i,*}\mathrm{PSI}\langle n_S\rangle$
                Vector<$n_G$> $e^{22}$ = $E^{22}_{i,*}\mathrm{PSI}\langle n_S\rangle$

                Vector<$n_G$> $P$   = $P^\star \cdot h * \exp\big(-20(1-a)\big)$
                Vector<$n_G$> $D$   = $\big(\Delta^2_{\text{min}} + \frac{5}{4}(E^{11}_{i,*} * E^{11}_{i,*} + E^{22}_{i,*} * E^{22}_{i,*}) + \frac{3}{2}E^{11}_{i,*} * E^{22}_{i,*} + E^{12}_{i,*} * E^{12}_{i,*}\big)^{\frac{1}{2}}$
                Vector<$n_G$> $P_D$ = $P/D$

                $S^{11}_{i,*}$ = $(1-\alpha^{-1})S^{11}_{i,*} + \alpha^{-1}M_i^{-1}\big(P_D * (\frac{5}{8}e^{11} + \frac{3}{8}e^{22}) - \frac{1}{2}P\big)$
                $S^{12}_{i,*}$ = $(1-\alpha^{-1})S^{12}_{i,*} + \alpha^{-1}M_i^{-1}\big(P_D * \frac{1}{4}e^{12}\big)$
                $S^{22}_{i,*}$ = $(1-\alpha^{-1})S^{22}_{i,*} + \alpha^{-1}M_i^{-1}\big(P_D * (\frac{5}{8}e^{22} + \frac{3}{8}e^{11}) - \frac{1}{2}P\big)$
        }
}
```

Listing 1. Implementation of the mEVP iteration equation (3). Stress and strain tensor components $S^{11}, S^{12}, S^{22}, E^{11}, E^{12}, E^{22} \in \mathbb{R}^{N\times n_S}$ are stored as matrices where N is the number of elements and n_S the number of local DOFs in the stress space. Ice height and concentration are denoted as $H, A \in \mathbb{R}^{N\times n_A}$, where n_A is the number of local DOFs in the advection space. By $H_{i,*} \in \mathbb{R}^{n_A}$ (and similar for the stress and the strain) we denote the local row vector of the DOFs belonging to element i. The matrices $\mathrm{PSI}\langle n_A\rangle \in \mathbb{R}^{n_A\times n_G}$ are given at compile time and they evaluate the dG functions in the Gauss points with n_G being the number of Gauss points. The scalars $P^\star, \Delta_{\text{min}} \in \mathbb{R}$ are physical parameters and constant for the duration of the simulation. The matrices $M_i^{-1} \in \mathbb{R}^{n_S\times n_G}$ are pre-assembled and stored for each element. They represent the local inverse mass matrix scaled with the weights coming from the transformation of the mesh elements and multiplied with the matrix $\mathrm{PSI}\langle n_S\rangle \in \mathbb{R}^{n_S\times n_G}$. By "$*$" we denote the element-wise Hadamard product of matrices.

optimization	time [s]	speedup
CUDA baseline	0.366	1.0
CUDA shared memory	0.371	0.99
CUDA column-major	0.419	0.87
CUDA on-the-fly map	0.323	1.13
AdaptiveCPP baseline	0.466	1.0
AdaptiveCPP shared memory	0.531	0.88
AdaptiveCPP on-the-fly map	0.375	1.24
Kokkos baseline	0.522	1.0
Kokkos shared memory	0.551	0.95
Kokkos on-the-fly map	0.386	1.35

Table 2. Total time spend on the stress computation over 30 time-steps for the different implementations on an A100. Each modification is tested independently and speedup is relative to the respective baseline.

original C++ CPU code, the constant matrices are defined as static class members with explicit template specialization to enable selection of the proper matrix for the specified dG-degree at compile time. Since static member variables are not supported in CUDA, we instead declare separate variables and utilize **if** constexpr to achieve the same flexibility:

```
__constant__ constexpr T PSI_1_1[1] = {1.0};
template<int n, int nG> __device__ auto PSI(){
    if constexpr (n == 1 && nG == 1) {
        return Map<const Matrix<T,1,1>>(PSI_1_1);
    }
}
```

Another important modification for the use of Eigen on the GPU is the use of 32-bit integers as index type, since the default 64-bit integers are only emulated on the GPU.

We tried a number of optimizations to speed up the Eigen CUDA code, the results of which are shown in Table 2. The bottleneck on the GPU is often memory access. One remedy is the manual use of the L1-cache, called *shared memory* in CUDA. Shared between all threads in a thread block, it can significantly speed up reads of data that is needed multiple times and by multiple threads or when scattered memory reads/writes are necessary. In Listing 1, the only data that are used multiple times and by multiple threads are the PSI matrices. Only minor changes to the code are needed to load the PSI matrices into shared memory before use. However, we see no benefit from this change, cf. Table 2, since constant cache is just as fast for the compile time matrices. Shared memory would therefore only be worthwhile if we expect to run out of constant memory. However, the size of the compile time matrices depends only on the local degrees of freedom of the discretization. If we consider all of the implemented discretization orders together, roughly 26 KB of memory are needed. This is still less than half of the 64 KB constant memory available (NVIDIA, 2023a).

Another potential avenue to accelerate memory accesses is to carefully prepare the layout of the data. For the C++ CPU code, variables such as S^{11} are stored in row-major order, meaning that coefficients belonging to the same cell are contiguous in memory. This locality is beneficial both for effective cache usage and for vectorized memory accesses. On the GPU, the most efficient way to access global memory is through *coalesced* reads whereby neighboring threads access neighboring addresses. Since each thread processes one cell, this can be achieved by storing variables in column-major order. Nonetheless, as we can see in Table 2, the switch to column-major storage order leads to a measurable slowdown. A profiler revealed that the use of a column-major layout does improve the memory access patterns and the number of excessive sectors loaded from global memory decrease from $59\ \%$ for the row-major version to just $2\ \%$. However, this difference is rendered ineffective by the cache. In particular, data that is seemingly loaded without need in the row-major version due to the strided access is, in fact, required by subsequent computations when it can be read from the cache. Furthermore, the column-major version performs more instructions for index computations, leading to the overall slowdown.

A third option to reduce global memory accesses is to trade off reads with more computations. This is beneficial when the code is memory-bound, as is often the case on the GPU, especially with classical linear algebra (Dublish et al., 2017). In our code, the I/O can be reduced by re-computing the inverse parametric map M^{-1}, which depends only on the geometry of the mesh and compile time constants. In particular, each matrix has a size of $n_S \times n_G$ while each mesh cell's geometry is fully described by 4 vertices with 2 values each, which are furthermore shared with neighboring cells. So, disregarding constants, even for a small dG-degree such as $n_S = 3$, fewer reads are required if we compute the matrices on-the-fly, see also Richter et al. (2023, Sect. 5.3.3). Upon closer inspection, we also find that when stored in column-major order, vertex reads are coalesced while reads to M^{-1} are not, due to the fact that M^{-1} is implemented as an array of matrices. Since the Eigen matrix type only deals with two dimensions, adjusting the storage order of M^{-1} to allow for coalesced accesses would be difficult. We find that the on-the-fly map computation indeed delivers a speedup of $13\ \%$ over the CUDA baseline on an NVIDIA A100 GPU.

The above optimizations illustrate that GPU performance remains hard to predict and that for low level GPU programming, proper profiling is essential to identify bottlenecks and to develop efficient code. This applies not only to optimizations that address well-known bottlenecks, as above, but also to work on inconspicuous details in the code such as the order of expressions. An illustrative example of this is, again, found in the treatment of precomputed M^{-1}; this is relevant in particular for higher order discretizations that will be examined in greater detail in Sect. 4.3. Accessing the matrix by reference or by making an explicit copy has no impact on the performance for smaller matrix sizes like 3×4. However, when M^{-1} has size 8×9, i.e. in a second order discretization, the copy results in a kernel that is $43\ \%$ faster overall. Curiously, the slowdown with the access by reference is largely not caused by a memory bandwidth bottleneck. Instead, the massive number of unique memory accesses overwhelms the instruction queue that is responsible for executing cached memory accesses. The copy of M^{-1} alleviates this by encouraging the use of more registers to store the coefficients, thereby avoiding memory accesses.

3.2 OpenACC and OpenMP

A simple approach for moving computations to the GPU is to use a directive based programming model like OpenACC or OpenMP. In this case only small or no changes to the code are required. Targeting C, C++ and Fortran, both OpenACC or OpenMP define directives to annotate loops. These instruct the compiler to offload the computations onto the GPU.

OpenACC or OpenMP differ in how the parallel execution is described. OpenMP is *prescriptive*, meaning that the programmer has to detail how a loop should be parallelized. On the other hand, OpenACC provides a simpler *descriptive* directive that leaves more decisions to the compiler. See (Usha et al., 2020) for more details on the differences between both approaches. In practice, OpenACC tends to give better performance (Usha et al., 2020; Đukić and Mišić, 2023). However, it has more limited compiler support. Except for basic support in GCC, OpenACC can only be used with experimental and commercial compilers that primary target NVIDIA hardware. Therefore, efforts exist to automatically translate OpenACC to OpenMP to access the larger ecosystem of OpenMP (Denny et al., 2018; Servat et al., 2022).

To accelerate our code, we tried three different compilers: GCC-12.2 and NVIDIA HPC-23.5 with support for both OpenMP and OpenACC, and Clang-16.0 which currently only supports OpenMP. However, we found that all three compilers fail for our code. The NVIDIA compiler refused to compile Eigen code, while GCC and Clang either crashed during compilation or produced a broken program that would crash once executed. Runtime crashes can be attributed to incorrect memory transfers, for which only Clang provided some diagnostics in the form of compile-time warnings. In particular, objects which are not trivially copy-able, such as Eigen matrices with at least one dynamic dimension, are not captured properly. While directives are provided to manually specify the needed buffers, this is cumbersome to do for the complicated template-based Eigen types in our code. Furthermore, it voids the main advantage of the directive based approach, namely its simplicity. Use of OpenMP and OpenACC was therefore not pursued further.

3.3 SYCL

SYCL (The Khronos Group, 2023a) is an open standard for heterogeneous computing developed by the Khronos group. The standard proposes a high-level API extending C++17 that allows the same code to run on various devices such as CPUs, GPUs and FPGAs. There are currently two major implementations of the SYCL standard, both of which are open source and build on LLVM. Development of AdaptiveCPP (Alpay and Heuveline, 2023), previously known as hipSYCL and OpenSYCL, is lead by Heidelberg University. While various backends are available, the focus is on NVIDIA and AMD GPUs. The other major implementation of SYCL is Data Parallel C++ (DPC++), which developed by Intel. DPC++ primarily targets Intel CPUs, GPUs and FPGAs.

SYCL builds on top of standard C++ to minimize the effort of adapting existing code. However, the SYCL standard forbids recursion and function pointers in kernel code (The Khronos Group, 2023b), both of which are used in Eigen's expression templates. DPC++ does not allow one to compile the neXtSIM-DG code because of these limitations, although the function calls should be entirely inlined in the compiled code. AdaptiveCPP requires more effort for setup but the tool chain compiles Eigen. We therefore limit our investigations to AdaptiveCPP in the following.

SYCL automates device memory management and movement of data between host and device, but memory requirements of a kernel need to be declared explicitly. To this end, a *buffer* needs to be defined, pointing to already allocated memory on the host. Then, a *command group* is created which collects all information needed to run a task in parallel. Inside the command group, *accessors* allow us to explicitly describe which buffers need to be accessed and how, i.e. read or write. Once pushed into a *queue*, the SYCL runtime uses these memory requirements as well as optional dependencies on other command groups to select the best suited memory region to perform needed memory transfers and to schedule the execution. Inside the command group, we can declare a parallel for-loop and construct Eigen maps analogous to CUDA with pointers provided by the accessors.

We can investigate the same optimizations as with the CUDA code. While shared memory did not improve performance for native CUDA, it is still of interest to see how it affects the SYCL implementation, since memory management works differently there. To access *local memory* in SYCL, which is the name used for CUDA's shared memory, we have to declare a *local_accessor* in the command buffer. In addition, local memory only makes sense in the context of thread blocks, so we need to use a more complicated for-loop which makes thread blocks explicit. Unfortunately, such a construct is known to perform far worse on the CPU than a simple loop and work on reducing this gap is an active area of research (Meyer et al., 2023). Therefore, if the code is to be efficient both on CPU and GPU, local memory should be introduced only in code paths specialized for the GPU. For the code snippet under study, this additional effort was not considered worthwhile. Returning to Table 2 we see using shared memory makes the kernel moderately slower. On the other hand, computing M^{-1} on-the-fly leads to a more substantial relative speedup over the AdaptiveCPP baseline than the same optimization in native CUDA.

3.4 Kokkos

Kokkos (Trott et al., 2022) is another programming model to enable heterogeneous computing in modern C++, currently with support for CPU as well as NVIDIA and AMD GPUs. Kokkos is developed as part of the Exascale Computing Project by the US Department of Energy. The main difference to SYCL is that Kokkos is a library while SYCL requires compiler integration. The library-based approach greatly simplifies deployment of projects using Kokkos but potentially limits available optimizations and features.

Kokkos consists of macros and wrappers that provide a unified API for the different backends with the final code being processed by the chosen compiler. Therefore, we can once again start from the CPU code shown in Listing 1, knowing that it works in native CUDA. The primary mechanism to manage memory in Kokkos are *Views*, which are basically a shared pointer to a multi-dimensional array. Typically, both a device view and a mirrored host view are created to facilitate data transfers. For our use case, it is possible to create a view on already allocated memory with the *unmanaged* trait. However, unmanaged views do not play well together with the mirrored views concept in backend agnostic code, leading to unnecessary copies during the execution on the CPU. Since in general the device view needs its own buffer, copies between the mirrored views will be performed regardless of whether they already reside in the same memory space. These extra copies can be avoided adding a special case for just the view creation on CPU and are therefore not a major problem for portability. Once properly setup, data is accessible in the kernel through the device view and we can use the underlying pointer to create an Eigen map in the same manner as in CUDA.

Possible code optimizations in Kokkos are similar to those available in SYCL. CUDA's shared memory, called *scratch memory* in Kokkos, can be accessed by specifying a *TeamPolicy* with a thread block size instead of using a simple parallel for-loop. Here a nuisance of the library becomes apparent as the total scratch memory needed for a particular kernel has to be set manually. Furthermore, parallelism described with explicit thread blocks has the same downside as in SYCL, namely that it leads to strongly degraded CPU performance. In our tests we find that usage of scratch memory introduces a small overhead in Kokkos, see Table 2. On-the-fly map computation is again beneficial and it results in a large speedup of 35 $\%$.

3.5 PyTorch

PyTorch (Paszke et al., 2019) is one of the most popular libraries for machine learning (Aoun et al., 2022). It consists of a simple-to-use Python frontend and a high-performance C++ backend that has a dedicated compiler to optimize code execution and maps execution for different hardware such as CPUs, GPUs, and TPUs. Full support is available for CPUs, NVIDIA GPUs and AMD GPUs.

To make effective use of PyTorch and the optimizations it implements, computations have to be reformulated in terms of large tensors. For this, we remove the main loop in Line 4 of Listing 1 and treat the element dimension N as the batch dimension of variable size. The matrix-vector products then become matrix-matrix products and element-wise operations remain unchanged. Some care is necessary to perform the products with the per-element inverse maps, e.g. Line 16. Since we have a third dimension in M^{-1}, this is not a standard matrix-matrix product. However, we can map this operation to a batched matrix-matrix product (bmm) by appending a dimension of size 1 to the second argument and removing it again afterward (squeezing in PyTorch terminology). Alternatively we can formulate this computation as an element-wise product by adding a dimension corresponding to n_S to the second argument, followed by a sum over that dimension. The latter operation turns out to be 4 to 5 times faster across different backends, indicating that PyTorch is not tuned for our use case where the matrices are much smaller than those is common in machine learning workloads.

To integrate the PyTorch code into our C++ simulation, we have multiple options. With minor syntactic changes compared to the Python version, we can implement the computations directly with PyTorch's C++ API. However, this is inefficient since each operation is executed as a separate kernel with no kernel-fusion taking place, resulting in many reads and writes of the same data. A second option is to define the computation as a PyTorch model in Python. This model can be exported as *Torch-Script* and loaded in C++. Part of the C++ runtime is a just-in-time compiler which attempts to optimize the model execution on repeated use. However, more recent efforts to accelerate PyTorch models have been focused on *TorchDynamo*, a compiler first released with PyTorch 2.0. While the front-end of TorchDynamo is written in Python, various backends are available, some of which can be used without the Python runtime. Most promising among those we tested is the built-in *TorchInductor* which leverages the compiler Triton (Tillet et al., 2019) to produce highly optimized fused-matrix multiplications (PyTorch-devs, 2023). In particular, PyTorch 2.2 introduces AOTInductor, a version of TorchInductor that exports the entire model as a shared library with a single wrapper function to call directly from C++. Another way to deploy the PyTorch model in C++ is through

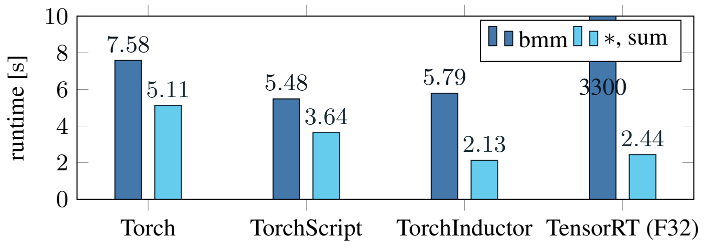


Figure 1. Total time spend on the stress computation over 30 time-steps for the different PyTorch variants on an A100. The products with M^{-1} are implemented either as batched matrix-matrix product (bmm) or element-wise product and sum ($*$,sum). TensorRT uses single precision (F32) since double is not supported.

the extension Torch-TensorRT (Torch-TensorRT-devs, 2024), which uses NVIDIA's inference engine TensorRT (NVIDIA, 2023b) as backend. One limitation of TensorRT is that it does not support double precision.

We compare the four proposed variants to integrate the PyTorch model into C++ in Fig. 1. Although they use the same tensor primitives, the native C++ interface is considerably slower than TorchScript. The new compiler, TorchInductor, with its Triton-optimized kernels, is significantly faster than the alternatives when using the element-wise product. When implemented with bmm, the compiler fails to optimize the operation due to a lack of GPU memory. The fact that a batch size of 2.6×10^5 is already too large, although the operands require less than $5\ \mathrm{MB}$ of memory, points to it being an edge case not properly considered by the optimizer. TensorRT is slower than TorchInductor for our use case, even while running in lower precision and has even more trouble with the bmm operation. Optimization of the model takes over a minute and the inference is orders of magnitude slower than the sum-based version.

3.6 Development and deployment effort

The development of dedicated CUDA code is time-consuming and error-prone. One purpose of the alternatives we considered in this work is to reduce this high development effort. Furthermore, most of them support a unified code for a variety of compute hardware. We therefore do a qualitative comparison of the different approaches, considering ease of development but also deployment of the finished code on a target system.

With their modern C++ interface, both Kokkos and SYCL make it easier to write correct code compared to CUDA. Simplified resource management and stricter types reduce the risk of memory related bugs and make more errors visible at compile time. The simple parallelism constructs also hide GPU specific scheduling based on blocks and grids from a developer. A further advantage of SYCL is that explicit annotations of device functions are unnecessary. SYCL's memory model fully automates transfers between host and device, eliminating another source of errors. It should be noted, however, that the more advanced C++ features used by Kokkos and SYCL can make the frameworks less approachable for non-C++ experts than the C-like interface of CUDA. PyTorch follows a completely different programming paradigm from the other options. While, when coming form a system programming language background, PyTorch takes time to get used to, development in PyTorch is overall much simpler. There is no memory management or explicit parallelism to take care of and rapid prototyping in Python is possible. A potential downside can be that some computations are hard to express in terms of tensor operations, in which case a low-level, manual implementation is still needed. Another downside for our particular case was that code had to be completely rewritten in PyTorch, while for the other options the C++ CPU code could be largely reused.

For running the code on a target system, pure CUDA is easiest. Usually pre-installed on clusters, no additional setup is required. Furthermore, CUDA (or the AMD equivalent ROCm) is a prerequisite for the other frameworks to use the GPU, so if a manual installation is needed, this effort is unavoidable for every framework. The Kokkos library can be easily integrated into a project's CMake based build system and then works out-of-the-box. In combination with automatic fetching, e.g. via git submodule, the library setup becomes transparent. SYCL requires a specialized toolchain. For AdaptiveCPP, this means that its compiler wrapper has to be first build from source. The manual configuration that is needed for AdaptiveCPP to find the proper compilers is cumbersome and, in some cases, necessitate building a suitable version of LLVM first. For PyTorch, prebuild C++ libraries are available for all supported platforms. To use TorchInductor, the Python package is needed to generate the code on the target system, but it is easily acquired through a package manager.

4 Numerical experiments

To analyze the performance of our implementations, we use the established VP benchmark of a moving cyclone over a sea ice region (Mehlmann et al., 2021). We simulate a duration of 1 h, which requires 30 advection steps and $3,000$ stress updates. For the discretization we choose first order continuous Galerkin (cG) elements and discontinuous Galerkin (dG) elements with 3 degrees of freedom for the advection. The original C++ CPU version of the code has already been validated on this benchmark, see (Richter et al., 2023). We therefore compare to the CPU version to determine the accuracy of the computed results.

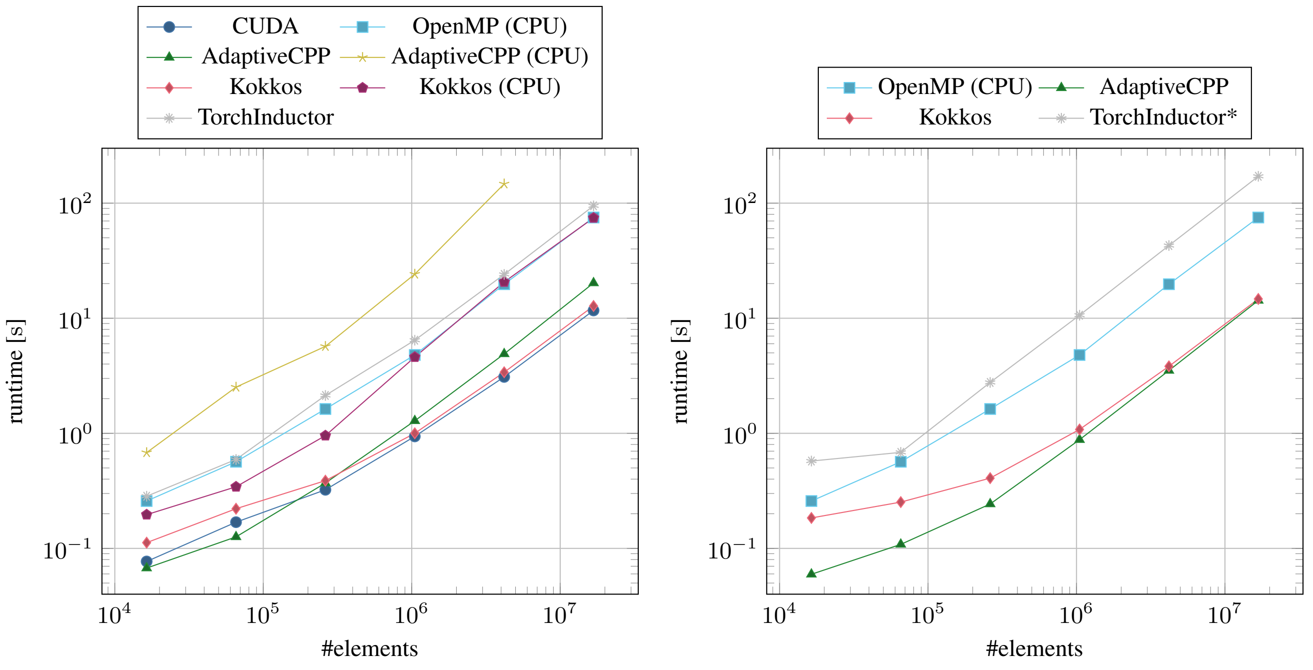
In the GPU implementations, significant time is required to transfer memory between host and device. Nonetheless, we only consider the kernel execution times in the following since the final objective of our work is a full GPU implementation of the dynamical core. While transfers are still necessary for coupling with other models, the major effort of simulating the sea-ice dynamics is in the mEVP iteration with the many sub-iterations considered in this work. This will amortize the costs of the memory transfer. To ensure accurate timings, synchronization barriers are inserted as needed before and after the kernel invocation. In SYCL, memory transfers are implicit, so we rely on the built-in profiling instead to obtain the timings. Details on the software and hardware used in the experiments are listed in Table 3.

4.1 Performance scaling

Of particular importance for coupled climate simulations is the scaling of the performance as a function of grid resolution. With a fixed domain size of 512 km, we reduce the cell size from 4 km to 0.25 km. This corresponds to an increase in the number of elements from 1.6×10^4 to 1.7×10^7 (i.e. one has a quadratic scaling of the element number in the resolution). In Fig. 2 we compare the best implementation for each approach as a function of elements for two different data center GPUs.

	System 1	System 2
CPU	2× AMD EPYC Rome 7402, 2 × 24 cores @ 2.8 GHz	AMD EPYC 7A53, 64 cores @ 2.75 GHz
GPU	NVIDIA A100, 40 GB HBM2e	AMD Instinct MI250X, 128GB HBM2e
OpenMP (CPU) compiler	GCC-12.3	-
GPU software stack	CUDA 12.2	ROCm 5.6.1
Kokkos	4.1.0	4.1.0
AdaptiveCPP	23.10.0 based on Clang-17.04	23.10.0 based on AMD Clang-16.0
PyTorch	2.3 Nightly (24-November-2023)	2.2

Table 3. The two systems on which performance measurements where conducted.



(a) System 1: NVIDIA A100 (b) System 2: AMD Instinct MI250X

Figure 2. Timings of the stress update using the best performing version for each framework. The size of the mesh cells size is scaled from 4 km to 0.25 km while keeping the domain size constant to increase the number of elements. On the AMD Instinct MI250X, only one of the two graphics compute dies (GCDs) is used. The values for OpenMP (CPU) are the same in both plots and where measured on System 1.

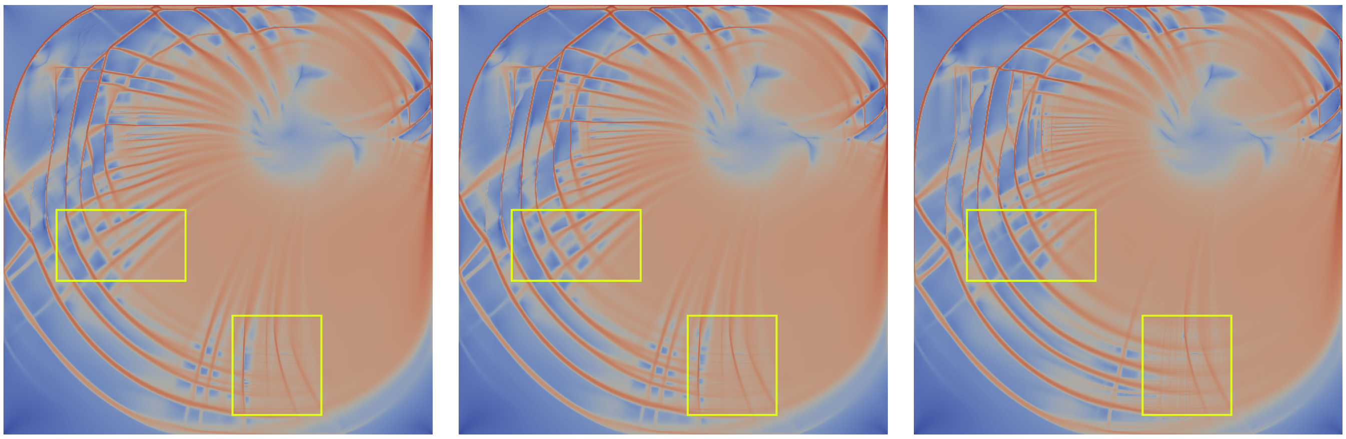
In the GPU versions of CUDA, Kokkos and AdaptiveCPP, we compute the inverse maps on-the-fly. In case of Kokkos and AdaptiveCPP, the simple for-loop is used to run the update in parallel, which also makes an execution on the CPU efficient, albeit with precomputed maps. For PyTorch, we take the implementation generated by TorchInductor with the element-wise product and sum.

Running on a NVIDIA A100 GPU (Fig. 2a), our CUDA implementation delivers a significant speedup over the OpenMP CPU reference implementation. For the smallest problem with 1.6×10^4 elements, CUDA is 3 times faster, scaling up to 6.4 for 1.7×10^7 elements. Kokkos achieves asymptotically the same performance as CUDA on the GPU and as OpenMP on the CPU. This is to be expected since the very same compilers (NVCC and GCC) are used by Kokkos and only memory buffers and kernel dispatch are abstracted. On small problems, Kokkos overhead makes it $50\ \%$ slower than CUDA but, surprisingly, the CPU version is slightly faster than raw OpenMP for the same number of elements. AdaptiveCPP scales worse than the other GPU accelerated codes, being $70\ \%$ slower than CUDA for the largest problem size we tested. However, it still provides a significant improvement over the CPU OpenMP version. The good performance of AdaptiveCPP for small problem sizes is likely an artifact from the different time measuring method. On the CPU, we were not able to run a meaningful experiment with AdaptiveCPP. Best performance was achieved with a restriction to just 24 threads, indicating that the available CPUs are not utilized properly. The documentation states that performance of the CPU backend should be similar to raw OpenMP and that a significant deviation is likely caused by an improperly configured toolchain. However, we were unable to obtain stable results on three different systems, illustrating the substantially greater difficulty in using the framework compared to Kokkos. TorchInductor on GPU is slower than the OpenMP CPU code for every problem size tested and TorchInductor's CPU code (not shown) is an order of magnitude slower than the GPU version.

To test the portability of the heterogeneous compute frameworks we also ran the experiments on System 2, equipped with an AMD MI250X GPU. The results are shown in Fig. 2b. Kokkos, AdaptiveCPP and PyTorch work without modifications but only utilize half of the MI250X, since it is a dual graphics compute die (GCD) design. AdaptiveCPP performs better on the AMD GPU, while Kokkos is somewhat slower than on the NVIDIA A100. Both thereby achieve a similar runtime, roughly $25\ \%$ higher than that of CUDA running on the A100. For PyTorch, the MI250X is $50\ \%$ slower than the A100 in our experiments. On paper, a single graphics compute die of the MI250X has the same memory bandwidth as the A100 and a F64 peak performance more than twice as high (A100 9.7 TFLOPS, MI250X 23.9 TFLOPS). These results indicate that the AMD ecosystem is still less mature. However, with the performance currently achieved it is still a worthwhile target platform.

4.2 Mixed Precision

One avenue to further speed up the simulation is to perform computations with lower precision float types. Switching from double precision (F64) to single precision (F32) halves the memory required and doubles the theoretical peak performance achievable on the A100 (F64 9.7 TFLOPS, F32 19.5 TFLOPS). Modern GPUs support even lower precision types that promise further speed-ups. Of particular interest is tensor float (TF32), a format used in tensor cores, which are specialized matrix multiplication hardware found on NVIDIA GPUs and originally introduced for machine learning workloads. TF32 uses the same exponent as F32, so that the range of representable numbers is the same as for F32, combined with a half precision



(a) Reference F64 (b) F32 (c) F32 / TF32

Figure 3. Shear deformation ($\log_{10}$) for the benchmark after 48 h where the stress update is computed with different floating point types.

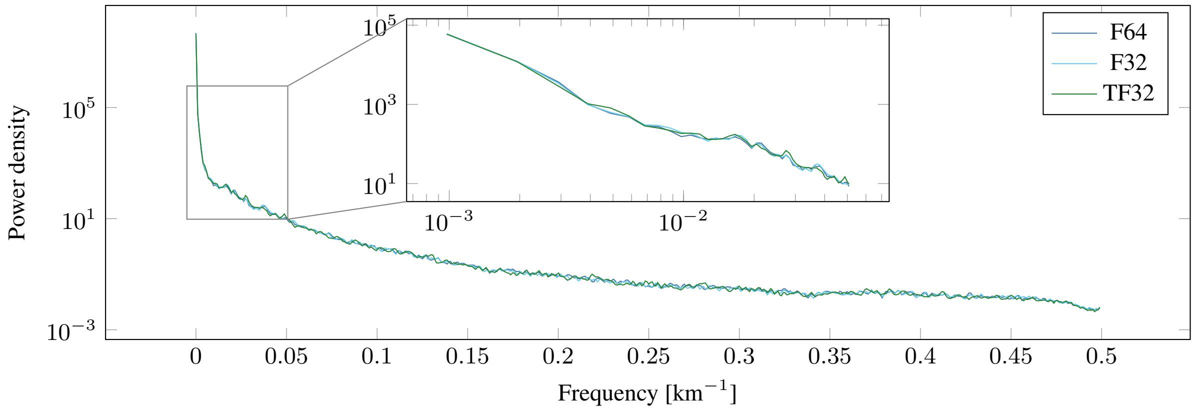
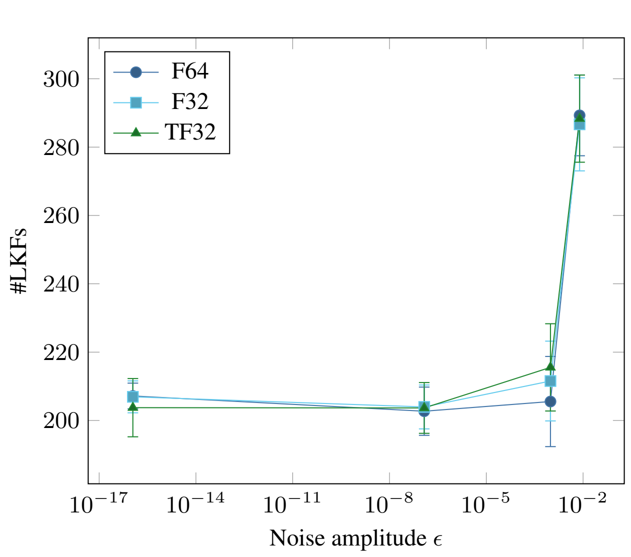


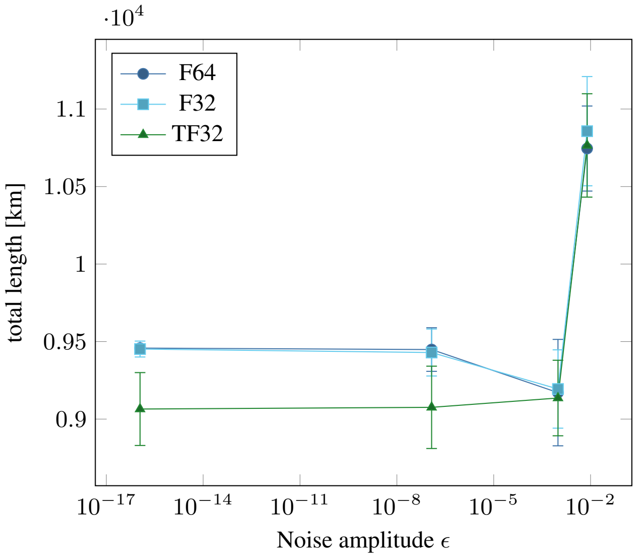
Figure 4. Radially averaged power spectral density (RAPSD) of the shear deformation ($\log_{10}$).

mantissa with just 10 bits that yields a significantly higher peak performance for matrix multiplications (156 TFLOPS) compared to F32.

First we evaluate the impact of lower precision floats on the quality of the results. To this end, we perform the stress update in F32 while the rest of the simulation still runs in F64. To make use of the tensor cores, we run the PyTorch version, where TF32 can be enabled with a simple switch. Results for the benchmark after 48 h simulation time are shown in Fig. 3. Visually, F64 and F32 are indistinguishable, while some movement of larger cracks and additional fine features can be seen with TF32. Nevertheless, the overall distribution of features in the shear deformation remains very similar and there is good agreement in the frequency spectrum of the features, as can be seen in Fig. 4.



(a) Number of LKFs



(b) Length of LKFs

Figure 5. Distribution of linear kinematic features (LKFs) after 48 h when the initial state of ice height H and ice concentration A are perpetuated with uniform noise sampled from $[-\epsilon, \epsilon)$. The noise magnitude is chosen as the interval machine precision for different floating point types, i.e. 2^{-53} (F64), 2^{-23} (F32), 2^{-10} (F16/TF32), 2^{-7} (BF16). Points show the mean of 24 runs, while the whiskers indicate the standard deviation.

To quantify the stability of the physical model itself and whether computations with lower accuracy amplify instabilities, we look at the distribution of linear kinematic features (LKFs) (Kwok, 2001) using a script provided by Hutter (Hutter et al., 2019). We investigate the influence of randomly perturbed initial conditions on the formation of LKFs in the benchmark simulation. For F64 and F32, both mean and standard deviation are largely the same and perturbations up to F32 machine precision around 10^{-7} appear to have no impact on the distribution. This indicates that, F32 can be used without impacting the results. For TF32, there is a statistically significant deviation in the length of LKFs even for very small perturbations. However, the uncertainty in the sea-ice tracers from data assimilation is at least of the order 10^{-2} (Liu et al., 2019; Xie et al., 2017). In our experiment, the variation of the initial conditions has a larger effect on the LKFs than the floating point type used at this point, so TF32 could still be useful in practice.

The performance gains from switching the stress update to F32 or TF32 are plotted in Fig. 6. For CUDA, switching from F64 to F32 gives a speedup of 80 %, which is in line with the expected speedup for the A100. We expect to see similar gains from using F32 in Kokkos and AdaptiveCPP. For TorchInductor, the relative speedup with F32 over the F64 version is much higher at 315 %. As machine learning tasks rarely use double precision, the optimizer is likely tuned much more for the single precision case. The variant with tensor cores enabled, TorchInductor (TF32), has speedup of 412 % over F64 which is not quite as large as the theoretical peak performance would suggest. This is because not all operations can make use of the tensor

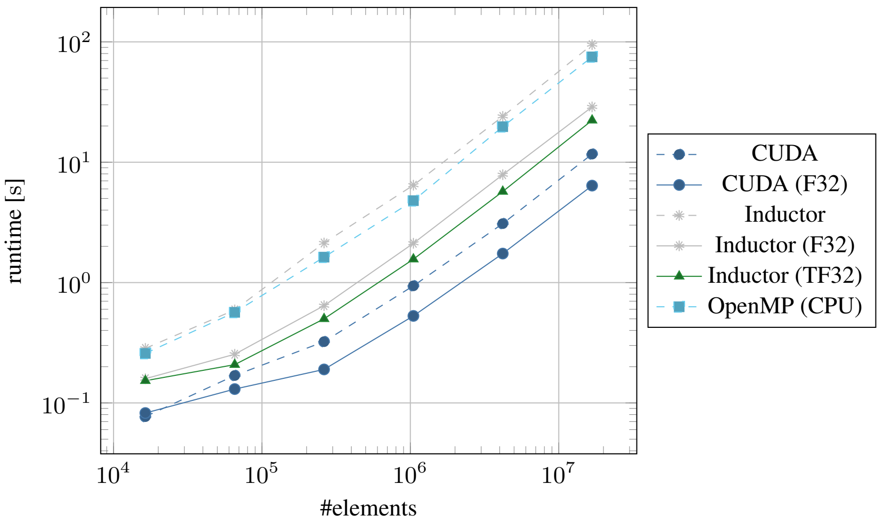


Figure 6. Timings of the stress update for lower precision floating point types on the NVIDIA A100. The dashed lines are references in F64, taken from Fig. 2.

cG \ dG	$n_A = 1$	$n_A = 3$	$n_A = 6$
1 ($n_S = 3, n_G = 4$)	1.30	1.31	1.31
2 ($n_S = 8, n_G = 9$)	1.29	1.32	1.38

Table 4. Estimated arithmetic intensity [FLOP byte^{-1}] of kernels with different discretization orders for F32. The roofline for the A100 is 9.73 FLOP byte^{-1} (or 12.5 FLOP byte^{-1} with boost clock).

cores and where it is possible, the matrices involved are too small to take full advantage of the tensor cores. In absolute terms, TorchInductor (TF32) still takes almost twice as long as CUDA (F64), even with access to tensor cores.

4.3 Higher order scaling

Our finite-element code makes it easily possible to change the local number of degrees of freedom. So far we have focused on just one discretization, cG1-dG3, i.e. first order continuous Galerkin (cG) elements for velocity and 3 degrees of freedom discontinuous Galerkin (dG) for the advection. A higher order discretization significantly increases the compute load which could make it more efficient on the GPU when compared to the CPU, since the GPU is limited by memory bandwidth in our code. On the A100, compute throughput reported by the profiler for cG1-dG3 varies from 28 % to 50 %, depending on whether the precomputed maps are used, while the memory throughput is at 70 % to 80 %.

A simple roofline model of the A100 GPU is shown in Fig. 7. For our code, the relevant bottleneck is clearly the *arithmetic intensity*, i.e. the ratio of compute operations per byte accessed in global memory. The values reported by the profiler for our

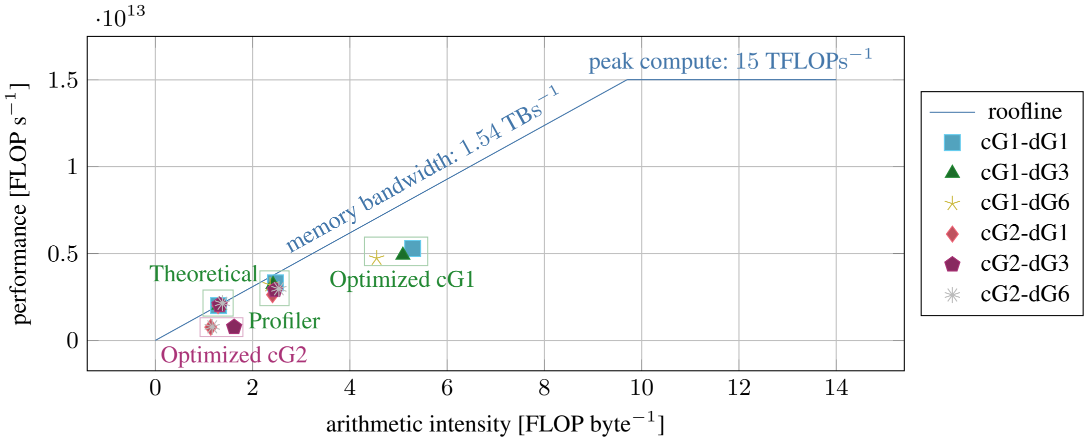


Figure 7. Roofline model of the F32 performance of the A100 with different variants of our CUDA kernel. Theoretical values are computed manually, while the other values are taken from Nsight Compute, the NVIDIA GPU profiler. The peak compute value given here assumes base clock speed to be consistent with the profiler results. Optimized kernels perform the on-the-fly map computation which is not viable for higher cG order.

baseline kernels with precomputed maps are all close to the bandwidth limit, which underscores our observations from Sect. 3, i.e. that the only effective optimization is a reduction of the needed data.

In contrast, the optimized kernels with on-the-fly parametric map computation present a different picture where neither memory bandwidth nor compute are a bottleneck. Instead, other limitations become apparent which can be attributed to the complexity of our kernel. One aspect that is exacerbated by the on-the-fly map computation, is the high number of registers that are needed by each thread. More temporary variables are needed and though, for the tested cG1 kernels, these variables still fit in the registers, the higher register usage leads to a lower *occupancy*, i.e. the ratio of active warps that fit on each execution unit at the same time and the theoretical maximum supported by the hardware (for cG1-dG3: $57\ \%$ unoptimized, $40\ \%$ optimized). As a result, the scheduler is unable to keep the different pipelines busy and more cycles are wasted just waiting.

While the gains in arithmetic intensity from the optimization are clearly worth it for cG1, cf. Table 2, the same is not true for cG2. With second order cG elements, the increased workload per thread, caused by the on-the-fly map computation, becomes a major problem. The computation involves a matrix inverse that becomes both too expensive and memory intensive for higher orders. For first order cG, this matrix has size 3×3 and we can use a closed-form formula to compute the inverse that results in efficient code. For second order cG elements, the matrix has size 8×8 and have to we rely on a generic inverse implementation that becomes a major bottleneck. As we can see in Fig. 7, the arithmetic intensity of the “optimized” cG2 kernels is, in fact, lower than that of the kernels with precomputed maps because of a large number of temporary variables that end up getting stored in global memory, since the available per-thread register space is exhausted (in technical terms data is *spilled* to *local memory*). Subsequently, the unoptimized kernel is roughly 8 times faster. In the following, we therefore limit our analysis to the version with precomputed maps.

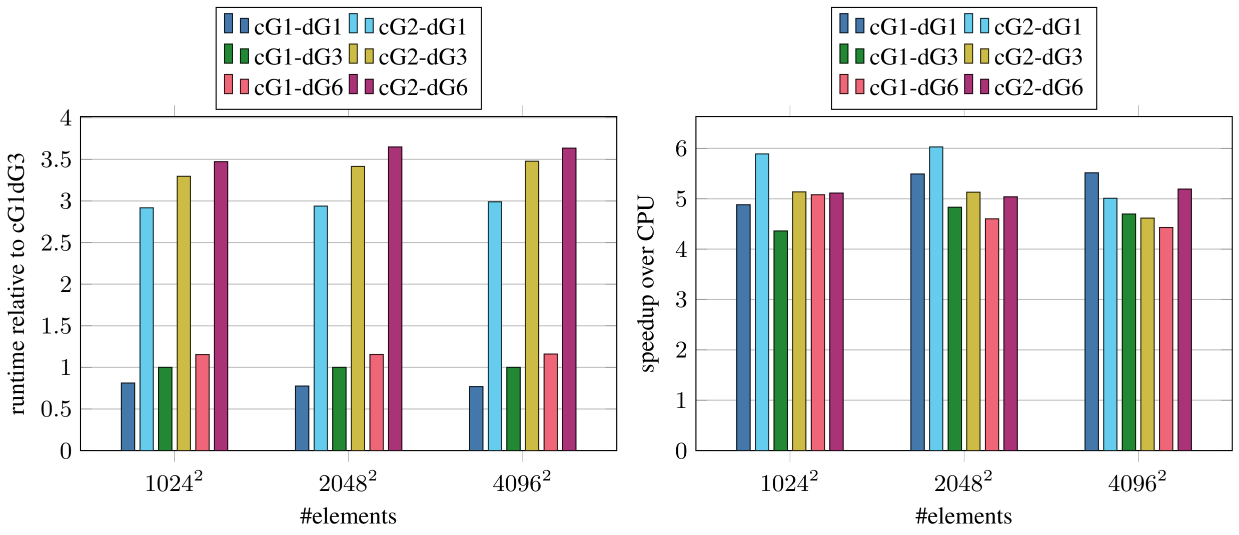


Figure 8. Performance of the CUDA implementation with precomputed map M^{-1} for different discretization orders on the A100. The three largest problem sizes from the scaling test are shown.

To analyze the possible throughput independent of (sub-optimal) code generation, we also compute a theoretical bound based on the algorithmic description in Listing 1. For memory accesses we only count the data that is unique for each kernel invocation, i.e. $S_{i,*}, E_{i,*}, H_{i,*}, A_{i,*}$ and M_i^{-1}, giving us a total of $9n_S + 2n_A + n_S n_G$ float read and writes. For compute
435 operations we only count float operations and assume F32, since in that case, all operations needed are native instructions with well documented throughput (NVIDIA, 2023a). Additions are not counted since they can all be executed as fused-multiply-add, $\min$ and $\max$ have the same throughput as compute operations, and exp2 and sqrt^{-1} count as 4 operations each (compute capability 8.0). Therefore, the total number of operations is

$$6n_S + 3n_A + 6n_S n_G + 2n_A n_G + 28n_G,$$

440 giving us the theoretical arithmetic intensities recorded in Table 4. We can see that, in agreement with the profiler results, an increase in either order is insignificant for the arithmetic intensity. The largest difference of just 6 % is from cG2-dG1 to cG2-dG6, which should not have much of an impact on the memory access bottleneck.

To verify this scaling model we measure the runtime for different discretization orders and compare with the OpenMP CPU baseline in Fig. 8. We find that the speedup from using the GPU ranges from 4.4 to 6.0 with no clear trend for either
445 order parameter, inline with the theoretical prediction. The observed differences can mainly be attributed to the problem size impacting the behavior of the CPU version, since the relative runtime between the different GPU variants is very consistent across problem sizes.

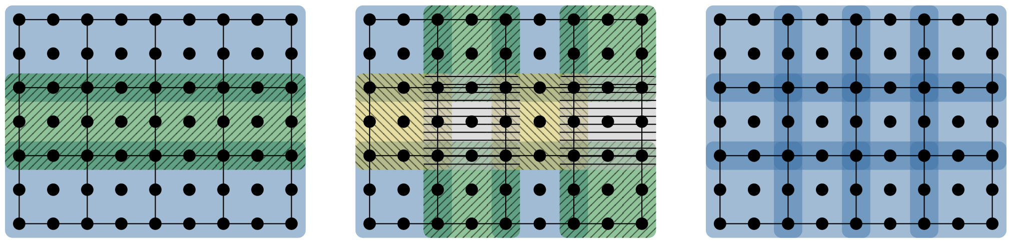


Figure 9. Parallel processing of biquadratic cG2-vectors. The mesh has 12 elements with $N_x = 4, N_y = 3$. Each block corresponds to one thread and blocks of the same color can be processed in parallel. The maximum number of parallel tasks is, from left to right: $\frac{N_y}{2}$ (row-wise), $\frac{N_x N_y}{4}$ (strided), $N_x N_y$ (atomic).

5 NeXtSIM-DG implementation

Based on the results from Sect. 4 we decide on Kokkos for the full implementation of the neXtSIM-DG sea ice dynamical core. In our evaluation Kokkos offers performance almost competitive to CUDA but with greater ease of development, multi-vendor GPU support and the potential to replace the dedicated CPU implementation as well.

The port of the complete mEVP iteration to Kokkos is, for the most part, straightforward with the experience gained from the stress implementation. Of note is the need for a different parallelization strategy for operations that involve neighboring cells of the mesh such as the divergence computation. In the OpenMP implementation, race-conditions are circumvented by performing the update in two steps, whereby every other row is processed in parallel (Richter et al., 2023). However, as illustrated in Fig. 9, this parallelization strategy along just one dimension does not provide nearly enough work to saturate a GPU. One way to further increase parallelism without introducing contention at the edges is to perform four separate steps in a strided checkerboard pattern instead. Another way, that turns out to be faster for the divergence computation, is to process every cell in parallel and to rely on atomic operations to ensure that the values are updated correctly. The use of atomics in such a way could be suboptimal when running on CPU, but it does not appear to be a problem in our experiments.

Another concern for portability, also related to cross-cell updates, is the choice of the parallel loop construct. To easily address neighbors it can be tempting to use Kokkos's 2D range policy. However, this policy splits work into tiles instead of rows, which does not play well with our underlying data layout on the CPU. The whole mEVP update becomes 3 times slower on CPU when using 2D-loops for strain and divergence computations, instead of 1D-loops with manual index. GPU performance is, in our case, largely unaffected by this choice.

While the new Kokkos code is not fully optimized yet and limited to F64, it already provides a sizable speedup over the OpenMP code. Results for the benchmark setup described in Sect. 4 are presented in Fig. 10. In contrast to the previous experiments, the timings for the full mEVP iteration include the necessary data transfers to and from the device. Running on GPU, the mEVP iteration is faster by a factor of 6, even for small problem sizes. Running on CPU, the Kokkos code performs just as well as the OpenMP code. The cost of data transfers for the GPU version is still non-negligible at a constant 10 % for

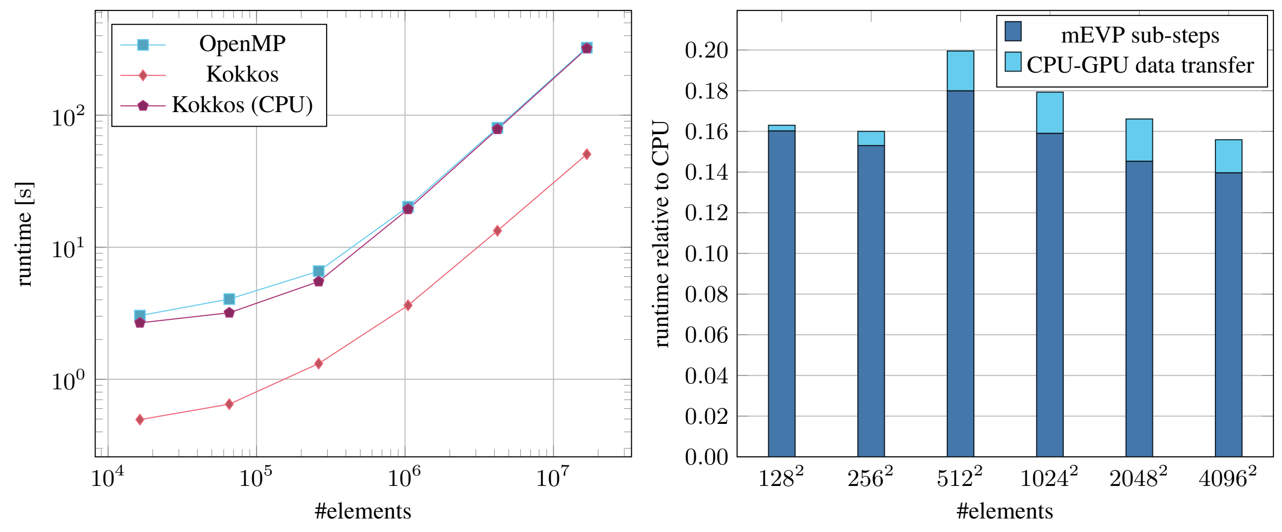


Figure 10. Timings of the full mEVP iteration on the NVIDIA A100. The values for Kokkos include the necessary data transfers between CPU and GPU.

larger meshes, even though the 100 mEVP sub-steps are performed entirely on the device. This underlines the importance of running larger parts of a simulation fully on the GPU to minimize necessary data transfers.

6 Conclusions

We implemented and evaluated different options for the GPU parallelization of the neXtSIM-DG dynamical core. According to our results, CUDA remains the most reliable option both in terms of performance and with regard to the toolchain. Thanks to the CUDA support of Eigen, we were also able to use the CPU C++ code with minimal modifications in CUDA.

Kokkos benefits in the same way from Eigen's CUDA library support, while SYCL does not need explicit support which makes it well suited for an incremental port of existing C++ code in general. The streamlined memory model and simplified parallel constructs of Kokkos and SYCL facilitate more effective development, but at some performance cost. Using dedicated GPU features such as shared memory remains an issue because it leads to code that is very inefficient on the CPU, breaking the promise of the heterogeneous computing paradigm. However, our study demonstrates that this specialization is not always needed to achieve good performance and we can therefore recommend Kokkos as an alternative to CUDA. While SYCL shares the same benefits on paper, it suffers from immature implementations and is currently too unreliable.

PyTorch currently lacks far behind the more conventional options in terms of performance and is therefore mostly worth considering for rapid prototyping. However, the optimizer heuristics are clearly far from optimal yet for our use case and the underlying compilers are developing quickly, so we expect performance improvements in the future. Furthermore, PyTorch and similar machine learning frameworks are interesting alternatives because of their ease of development and the access to automatic differentiation they provide. The latter is of great relevance for hybrid methods that combine a conventional discretization with a machine learning component, e.g. Bedrunka et al. (2021); Kochkov et al. (2021); Demeure et al. (2023); Kochkov et al. (2023).

Our investigation of mixed precision underscores previous results from the literature, i.e. that lower precision float types should be considered for GPU codes. Performing a major computation of our sea ice simulation in single precision shows no degradation in the results while almost doubling the performance. Although the application of tensor cores with their even lower precision does have a measurable impact on the results, further tests with more realistic scenarios will be needed to determine whether there is a practical impact of going below single precision.

Purely from a computational perspective, our finite element based GPU code does not favor higher order discretizations. While the speedup over CPU is considerable for all combinations tested, there are additional optimization opportunities for the first order continuous elements which makes them more efficient. A comprehensive evaluation of the trade-off between quality and speed of different discretization orders are left for future work.

For the GPU-parallelization of the entire neXtSIM-DG dynamic core we chose Kokkos. For the full mEVP iteration running entirely on GPU we obtain a speedup by a factor of 6, switching from a dual CPU node to a single A100 GPU in double precision. With some care in the implementation, the Kokkos implementation achieves the same performance on CPU as our manual OpenMP based implementation, making the latter obsolete. The code is not fully optimized yet and more work is needed to port advection and other rheologies. With components outside the dynamical core also still in active development, performance comparisons with currently used models are difficult. However, based on the results shown in this work, a move from the current $10\,\mathrm{km}$ resolution sea ice models (Ólason et al., 2021; Hutter et al., 2022) to practical kilometer-scale models seems tangible.

Code availability. The project neXtSIM-DG is under active development and hosted on GitHub (https://github.com/nextsimhub/nextsimdg). The different GPU implementations for just the stress are available in a self contained repository at https://kosinus.math.uni-magdeburg.de/Thanduriel/dynamical_core. A snapshot with all code needed to reproduce the experiments in this manuscript is available on Zenodo (Jendersie et al., 2024).

Author contributions. Robert Jendersie developed the GPU codes, conducted the experiments and wrote the bulk of the text. Christian Lessig gave substantial input on the research direction and analysis methods and worked on the final text. Thomas Richter provided the original CPU code as well as the model description and helped to improve the final text.

Competing interests. The authors declare that they have no conflict of interest.

Acknowledgements. This project is supported by Schmidt Sciences. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer JUWELS (Jülich Supercomputing Centre, 2021) at Jülich Supercomputing Centre (JSC). We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Regular Access call.

Appendix A: Introduction to GPUs

Graphics processing units are coprocessors originally designed to accelerate computer graphics. To generate images in quick succession, a massive number of operations, e.g. computing the color of each pixel on the screen, has to be performed. The computations are typically identical and it is thus a highly parallel problem. Initially, the specific algorithms used for this purpose were implemented directly in the hardware and the resulting *fixed function pipeline* offered only very limited possibilities for customization. However, to enable higher fidelity graphics, most parts of the pipeline where replaced with programmable stages over time. Attempts to use such programmable GPUs for scientific computing soon followed (Bolz et al., 2003; Krüger and Westermann, 2005). To further facilitate the application of GPUs beyond computer graphics, new programming interfaces where developed that represent GPUs as general purpose *stream processors* (Buck et al., 2004). In this paradigm, a program takes a long sequence (a stream) of data and applies *kernel functions* to each element in the sequence. Among the general purpose programming interfaces, CUDA (NVIDIA, 2023a) has been the most influential and the fundamental programming model defined by CUDA has since stayed the same.

In the following, we give an overview to GPUs as general purpose parallel processors, aimed at readers somewhat familiar with the architecture of CPUs. In the interest of brevity, we only use the nomenclature of CUDA, but the concepts and components described here also exist on GPUs of other vendors, sometimes under different names.

GPUs are designed for high throughput and can, given an embarrassingly parallel problem, deliver significant gains in performance and energy efficiency over CPUs. Naturally, this requires a trade-off, since chip resources need to be distributed differently, as Fig. A1 shows. For CPUs, much of the die space is allocated to the multi-level cache hierarchy and control circuits, e.g. speculative execution, with the goal to minimize latency as much as possible. In contrast, GPUs prioritize compute throughput with many more arithmetical logical units (ALUs) and a much wider execution pipeline. If sufficient work is available to saturate the compute pipeline, latency can, in principle, be hidden because computations in different stages of execution can keep every part of the hardware busy. For a program to take advantage of just a single GPU, it needs thousands of similar computations that can be run in parallel, as opposed to just a few tasks needed for a multi-core CPU. In practical terms, a good balance between operations is required to achieve high utilization, i.e. the compute units can only be kept busy if the data can be loaded and stored with the same speed as it is processed. The dedicated main memory on GPUs (VRAM) is therefore also optimized toward higher bandwidth whereas regular RAM for CPUs prioritizes lower latency.

To access data on the GPU it needs to be first moved to *global memory*, a logical space residing in the VRAM. Since transfers between the host memory (system RAM), and device memory (VRAM) are slow, the work offloaded to GPUs needs to be

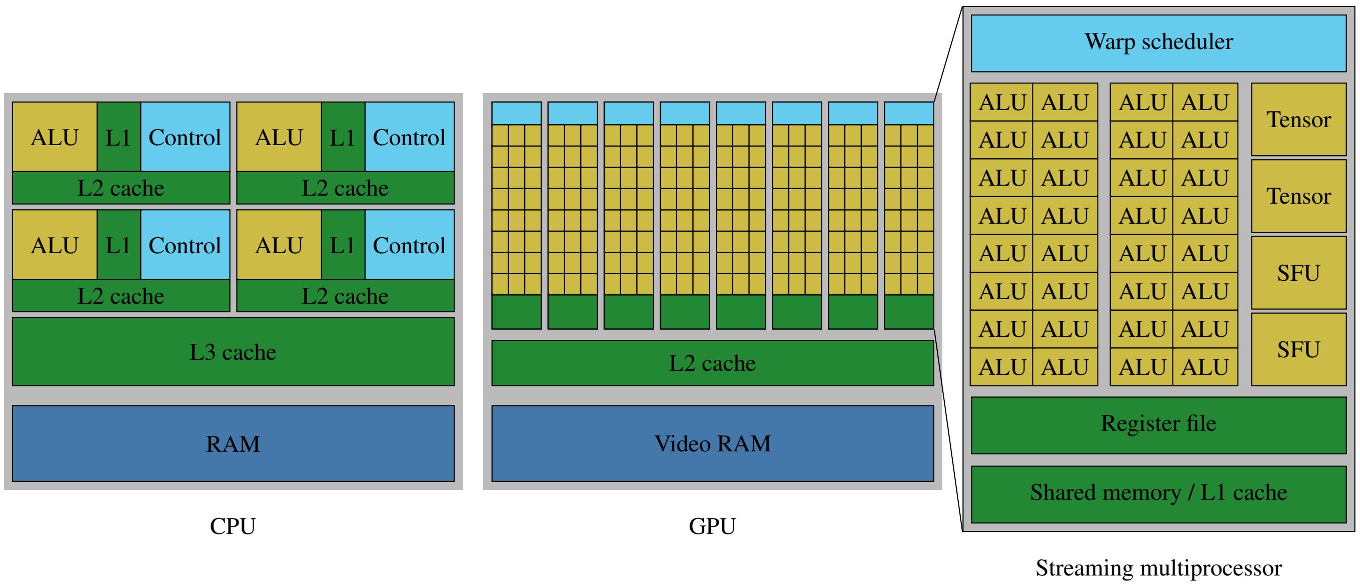


Figure A1. A schematic overview of an example CPU versus a GPU.

sufficiently expensive. Therefore, when a complex computation is performed, it can be worthwhile to perform an individual operation on the GPU even if it is not efficient when this minimizes the data transferred between GPUs and CPUs.

The main building block of a GPU is the **streaming multiprocessor (SM)**. Equipped with its own cache, registers, ALUs and control logic, a streaming multiprocessor can operate largely autonomously and is similar to a core on the CPU. The number of streaming multiprocessors varies widely between chips and performance considerations for a program with respect to this number are similar to varying the number of CPU cores, i.e. weak and strong scaling. High end GPUs have up to 128 streaming multiprocessors.

Conceptually, each streaming multiprocessor is a single instruction, multiple data (SIMD) processor that achieves computational efficiency by data parallel processing, i. e. through a number of threads that perform the same computation on different data. Such a **warp** of threads is comparable to a CPU thread where every operation is performed as SIMD. However, warps are much wider than the vector units commonly found on CPUs. Fig. A1 alludes to this with the ALUs (or *CUDA cores*) being grouped in sets of 16, though in practice, a common warp size is 32. As a consequence of the SIMD paradigm, branches ("if" statements) can be hugely detrimental to the performance if they cause frequent *warp divergence*, i.e. the selection of different code paths within a warp. In that case, the whole warp will execute each taken branch and undesired results are just masked out at the end. A similar problem exists for CPU code, where branches can prevent effective vectorization.

Each GPU thread has its own set of registers and private *local memory* if additional space is needed. Since local memory is just a special address space in the VRAM, it has very high latency, and it puts additional strain on the same memory bus as global memory accesses. While automated caches alleviate the issue of slow device memory to some extent, much better performance can often be achieved by utilizing **shared memory**. This memory space is part of the L1 cache, allowing for fast random access, but is manually programmed so that one can ensure that the right data is held in the cache. The same shared memory space is shared between multiple threads.

Multiple warps together form a **thread block** and all threads in the same block are resident on a single streaming multiprocessor at the same time. Instead of having a fixed number of registers available to each thread, on GPUs, registers are allocated as needed for each kernel from the **register file**. Keeping the state of multiple warps in registers is necessary to make context switches between them cheap which is key to achieve a high throughput. A high per-thread register requirement can therefore be detrimental to performance, because it decreases the possible thread block size. A lower *occupancy* means that fewer warps are available to the scheduler, which in turn increases the likelihood of wasted cycles. An important feature of thread blocks is their ability to effectively coordinate work. Synchronization is possible via a lightweight barrier and all threads in a block have access to the same shared memory, thereby making it possible to share intermediate results.

All thread blocks are organized into a **grid** and independently executed on the available streaming multiprocessors. Synchronization between blocks usually happens only once the whole grid is finished, although another intermediate level called *cooperative groups* is available on the latest hardware.

In addition to the general purpose ALUs, streaming multiprocessors are equipped with a number of fixed function units to accelerate specific computations. These include **tensor cores** which compute matrix-matrix multiplications and **special function units (SFUs)** that compute approximations for certain transcendental functions. Programming with tensor cores requires special consideration because tensor cores are controlled at the warp level, whereas the program is written in terms of threads.

585 ## Appendix B: Code

```
template<DG> using DGVec = Eigen::Matrix<T, Eigen::Dynamic, DG>;

template <int CG, int DGstress, int DGadvection>
void StressUpdateHighOrder(const VPParameters& vpparameters,
    const ParametricMomentumMap<CG>& pmap, const ParametricMesh& smesh,
    DGVec<DGstress>& S11, DGVec<DGstress>& S12, DGVec<DGstress>& S22,
    const DGVec<DGstress>& E11, const DGVec<DGstress>& E12, const DGVec<DGstress>& E22,
    const DGVec<DGadvection>& H, const DGVec<DGadvection>& A, double alpha, double beta)
{
    constexpr int NGP = ((DGstress == 8) || (DGstress == 6)) ? 3 : (DGstress == 3 ? 2 : -1);
    using EdgeVec = Eigen::Matrix<T, 1, NGP * NGP>;
#pragma omp parallel for
    for (size_t i = 0; i < smesh.nelements; ++i) {
        auto hGauss = (H.row(i) * PSI<DGadvection, NGP>).array().max(0.0).matrix();
        auto aGauss = (A.row(i) * PSI<DGadvection, NGP>).array().max(0.0).min(1.0).matrix();
        EdgeVec P = (_vpparameters.Pstar * hGauss.array()
            * (-20.0 * (1.0 - aGauss.array())).exp()).matrix();

        const EdgeVec e11Gauss = E11.row(i) * PSI<DGstress, NGP>;
        const EdgeVec e12Gauss = E12.row(i) * PSI<DGstress, NGP>;
        const EdgeVec e22Gauss = E22.row(i) * PSI<DGstress, NGP>;
        const auto DELTA = (vpparameters.DeltaMin * vpparameters.DeltaMin
            + 1.25 * (e11Gauss.array().square() + e22Gauss.array().square())
            + 1.50 * e11Gauss.array() * e22Gauss.array() + e12Gauss.array().square())
                    .sqrt().matrix();

        const T alphaInv = 1.0 / alpha;
        const T fac = 1.0 - alphaInv;
        const EdgeVec PDelta = P.array() / DELTA.array();
        S11.row(i) = fac * S11.row(i) + (pmap.iMJwPSI[i]
            * (alphaInv * (PDelta.array()
                * ((5.0 / 8.0) * e11Gauss.array() + (3.0 / 8.0) * e22Gauss.array())
                - 0.5 * P.array()).matrix().transpose())).transpose();
        S12.row(i) = fac * S12.row(i) + (pmap.iMJwPSI[i]
            * (alphaInv * (PDelta.array() * (1.0 / 4.0) * e12Gauss.array())
                .matrix().transpose())).transpose();
        S22.row(i) = fac * S22.row(i) + (pmap.iMJwPSI[i]
            * (alphaInv * (PDelta.array()
                * ((5.0 / 8.0) * e22Gauss.array() + (3.0 / 8.0) * e11Gauss.array())
                    - 0.5 * P.array()).matrix().transpose())).transpose();
    }
}

```

Listing 2. Implementation of the stress update with Eigen. The method is generic in the degrees of freedom of the different cG and dG elements. The `matrix()` and `array()` methods change the type of an expression to differentiate between matrix and component-wise operations and are no-ops during runtime.

References

XLA: an open-source machine learning compiler, https://github.com/openxla/xla, [accessed 28-November-2023], 2023.

Alpay, A. and Heuveline, V.: One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends, in: International Workshop on OpenCL, ACM, https://doi.org/10.1145/3585341.3585351, 2023.

Aoun, M. R. E., Tidjon, L. N., Rombaut, B., Khomh, F., and Hassan, A. E.: An Empirical Study of Library Usage and Dependency in Deep Learning Frameworks, https://doi.org/10.48550/ARXIV.2211.15733, 2022.

Banderier, H., Zeman, C., Leutwyler, D., Rüdisühli, S., and Schär, C.: Reduced floating-point precision in regional climate simulations: an ensemble-based statistical verification, Geoscientific Model Development, 17, 5573–5586, https://doi.org/10.5194/gmd-17-5573-2024, 2024.

Bauer, P., Dueben, P. D., Hoefler, T., Quintino, T., Schulthess, T. C., and Wedi, N. P.: The digital revolution of Earth-system science, Nature Computational Science, 1, 104–113, https://doi.org/10.1038/s43588-021-00023-0, 2021a.

Bauer, P., Stevens, B., and Hazeleger, W.: A digital twin of Earth for the green transition, Nature Climate Change, 11, 80–83, https://doi.org/10.1038/s41558-021-00986-y, 2021b.

Bedrunka, M. C., Wilde, D., Kliemank, M., Reith, D., Foysi, H., and Krämer, A.: Lettuce: PyTorch-Based Lattice Boltzmann Framework, pp. 40–55, Springer International Publishing, ISBN 9783030905392, https://doi.org/10.1007/978-3-030-90539-2_3, 2021.

Bolz, J., Farmer, I., Grinspun, E., and Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graph., 22, 917–924, https://doi.org/10.1145/882262.882364, 2003.

Bouchat, A., Hutter, N., Chanut, J., Dupont, F., Dukhovskoy, D., Garric, G., Lee, Y. J., Lemieux, J., Lique, C., Losch, M., Maslowski, W., Myers, P. G., Ólason, E., Rampal, P., Rasmussen, T., Talandier, C., Tremblay, B., and Wang, Q.: Sea Ice Rheology Experiment (SIREx): 1. Scaling and Statistical Properties of Sea-Ice Deformation Fields, Journal of Geophysical Research: Oceans, 127, https://doi.org/10.1029/2021jc017667, 2022.

Bouillon, S., Fichefet, T., Legat, V., and Madec, G.: The elastic–viscous–plastic method revisited, Ocean Modelling, 71, 2–12, https://doi.org/10.1016/j.ocemod.2013.05.013, 2013.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q.: JAX: composable transformations of Python+NumPy programs, http://github.com/google/jax, 2018.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware, ACM Trans. Graph., 23, 777–786, https://doi.org/10.1145/1015706.1015800, 2004.

Cao, K., Wu, Q., Wang, L., Wang, N., Cheng, H., Tang, X., Li, D., and Wang, L.: GPU-HADVPPM V1.0: a high-efficiency parallel GPU design of the piecewise parabolic method (PPM) for horizontal advection in an air quality model (CAMx V6.10), Geoscientific Model Development, 16, 4367–4383, https://doi.org/10.5194/gmd-16-4367-2023, 2023.

Dansereau, V., Weiss, J., Saramito, P., and Lattes, P.: A Maxwell elasto-brittle rheology for sea ice modelling, The Cryosphere, 10, 1339–1359, https://doi.org/10.5194/tc-10-1339-2016, 2016.

Demeure, N., Kisner, T., Keskitalo, R., Thomas, R., Borrill, J., and Bhimji, W.: High-level GPU code: a case study examining JAX and OpenMP., in: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, ACM, https://doi.org/10.1145/3624062.3624186, 2023.

Denny, J. E., Lee, S., and Vetter, J. S.: CLACC: Translating OpenACC to OpenMP in Clang, in: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), IEEE, https://doi.org/10.1109/llvm-hpc.2018.8639349, 2018.

Dublish, S., Nagarajan, V., and Topham, N.: Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in GPUs, in: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, https://doi.org/10.1109/ispass.2017.7975295, 2017.

Esau, I. N.: Amplification of turbulent exchange over wide Arctic leads: Large-eddy simulation study, Journal of Geophysical Research: Atmospheres, 112, https://doi.org/10.1029/2006jd007225, 2007.

Feltham, D.: Sea Ice Rheology, Annual Review of Fluid Mechanics, 40, 91–112, https://doi.org/10.1146/annurev.fluid.40.111406.102151, 2008.

Guennebaud, G., Jacob, B., et al.: Eigen v3, http://eigen.tuxfamily.org, 2010.

Hatfield, S., Chantry, M., Düben, P., and Palmer, T.: Accelerating High-Resolution Weather Models with Deep-Learning Hardware, in: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '19, ACM, https://doi.org/10.1145/3324989.3325711, 2019.

Hibler, W. D.: A Dynamic Thermodynamic Sea Ice Model, Journal of Physical Oceanography, 9, 815–846, https://doi.org/10.1175/1520-0485(1979)009<0815:adtsim>2.0.co;2, 1979.

Hunke, E., Allard, R., Blain, P., Blockley, E., Feltham, D., Fichefet, T., Garric, G., Grumbine, R., Lemieux, J.-F., Rasmussen, T., Ribergaard, M., Roberts, A., Schweiger, A., Tietsche, S., Tremblay, B., Vancoppenolle, M., and Zhang, J.: Should Sea-Ice Modeling Tools Designed for Climate Research Be Used for Short-Term Forecasting?, Current Climate Change Reports, 6, 121–136, https://doi.org/10.1007/s40641-020-00162-y, 2020.

Hutter, N., Zampieri, L., and M. Losch, M.: Leads and ridges in Arctic sea ice from RGPS data and a new tracking algorithm, The Cryosphere, 13, 627–645, https://doi.org/10.5194/tc-13-627-2019, 2019.

Hutter, N., Bouchat, A., Dupont, F., Dukhovskoy, D., Koldunov, N., Lee, Y. J., Lemieux, J., Lique, C., Losch, M., Maslowski, W., Myers, P. G., Ólason, E., Rampal, P., Rasmussen, T., Talandier, C., Tremblay, B., and Wang, Q.: Sea Ice Rheology Experiment (SIREx): 2. Evaluating Linear Kinematic Features in High-Resolution Sea Ice Simulations, Journal of Geophysical Research: Oceans, 127, https://doi.org/10.1029/2021jc017666, 2022.

Ikuyajolu, O. J., Roekel, L. V., Brus, S. R., Thomas, E. E., Deng, Y., and Sreepathi, S.: Porting the WAVEWATCH III (v6.07) wave action source terms to GPU, Geoscientific Model Development, 16, 1445–1458, https://doi.org/10.5194/gmd-16-1445-2023, 2023.

Jakob, C., Gettelman, A., and Pitman, A.: The need to operationalize climate modelling, Nature Climate Change, 13, 1158–1160, https://doi.org/10.1038/s41558-023-01849-4, 2023.

Jendersie, R., Richter, T., and Lessig, C.: neXtSIM_DG dynamical core GPU experiments, https://doi.org/10.5281/ZENODO.13711171, 2024.

Jülich Supercomputing Centre: JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre, Journal of large-scale research facilities, 7, https://doi.org/10.17815/jlsrf-7-183, 2021.

Kauker, F., Bertino, L., Bracher, A., Gabarró, C., Garric, G., Hughes, N., Kaminski, T., Lavergne, T., Malnes, E., Mustonen, T., Pedersen, L. T., Schauer, U., Scholze, M., Schyberg, H., Tietsche, S., Wagner, P., and Wilkinson, J.: A roadmap towards a European end-to-end operational system for monitoring and forecasting of the Polar Regions, KEPLER Deliverable Report, https://kepler380449468.files.wordpress.com/2021/08/kepler-deliverable-report-5.2-1.pdf, 2021.

Kimpson, T., Paxton, E. A., Chantry, M., and Palmer, T.: Climate-change modelling at reduced floating-point precision with stochastic rounding, Quarterly Journal of the Royal Meteorological Society, 149, 843–855, https://doi.org/10.1002/qj.4435, 2023.

Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S.: Machine learning–accelerated computational fluid dynamics, Proceedings of the National Academy of Sciences, 118, https://doi.org/10.1073/pnas.2101784118, 2021.

Kochkov, D., Yuval, J., Langmore, I., Norgaard, P., Smith, J., Mooers, G., Lottes, J., Rasp, S., Düben, P., Klöwer, M., Hatfield, S., Battaglia, P., Sanchez-Gonzalez, A., Willson, M., Brenner, M. P., and Hoyer, S.: Neural General Circulation Models, 2023.

Krüger, J. and Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms, in: ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, p. 234–es, Association for Computing Machinery, New York, NY, USA, ISBN 9781450378338, https://doi.org/10.1145/1198555.1198795, 2005.

Kwok, R.: Deformation of the Arctic Ocean sea ice cover between November 1996 and April 1997: a qualitative survey, Solid Mech. Appl., 94, 315–322, https://doi.org/10.1007/978-94-015-9735-7_26, 2001.

Lang, S. T. K., Dawson, A., Diamantakis, M., Dueben, P., Hatfield, S., Leutbecher, M., Palmer, T., Prates, F., Roberts, C. D., Sandu, I., and Wedi, N.: More accuracy with less precision, Quarterly Journal of the Royal Meteorological Society, 147, 4358–4370, https://doi.org/10.1002/qj.4181, 2021.

Liu, J., Chen, Z., Hu, Y., Zhang, Y., Ding, Y., Cheng, X., Yang, Q., Nerger, L., Spreen, G., Horton, R., Inoue, J., Yang, C., Li, M., and Song, M.: Towards reliable Arctic sea ice prediction using multivariate data assimilation, Science Bulletin, 64, 63–72, https://doi.org/10.1016/j.scib.2018.11.018, 2019.

Marcq, S. and Weiss, J.: Influence of sea ice lead-width distribution on turbulent heat transfer between the ocean and the atmosphere, The Cryosphere, 6, 143–156, https://doi.org/10.5194/tc-6-143-2012, 2012.

Marsan, D., Stern, H., Lindsay, R., and Weiss, J.: Scale Dependence and Localization of the Deformation of Arctic Sea Ice, Physical Review Letters, 93, 178 501, https://doi.org/10.1103/physrevlett.93.178501, 2004.

Mehlmann, C. and Richter, T.: A modified global Newton solver for viscous-plastic sea ice models, Ocean Modeling, 116, 96–107, https://doi.org/10.1016/j.ocemod.2017.06.001, 2017.

Mehlmann, C., Danilov, S., Losch, M., Lemieux, J. F., Hutter, N., Richter, T., Blain, P., Hunke, E. C., and Korn, P.: Simulating Linear Kinematic Features in Viscous-Plastic Sea Ice Models on Quadrilateral and Triangular Grids With Different Variable Staggering, Journal of Advances in Modeling Earth Systems, 13, https://doi.org/10.1029/2021ms002523, 2021.

Meyer, J., Alpay, A., Hack, S., Fröning, H., and Heuveline, V.: Implementation Techniques for SPMD Kernels on CPUs, in: International Workshop on OpenCL, IWOCL '23, ACM, https://doi.org/10.1145/3585341.3585342, 2023.

NVIDIA: CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, [accessed 28-November-2023], 2023a.

NVIDIA: TensorRT: an SDK for high-performance deep learning inference, https://developer.nvidia.com/tensorrt, [accessed 28-November-2023], 2023b.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library, in: Advances in Neural Information Processing Systems 32, edited by Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché Buc, F., Fox, E., and Garnett, R., pp. 8024–8035, Curran Associates, Inc., http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf, 2019.

Paxton, E. A., Chantry, M., Klöwer, M., Saffin, L., and Palmer, T.: Climate Modeling in Low Precision: Effects of Both Deterministic and Stochastic Rounding, Journal of Climate, 35, 1215–1229, https://doi.org/10.1175/jcli-d-21-0343.1, 2022.

PyTorch-devs: TorchInductor: a PyTorch-native Compiler with Define-by-Run IR and Symbolic Shapes, https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747, [accessed 13-May-2024], 2023.

Rasmussen, T. A. S., Poulsen, J., Ribergaard, M. H., Sasanka, R., Craig, A. P., Hunke, E. C., and Rethmeier, S.: Refactoring the EVP solver for improved performance – a case study based on CICE v6.5, https://doi.org/10.5194/gmd-2024-40, 2024.

Richter, T., Dansereau, V., Lessig, C., and Minakowski, P.: A dynamical core based on a discontinuous Galerkin method for higher-order finite-element sea ice modeling, Geoscientific Model Development, 16, 3907–3926, https://doi.org/10.5194/gmd-16-3907-2023, 2023.

Sauer, J. A. and Muñoz-Esparza, D.: The FastEddy® Resident-GPU Accelerated Large-Eddy Simulation Framework: Model Formulation, Dynamical-Core Validation and Performance Benchmarks, Journal of Advances in Modeling Earth Systems, 12, https://doi.org/10.1029/2020ms002100, 2020.

Schär, C., Fuhrer, O., Arteaga, A., Ban, N., Charpilloz, C., Girolamo, S. D., Hentgen, L., Hoefler, T., Lapillonne, X., Leutwyler, D., Osterried, K., Panosetti, D., Rüdisühli, S., Schlemmer, L., Schulthess, T. C., Sprenger, M., Ubbiali, S., and Wernli, H.: Kilometer-Scale Climate Models: Prospects and Challenges, Bulletin of the American Meteorological Society, 101, E567–E587, https://doi.org/10.1175/bams-d-18-0167.1, 2020.

Servat, H., Rossi, G., Duran, A., and Narayanaswamy, R.: On the Migration of OpenACC-Based Applications into OpenMP 5+, in: OpenMP in a Modern World: From Multi-device Support to Meta Programming, pp. 127–141, Springer International Publishing, https://doi.org/10.1007/978-3-031-15922-0_9, 2022.

Stevens, B., Satoh, M., Auger, L., Biercamp, J., Bretherton, C. S., Chen, X., Düben, P., Judt, F., Khairoutdinov, M., Klocke, D., Kodama, C., Kornblueh, L., Lin, S.-J., Neumann, P., Putman, W. M., Röber, N., Shibuya, R., Vanniere, B., Vidale, P. L., Wedi, N., and Zhou, L.: DYAMOND: the DYnamics of the Atmospheric general circulation Modeled On Non-hydrostatic Domains, Progress in Earth and Planetary Science, 6, 61, https://doi.org/10.1186/s40645-019-0304-z, 2019.

Sun, J., Dennis, J. M., Mickelson, S. A., Vanderwende, B., Gettelman, A., and Thayer-Calder, K.: Acceleration of the Parameterization of Unified Microphysics Across Scales (PUMAS) on the Graphics Processing Unit (GPU) With Directive-Based Methods, Journal of Advances in Modeling Earth Systems, 15, https://doi.org/10.1029/2022ms003515, 2023.

The Khronos Group, I.: SYCL: a cross-platform abstraction layer for heterogeneous computing, https://www.khronos.org/sycl, [accessed 30-November-2023], 2023a.

The Khronos Group, I.: SYCL 2020 standard - language restrictions, https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_language_restrictions_in_kernels, [accessed 21-November-2023], 2023b.

Tillet, P., Kung, H. T., and Cox, D.: Triton: an intermediate language and compiler for tiled neural network computations, in: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, PLDI '19, ACM, https://doi.org/10.1145/3315508.3329973, 2019.

Tintó Prims, O., Acosta, M. C., Moore, A. M., Castrillo, M., Serradell, K., Cortés, A., and Doblas-Reyes, F. J.: How to use mixed precision in ocean models: exploring a potential reduction of numerical precision in NEMO 4.0 and ROMS 3.6, Geoscientific Model Development, 12, 3135–3148, https://doi.org/10.5194/gmd-12-3135-2019, 2019.

Torch-TensorRT-devs: Torch-TensorRT: Ahead of Time (AOT) compiling for PyTorch JIT and FX, https://github.com/pytorch/TensorRT, 2024.

Trott, C. R., Lebrun-Grandie, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J.:

Kokkos 3: Programming Model Extensions for the Exascale Era, IEEE Transactions on Parallel and Distributed Systems, 33, 805–817, https://doi.org/10.1109/tpds.2021.3097283, 2022.

Ðukić, J. and Mišić, M.: An Evaluation of Directive-Based Parallelization on the GPU Using a Parboil Benchmark, Electronics, 12, 4555, https://doi.org/10.3390/electronics12224555, 2023.

Usha, R., Pandey, P., and Mangala, N.: A Comprehensive Comparison and Analysis of OpenACC and OpenMP 4.5 for NVIDIA GPUs, in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, https://doi.org/10.1109/hpec43674.2020.9286203, 2020.

Xie, J., Bertino, L., Counillon, F., Lisæter, K. A., and Sakov, P.: Quality assessment of the TOPAZ4 reanalysis in the Arctic over the period 1991–2013, Ocean Science, 13, 123–144, https://doi.org/10.5194/os-13-123-2017, 2017.

Ólason, E., Rampal, P., and Dansereau, V.: On the statistical properties of sea-ice lead fraction and heat fluxes in the Arctic, The Cryosphere, 15, 1053–1064, https://doi.org/10.5194/tc-15-1053-2021, 2021.

Ólason, E., Boutin, G., Korosov, A., Rampal, P., Williams, T., Kimmritz, M., Dansereau, V., and Samaké, A.: A New Brittle Rheology and Numerical Framework for Large-Scale Sea-Ice Models, Journal of Advances in Modeling Earth Systems, 14, https://doi.org/10.1029/2021ms002685, 2022.