Porting Real-World Applications to GPU Clusters: A Celerity and CRONOS Case Study

Philipp Gschwandtner*, Ralf Kissmann†, David Huber†, Philip Salzmann‡,
Fabian Knorr‡, Peter Thoman‡, Thomas Fahringer‡,
*Research Center HPC, University of Innsbruck, Innsbruck, Austria
philipp.gschwandtner@uibk.ac.at
†Institute for Astro- and Particle Physics, University of Innsbruck, Innsbruck, Austria
{ralf.kissmann, david.huber}@uibk.ac.at
‡Department of Computer Science, University of Innsbruck, Innsbruck, Austria
{philip.salzmann, fabian.knorr, peter.thoman, thomas.fahringer}@uibk.ac.at

***Abstract*—Accelerator clusters are an ongoing trend in high performance computing, continuously gaining traction and forming a ubiquitous hardware resource for domain scientists to run large-scale simulations on. However, there is often a gap between new hardware technologies and adoption by legacy code bases. Porting real-world applications to new programming models is a difficult undertaking, aggravated by the need for support for both distributed-memory and accelerator parallelism. In this work, we present a case study of porting CRONOS, a real-world code from the field of magnetohydrodynamics, to Celerity, a high-level programming model for distributed-memory accelerator clusters. We discuss the numerical, algorithmic and implementation properties of the application and motivate our decisions for adapting them where necessary. Preliminary results show a parallel efficiency of up to 87% for 16 GPUs.**

***Index Terms*—accelerators, gpus, sycl, celerity, distributed memory, parallel programming**

I. INTRODUCTION

High performance computing (HPC) is a branch of science that connects communities from computer science and the domain sciences. The work of researchers and engineers, who develop hardware and software stacks—usually aimed at offering the maximum performance possible—is employed by scientists in the domain sciences in order to advance their own field of research and compute solutions to problems of ever-increasing complexity.

Application software is typically developed by domain scientists, who are naturally not experts in the field of computer science. However, contemporary HPC architectures are increasingly complex systems, with multiple nodes, sockets, cores, or on- and off-chip interconnects, all of which need to be mastered in order to obtain maximum performance. Furthermore, HPC is fast-evolving. Many programming language and model standards are updated at least every few years and hardware platforms offer a continuously increasing plethora of new features and performance characteristics at a similar pace. New hardware technologies include extended instruction sets, continuously increasing core counts or improved interconnects on all levels of HPC hardware. Specifically, accelerator technologies are a good example for such technologies, showing a comparatively high adoption rate and increasing market share in HPC. The Top500 list [1] of the fastest supercomputers shows that the number of systems equipped with accelerators or co-processors increased by 71% between November 2016 and November 2020, with a current share of 147 systems.

The fact that two paradigms—distributed memory parallelism and accelerator computing—have established themselves in HPC makes application development a tedious and error-prone process. While programming models for both exist, e.g. MPI and CUDA, naively combining them to a programming paradigm often referred to as MPI+X has several drawbacks, including the application developer's responsibility for efficient domain decomposition—a key factor for performance in distributed-memory parallelism. There are several endeavors that aim at mitigating this issue, including Celerity [2], a SYCL-based high-level distributed-memory accelerator programming model that aims at automatic distributed-memory parallelism.

In this work, we present a case study of CRONOS [3], a large magnetohydrodynamics application originally implemented using MPI only, which we port to Celerity. Specifically, our contributions are

- a partial Celerity implementation of CRONOS,
- detailed discussion of the algorithm and implementation considerations and changes necessary when porting such applications to SYCL and Celerity, and
- performance analysis of our implementation.

The paper is structured as follows: Section II details on SYCL and Celerity, focusing on the features and characteristics pertinent to our work. Section III presents our use case application, CRONOS. Our requirements, considerations and experiences are described in Section IV, whereas Section V discusses our performance results. Finally, Section VI provides our conclusion and outlook to future work.

II. SYCL AND CELERITY

In this section, we will briefly describe the key characteristics of our chosen programming model and the motivation for our decision.

A. Accelerator Programming Models

There are several programming languages and models to choose from when working with accelerators and various ways of classifying them. First, there is the distinction of open or vendor-specific models. CUDA for example is very popular and established in the HPC community, but it lacks support for devices from vendors other than Nvidia. On the other hand, one can categorize these accelerator-enabling technologies by their nature, i.e. whether they are new programming languages such as Chapel [4], programming language extensions such as OpenMP [5], or rather libraries that are fully embedded in a host language such as SYCL [6]. They all vary in their usage and parallelism feature support (e.g. accelerators, automatic data decomposition and distribution, fault tolerance) and often offer a difficult trade-off when facing the task of implementing completely new applications.

However, when porting legacy applications, one can make the case for focusing on a specific subset of these programming models. Specifically, the new programming model should have strong debugging support due to the already inherently tedious task of working with large legacy code bases. As we will outline below, this leads us to believe that Celerity [2], a SYCL-based high-level distributed-memory parallel programming model, is a good candidate.

SYCL is an open industry standard for programming both CPUs and accelerators using modern, single-source, high-level C++. Its API can be regarded as an embedded domain-specific language and offers the capability of specifying callable objects in C++ that are executed as kernels on compute devices. SYCL offers several operators such as `parallel_for` that resemble frequently-used parallelism patterns. Being single-source, C++-based, and vendor-inspecific, SYCL has strong advantages over competitor models such as OpenCL or CUDA, by facilitating implementations with debugging support. Given that the application developer is working with pure C++, SYCL code can be easily executed on CPUs, giving access to a plethora of existing debugging and analysis tools. As we will describe in Section IV, this facilitates porting legacy applications—in contrast to parallel programming models based on extensions or completely new languages, which require explicit toolchain support.

B. Distributed Accelerator Programming

One disadvantage of SYCL is its lack of support for distributed memory systems. In order to fully use e.g. a cluster consisting of several nodes equipped with GPUs, SYCL still requires a second programming model such as MPI to take care of inter-node communication (intra-node communication between host and device memory are handled by SYCL). This MPI+X programming paradigm is frequently used in the community but has several drawbacks. First, it requires the developer to master two distinct programming models as well as their potentially complex interaction in terms of both functionality and performance. Second, it shifts the burden of writing efficient code to the application developer, since the performance and scalability of many large-scale distributed memory applications is largely driven by efficient domain decomposition, which neither MPI nor SYCL handle automatically.

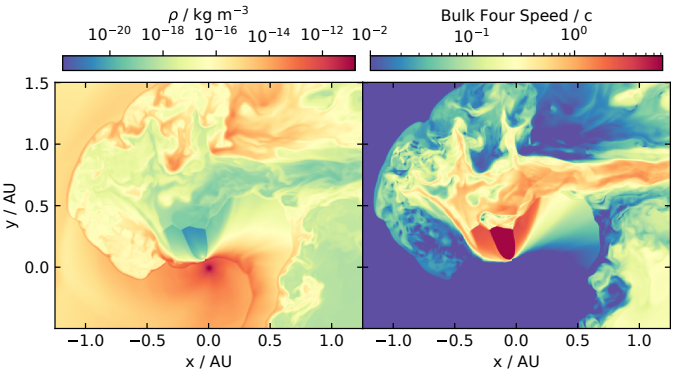

Fig. 1: Simulation of the stellar and pulsar wind interaction in the gamma-ray binary LS 5039 performed using CRONOS [7].

A possible solution to this approach is Celerity, a programming model based on SYCL, developed and maintained by the University of Innsbruck and the University of Salerno. Celerity attempts to reduce the aforementioned developer responsibilities by enabling automatic work- and data decomposition and distribution for a large subset of parallel applications. Furthermore, one of the design goals of Celerity is an API that closely resembles SYCL in order to minimize any adoption barriers. As a result, Celerity applications look very similar to SYCL applications [2], with the most substantial change being the addition of so-called *range mappers*. These range mappers, to be specified by the user, are the means by which Celerity accomplishes automatic data decomposition. They specify the spatial range of access, whether for input or output data, for every work item (i.e. spatial point of computation). Implementation-wise, they are callable C++ objects that take a `chunk` (a range of work item IDs) and return the computed data access range. Celerity provides several pre-defined range mappers that fit commonly-used programming patterns such as linear algebra or structured grid methods.

III. CRONOS

The CRONOS code is a magnetohydrodynamics (MHD) code developed for the solution of plasma-dynamical problems in astrophysics and space science. The details of the code can be found in [3]. The code solves hyperbolic conservations laws of the form

$$\frac{\partial \mathbf{w}}{\partial t} + \nabla \cdot \mathbf{F}\left(\mathbf{w}, \mathbf{r}, t\right) = \mathbf{s} \tag{1}$$

where $\mathbf{w}$ is a density of a conserved quantity and $\mathbf{F}$ is the related hyperbolic flux.

A. Physical Simulations Using Cronos

Currently, CRONOS contains three different systems of hyperbolic partial-differential equations (PDEs): classical hydrodynamics (HD) or MHD and relativistic hydrodynamics (RHD).

Using HD as an example, CRONOS solves the following system of PDEs

$$\frac{\partial n}{\partial t} + \nabla \cdot (n\mathbf{u}) = 0 \tag{2}$$

$$\frac{\partial (mn\,\mathbf{u})}{\partial t} + \nabla \cdot (mn\,\mathbf{uu}) + \nabla p = \mathbf{f} \tag{3}$$

$$\frac{\partial e}{\partial t} + \nabla \cdot ((e+p)\,\mathbf{u}) = \mathbf{u} \cdot \mathbf{f}, \tag{4}$$

where n denotes the number density of the gas particles, m their average mass, $\mathbf{u}$ the velocity of the flow, p its thermal pressure, and e the total energy density comprised of thermal and kinetic energy density. Additionally, we allow for an external force density f that can either be a result of the considered geometry or can be supplied by the user to consider additional physics. Also in case of MHD or RHD the equations show the same fundamental form.

This means that CRONOS was developed to model (magnetized) fluids both in a classical and a relativistic framework with astrophysical applications in mind. In this context, the formation and evolution of shocks or turbulence is often of central interest. In order to be able to correctly describe their dynamics in the modelled fluids, CRONOS employs a finite-volume scheme to solve the respective equations. Therefore, the code is particularly efficient and stable for high-Mach-number flows, whereas a finite-difference code would be more efficient in the low-Mach-number regime. As required by the first scientific applications, high-Mach-number turbulence needs to be reliably modelled using CRONOS. These applications include investigations of turbulence in the interstellar medium [8] or in accretions disks [9].

Recently, applications of CRONOS focus on the evolution and interaction of highly supersonic winds. These applications either relate to the Sun [10], [11] or to massive stellar binary systems [12], for which the code was extended by using a number of passive tracer-fields to account for the transport of energetic particles [13], [14].

In the latter cases, we aim to model the gamma-ray emission from colliding-wind binary systems [15] and from so-called gamma-ray binaries [7], such as depicted in Fig. 1. For this, we model the dynamics of the winds launched by both companions, which eventually collide and form a so-called wind-collision region (WCR), either using MHD for the case of colliding-wind binaries or using RHD in gamma-ray binaries. At the shock-fronts enclosing the wind-collision region, we additionally inject energetic particles, for which an additional transport equation is solved. In particular, advection of the energetic particles with the fluid can readily be treated using the standard solution methods in CRONOS.

Especially for such binary systems, it is necessary to take a large computational domain into account. Extents that span over several times of the orbital separation are usually required to correctly describe the acceleration of the massive winds and/or their interaction leading to the formation of the WCR.

Within the WCR, the colliding winds are separated by a contact discontinuity. Since both winds are supersonic but reach different speeds at the collision, one is left with a very high velocity shear at the contact discontinuity. Even in the relativistic case, this leads to an efficient growth of Kelvin-Helmholtz instabilities at this location, which triggers the development of turbulence in the system. Since the scales on which these instabilities can grow are usually much smaller than the stellar separation, our simulations require high spatial resolutions to cover both the dynamics of the system and the evolving turbulence.

In particular, the simulation of gamma-ray binaries require very large spatial resolutions and simulation boxes together with a small time step.

B. Brief Overview of the Numerical Solver

To solve the hyperbolic system, CRONOS uses a finite-volume scheme based on the ideas introduced by Godunov [16]: when integrating a system as given by Eq. (1) over the volume of given cell, one can show that the cell average $\bar{\mathbf{w}}$, i.e. the value of $\mathbf{w}$ averaged over the cell, depends on the integral of the flux function $\mathbf{F}$ over the cell-surface. An advancement in time can thus be achieved by first obtaining a suitable estimation of the respective flux averages. For this, Godunov assumed in his original work that $\mathbf{w}$ is constant within each cell, which leads to jumps in the considered quantities at the interfaces to its neighboring cells. Such discontinuities are also known as Riemann problems, which have been extensively studied in the past for many different systems of equations and can be solved analytically by so-called Riemann solvers. Ultimately, these solvers are used to compute approximations of the fluxes at the cell interfaces that are required to update the cell averages for the next time step.

In CRONOS, we solve the hyperbolic system using a semi-discrete form of the finite-volume scheme, as further detailed in [3]. This means that the equations have only been integrated and hence discretized in space, but not in time. This allows the application of different standard time-integration schemes—in our case a second- or third-order Runge-Kutta TVD scheme [17], [18]—to integrate the system in time. In contrast toGodunov1959, we assume $\mathbf{w}$ to vary linearly in space, leading to a second-order spatial accuracy of the scheme. In the semi-discrete form of the scheme, we need to solve a Riemann problem for each sub-step of the Runge-Kutta time integrator. To save computation time, we use an approximate solution for the Riemann problem.

Dealing with a hyperbolic system of equations means that the speed at which signals and disturbances can propagate are finite. Applying the Courant-Friedrichs-Lewy (CFL) condition (see [19]) within the code, we made sure that any signal can only propagate less than the width of a numerical cell during one time step to maintain numerical stability. This also means that the numerical stencil is rather small, thus allowing for efficient parallelization of the code with only minimal need for communication of boundary values.

At each substep of these time-integration schemes, we then use the following procedure to solve the spatial part of the problem:

1) Reconstruction of the left- and right-sided point values for a given cell interface from $\bar{\mathbf{w}}$.
2) (Approximate) solution of the Riemann problem at the cell interfaces, yielding a numerical value for the flux functions.
3) Computation of the change of cell averages of the base quantities via application of the numerical fluxes.

Relating to the stencil, point values at the cell interfaces are computed using a linear, one-dimensional sub-grid reconstruction in each dimension, where the slope is determined from the cell averages of the local cell and its direct neighbors. The reconstruction of a Riemann problem at a given interface thus involves the values of the next two cells on either side of the interface and consequently a stencil consisting of four cells is needed to obtain the numerical fluxes. Since a cell depends on the fluxes across two interfaces in each coordinate direction, the stencil to advance a given cell in time comprises five cells, including itself. Up to now, CRONOS has been parallelized by dividing the computational domain into N_{cores} identical hyperslabs, where N_{cores} is the number of computer cores used for a given simulation. To assure continuity between the different volumes, a surface with a thickness of two cells needs to be communicated between the different processes. The related MPI ghost cell exchange is very efficient as long as the volumes per core are sufficiently large.

Coupled with the need for large-scale computation for complex simulations that exceed local and even national infrastructures, CRONOS is also used successfully in an ongoing a PRACE access project and has shown to scale well up to at least 16,000 cores on the Joliot-Curie Rome system [20]. For the same reasons, we expect CRONOS to be highly suitable for GPU parallelism.

IV. PORTING CONSIDERATIONS

Porting legacy applications to new programming models is often a tedious task. Many long-term applications are maintained by a continuously changing team of researchers over several years, leading to differences in programming style or use of programming model and language features. In this section, we detail on considerations that have to be taken into account when porting CRONOS to Celerity, as well as experiences gathered during this process.

A. Original Implementation

CRONOS consists of approximately 40,000 lines of C++ code, along with approximately 3,000 lines of Python for output processing and a few hundred lines of shell scripts and makefiles that form the build system. However, it should be mentioned that a large portion of the code base is unused legacy code or provides multiple implementation variants of the same component. The C++ code is divided into two main components, CRONOS itself and CRONOSNUMLIB, a self-written matrix library for storing multi-dimensional arrays. CRONOS can be compiled into a sequential or a parallel version using preprocessor defines and requires an MPI-2 compliant implementation for distributed memory execution. Furthermore, output handling is done via (parallel) HDF5.

For our work presented in this paper, we will focus on the numerical solver discussed in Section III-B.

B. Implementation Obstacles and Changes

CRONOS holds several obstacles that need to be overcome in order to be ported to a high-level programming model such as Celerity.

First of all, CRONOS relies on input-driven, dynamically dispatched virtual function calls for flexibility and low-effort extension with new component implementations. However, since SYCL was originally closely aligned with OpenCL, its specification lacks support for calling function pointers or dynamically dispatched virtual member functions in device kernels. For this reason, virtual function calls in to-be kernel code need to be converted to e.g. manual, static type enumeration or curiously recurring template patterns (CRTPs). Note that this only affects device kernels and does not pose any limitations on virtual function calls on the host side.

Second and related to the previous aspect, CRONOS contains a lot of manual memory allocation management code. A multitude of calls to `new` and `delete` manually allocate and free memory for data structures in every time step. This can be partially attributed to the age of the application, which dates back at least to the early 2000s when C++98 was considered new. While this kind of manual memory management is regarded as an error-prone programming style by today's standards and should be removed in favor of more resilient patterns such as RAII, it can be left in place for code that executes on the host only. Nevertheless, the distributed memory address spaces of host and accelerator make it impossible to dereference host pointers on the device and vice versa. For this reason, we place data structures that need to be exchanged between host and device in SYCL buffers. As it turns out, most data structures are only temporarily needed on the device and hence are placed in the register file instead of in host-accessible memory buffers.

Third, our application—like most structured and unstructured grid applications—relies on ghost cells for both enabling distributed memory parallelism via halo exchanges and establishing boundary conditions. For this reason, it uses the CRONOSNUMLIB component of CRONOS that offers arbitrary multi-dimensional array index ranges, including negative ones, to create 3-dimensional tensors that represent the physical domain to be simulated. However, SYCL buffers do not support this indexing scheme, requiring us to re-align loop iteration spaces to be non-negative and provide ample padding for ghost cells. As a side-effect, this removes the entailed performance overhead of the otherwise omnipresent offset computations whenever an array element is accessed.

Beyond the aforementioned aspects, there are also several minor issues present in the original code, such as silent memory corruption errors or occurrences of deprecated programming styles that impede fast development. While these naturally need to be addressed, they are not immediately

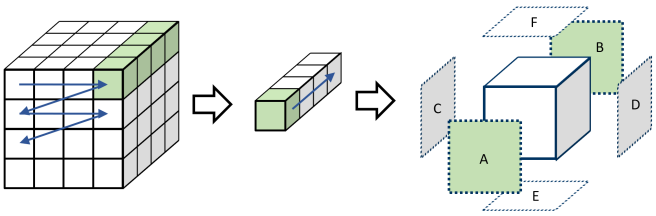

Fig. 2: Original sub-domain decomposition and simplified finite volume computation performed by each MPI rank for a single direction. Note that this is a simplified illustration for clarity, as only the center cell and its interfaces are shown. The actual computation requires two additional cells beyond interfaces A and B respectively, to correctly compute the fluxes, thus forming a 5-point stencil per direction.

relevant to our work described in this paper and are expected in any large legacy codebase without continuous integration or code coverage tests. For this reason, we omit their discussion.

Finally, in order to ensure productive porting and ease of development, we replace the platform-specific shell and Makefile scripting framework with a more modern and generic CMake project that includes Windows support and also provides the capability of automated integration testing with several input data sets and corresponding reference output. With very limited overhead, this allows us to establish our work as a multi-platform basis for future research such as a qualitative comparison of additional SYCL implementations. In addition, our experience has shown that toolchain and software stack diversity can help in spotting programming errors hidden behind implementation-defined semantics.

C. Algorithmic Changes

Besides implementation-focused code changes, we also performed a number of algorithmic modifications and selections. Note that while the algorithm in terms of the numerical solver involved remains unchanged, there are a number of high-level changes required for efficient GPU parallelism. Since these changes are programming style agnostic and rather relate to parallelism patterns, we consider them as algorithmic changes from an HPC point of view and discuss them in this subsection.

There are multiple implementation variants in CRONOS regarding sub-domain decomposition and iteration order. The original implementation, shown in Fig. 2, iterates over a two-dimensional space and extracts a full pole of data in the third dimension. It then iterates over the individual cells of this pole, performing computations and cell updates in the current direction (e.g. front and back) as required. After finishing all poles in one direction, this process is repeated for the second direction, computing and updating physical quantity contributions between left- and right-facing cells, and finally repeated for the third direction. After such a three-phase step has been completed for all three directions, all cells of this subdomain received contributions from each of their 6 interfaces.

This kind of domain decomposition is efficient since it works on one-dimensional poles which simplifies memory access patterns or optimizations such as vectorization. However, this scheme is not suitable for Celerity, since we aim for automatic decomposition of the global domain without user directions. It would entail submitting three kernels per time step, one per direction, and temporarily saving the computed updates before applying them in a fourth kernel. A second approach would be to distribute these updates throughout the system after every of the three phases, introducing additional communication overhead.

Furthermore, the original implementation of CRONOS follows an input-parallel scheme by computing the flux components between two cells only once and applying the corresponding change to both cells. This is an input-decomposition based strategy and leads to computational dependencies between individual iterations over the domain, which makes it difficult to parallelize them. In the MPI implementation of CRONOS, this is not an issue since each subdomain is computed by a single rank, and hence this computation is entirely sequential. While minimizing computational overhead by computing these components only once per cell pair, it is not suitable for distributed-memory GPU parallelism in Celerity since the computation at a single point in our three-dimensional space does not only update the cell at this point, but also its neighbors within the same kernel. Celerity currently does not support such intra-kernel dependencies and explicitly forbids using range mappers with overlapping data ranges for write access.

For this reason, we change the implementation in two aspects. First, we follow a blocked scheme where we de not iterate over a pole of cells at a time but individual cells. Second, we change the algorithm to an output-decomposition scheme such that the computation of a single position only requires an update operation of the cell at this position and does not imply updates to its neighbors. While this leads to a much higher degree of parallelism, it introduces computational overhead since we need to re-compute flux components between two cells A and B twice, once for updating A and again for updating B. A code excerpt illustrating this overhead will be shown and discussed in Section IV-D.

D. Porting Strategy

CRONOS offers both an MPI-parallel as well as a sequential implementation via preprocessor-conditional compilation. Since Celerity itself is a distributed-memory programming model and not suitable to be added incrementally to already parallel applications (contrary to e.g. OpenMP), we base our Celerity port on the sequential implementation. Since a full time step in CRONOS consists of several sub-steps and components, we add ported versions of these steps and components one after the other, comparing respective outputs via code assertions to ensure semantic correctness. Since these assertions require data transfers from device memory back to host memory for comparison, we disable them for production

```
1  queue.submit([=](celerity::handler& cgh) {
2
3    auto ReadMode = cl::sycl::access::mode::read;
4    auto WriteMode = cl::sycl::access::mode::read_write;
5
6    auto srcAcc = src.get_access<ReadMode>(cgh, celerity::access::neighborhood<3>(2,2,2));
7    auto dstAcc = dst.nom.get_access<WriteMode>(cgh, celerity::access::one_to_one<3>());
8
9    Problem problem = {...};
10
11   cgh.parallel_for<class Compute>(range, cl::sycl::id<3>{3,3,3}, [=](cl::sycl::id<3> item) {
12     int z = item.get(0);
13     int y = item.get(1);
14     int x = item.get(2);
15
16     numValsType numVals[DirMax], numValsX[DirMax], numValsY[DirMax], numValsZ[DirMax];
17
18     compute(srcAcc, numVals, problem, x, y, z);
19
20     compute(srcAcc, numValsX, problem, x + 1, y, z);
21     getChanges(dstAcc, problem, numVals, numValsX, x, y, z, DirX);
22
23     compute(srcAcc, numValsY, problem, x, y + 1, z);
24     getChanges(dstAcc, problem, numVals, numValsY, x, y, z, DirY);
25
26     compute(srcAcc, numValsZ, problem, x, y, z + 1);
27     getChanges(dstAcc, problem, numVals, numValsZ, x, y, z, DirZ);
28
29   });
30 });
```

Listing 1: Simplified code excerpt of the main Celerity computation kernel of CRONOS.

runs due to the overhead introduced by these data transfers over PCIe.

Since Celerity entails comparatively benign code changes when coming from SYCL applications, we first port our application to pure SYCL and only move to Celerity in a second step. This has the advantage of a purely single-node program during a major part of the development phase. While the design goal of Celerity is to alleviate the burden of thinking and programming in distributed memory parallelism or message exchange terms, it is still easier to debug a single process during the development phase. Second, SYCL implementations support a so-called `host device` which allows kernel execution on the CPU instead of an accelerator. This feature greatly improves any debugging experience, since it enables use of all the standard toolchain support that has been established for C/C++ programs throughout the years, such as debuggers and memory checkers (e.g. `gdb` or `valgrind`). After ensuring that we have semantically correct SYCL versions of our components of interest, we move to Celerity.

Listing 1 shows an abbreviated code version of the main Celerity kernel of CRONOS. Lines 1–30 submit a C++ lambda that holds both the actual kernel as well as preface code that sets up accessors to the input (line 6) and output (line 7) buffers. Note that the input buffer uses a `neighborhood<3>(2,2,2)` range mapper, meaning that for any point of computation in our three-dimensional domain, Celerity needs to provide input data from neighbor cells up to a distance of 2 in each of the three dimensions. The output buffer on the other hand is accessed with a `one_to_one<3>()` mapper due to our output-decomposition parallelism scheme. The original code, due to its input-decomposition scheme, would have required a `neighborhood<3>(1,1,1)` range mapper for write access, which is illegal in Celerity because of the resulting overlapping ranges for individual work items.

After establishing the accessors, a few use-case-specific parameters are set up in line 9 and the actual kernel is specified in lines 11–29. It is three-dimensional and starts with an offset (`id<3>(3,3,3)`) due to the ghost cell shift discussed in Section IV-B, whereas the original implementation simply used negative indices as supported by CRONOSNUMLIB and performed offset computations on the fly.

Finally, the actual computation happens in outlined compute calls in lines 18, 20, 23, and 26. Each of these calls performs point reconstruction, transformation to characteristic variables and the computation of fluxes as described in Section III-B. These individual functions are adopted from the original implementation, with minor to no code changes beyond the aspects detailed in Section IV-B. This lack of effort illustrates the benefits of adopting a programming model and host langauge close to that of the original implementation. Afterwards, the current cell is updated with the computed changes (lines 21, 24, and 27). While the original implementation held the same number of `getChanges()` calls, our output-decomposition-based parallelization strategy duplicates `compute()` calls, leading to lower absolute performance than possible. Among future work is to investigate the minimum set of computations required, as well as other strategies

TABLE I: Experimental platform description.

Nodes	GPUs per Node	CPUs per Node	RAM	Network	Compiler	SYCL impl.	Backend	MPI impl.
4	4x RTX 2070S	1x AMD 2950X	128 GB	10 GbE	llvm 11.0.1	hipSYCL 0.9.1	CUDA 10.1	OpenMPI 4.0.1

to minimize re-computation of intermediate results without affecting the numerical properties of the algorithm or introducing computational dependencies.

V. Performance Evaluation

In this section, we evaluate the performance of our Celerity port of CRONOS, discussing relevant hardware and software characteristics in detail when necessary.

A. Experiment Setup and Methodology

Our experimental target hardware platform consists of four nodes equipped with x86 CPUs and a total of 16 NVidia RTX 2070 Super 8 GB GPUs. Additional hardware details are listed in Table I. While the RTX 2070 Super is an affordable consumer model, its double-precision floating-point peak performance (approximately 283 GFLOPS) is considerably slower compared to single-precision (approximately 9.1 TFLOPS). Nevertheless, the numerical properties of CRONOS require double precision for major parts of the computation due to the necessary conservation of energy and impulse quantities. However, benchmarking on multiple consumer model GPUs still allows us to get a grasp on the scalability of our Celerity implementation, even though performance reductions must be taken into account.

Beyond hardware information, Table I also lists the most important software stack implementations and versions in use. Celerity itself is built from git commit `ae66f22`[1]. Compilation optimizations are enabled via `-O2` and we have hipSYCL use its CUDA backend to target GPU architecture `sm_75`.

We run a simple shock-tube test as our use case. Our stencil kernel is run once in a warm-up phase in order to eliminate measurement perturbation caused by the initial data transfer from host to device memory via PCIe. Afterwards, we run 10 iterations of this stencil operation and measure the time it takes for them to complete. Note that this measurement does not include the transfer back to the host memory since it is only required for file output. This output is triggered only sporadically by CRONOS and could be performed at least partially asynchronously.

In order to find the problem size to conduct our experiments with, we evaluate increasing domain sizes between 16^3 and 320^3 on a single GPU and observe the throughput in cell updates per second. Our expectation is that increasing the problem size reduces the relative amount of any runtime system overhead, thereby maximizing performance. The upper limit of 320^3 was chosen since it requires approximately 6.6 GB of our 8 GB of device memory and we intend to examine strong scaling behavior—larger problem sizes cannot be run reliably on a single GPU. Fig. 3 shows the results of these experiments, confirming our hypothesis that increasing the problem size increases performance, although at a slowing pace. Nevertheless the peak is reached at 320^3 with approximately 1.3×10^7 cell updates per second. Hence we choose this problem size for our scalability experiments. Furthermore, we run our experiments 10 times for every number of GPUs and report arithmetic mean and confidence interval data.

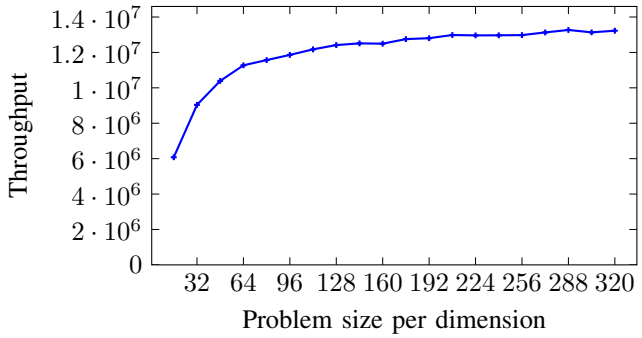

Fig. 3: Cell updates per second on a single GPU for cubic domains with a per-dimension length of multiples of 16.

B. Results

First, we examine the performance and scalability of the original implementation. Table II lists average wall times and parallel efficiency for increasing core counts on the host CPUs. The data shows that the scalability is very high, attributed to the efficient domain decomposition scheme and ghost cell exchange. This is expected, since the code has been shown to scale up to much larger core counts, as detailed in Section III-B. Parallel efficiency even seems to slightly increase again after 16 cores. This could be explained by the fact that the nodes of our target hardware platform are single-socket 16-core CPUs. Hence, when moving from 16 to 32 cores, we double all off-core entities, especially both the number of memory controllers available to the application, as well as the amount of L3 cache (from 32 MB to 64 MB), and again for 64 cores (128 MB). This is also illustrated in Fig. 4, which shows application throughput in number of cell updates per second per core and also indicates a throughput increase after 16 cores.

The results of our Celerity implementation are listed in Table III and illustrated in Fig. 5. First, it is evident that the parallel efficiency is very high, at around 99%, for up to 4 GPUs. Since up to 4 GPUs we are still within a single-node system, all communication happens via PCIe, which has much higher bandwidth and lower latency compared to the 10 Gigabit Ethernet inter-node interconnect. However, even with the full system of 4 nodes and 16 GPUs total, efficiency is still high at 87%.

[1] https://git.io/JOnAi

TABLE II: Average MPI wall times in seconds over 10 runs with 95%-confidence intervals, along with parallel efficiency.

Cores	Wall time [s]	Efficiency
1	$179.14 \pm 5.67 \times 10^{-1}$	1,00
2	$94.88 \pm 6.61 \times 10^{0}$	0,94
4	$48.72 \pm 5.93 \times 10^{0}$	0,92
8	$22.55 \pm 4.44 \times 10^{0}$	0,99
16	$11.96 \pm 2.72 \times 10^{0}$	0,94
32	$5.83 \pm 1.91 \times 10^{0}$	0,96
64	$2.80 \pm 5.80 \times 10^{-1}$	1,00

TABLE III: Average Celerity wall times in seconds over 10 runs with 95%-confidence intervals, along with parallel efficiency.

GPUs	Wall time [s]	Efficiency
1	$24.77 \pm 1.55 \times 10^{-3}$	1.00
2	$12.49 \pm 6.91 \times 10^{-4}$	0.99
4	$6.28 \pm 8.12 \times 10^{-4}$	0.99
8	$3.35 \pm 1.08 \times 10^{-3}$	0.93
16	$1.78 \pm 3.77 \times 10^{-4}$	0.87

Finally, comparing throughput numbers between MPI on the CPUs and Celerity on the GPUs, it can be observed that 8 cores of a 16-core AMD Threadripper 2950X CPU achieve roughly the same application throughput as an NVidia RTX 2070 Super in double-precision. This matches our expectations, as the 2950X offers 8 double-precision floating-point operations per second per core and clocks at approximately 4 GHz. The resulting peak performance of around 500-550 GFLOPS is roughly twice that of an RTX 2070 Super. Naturally, these approximate numbers do not allow us to draw conclusions about the specific efficiency of our application with respect to peak floating-point performance on either the CPU or the GPU. However, they seem to indicate that our application is running at comparable efficiency on both architectures.

VI. Conclusion

In this work we have shown a case study of porting a real-world application to GPU clusters by using a modern, high-level programming model, Celerity. We have discussed the numerical and algorithmic characteristics of the application

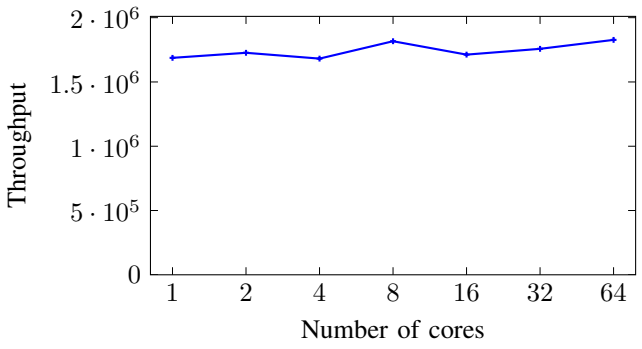

Fig. 4: Performance of the original MPI implementation in cell updates per second per CPU core.

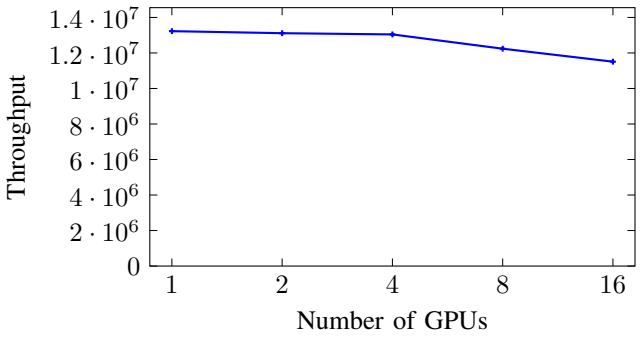

Fig. 5: Performance of the Celerity implementation in cell updates per second per GPU.

as well as its intricacies in parallelization and suitability for GPUs. In addition, the considerations to be taken when porting larger legacy applications to modern platforms and our motivational basis for taking design and implementation decisions have been presented. Results indicate that within the scope of our experiments, an efficiency of 87% can be obtained even without hardware architecture-specific optimizations.

Future work includes additional code coverage by porting more CRONOS components to Celerity and a more detailed performance analysis on multiple hardware platforms that offer higher double-precision floating-point performance. Furthermore, pending optimizations such as partially switching from double-precision to single-precision where possible will be explored.

Acknowledgment

This work is supported by the D-A-CH project CELERITY, funded by DFG project CO1544/1-1 and FWF project I3388.

References

[1] TOP500, "TOP500," https://www.top500.org/, 2021.
[2] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-level c++ for accelerator clusters," in *European Conference on Parallel Processing*. Springer, 2019, pp. 291–303.
[3] R. Kissmann, J. Kleimann, B. Krebl, and T. Wiengarten, "The CRONOS Code for Astrophysical Magnetohydrodynamics," ApJS, vol. 236, p. 53, Jun. 2018.
[4] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.
[5] OpenMP Architecture Review Board, "OpenMP Application Program Interface - Version 4.0," https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, 2013.
[6] Khronos OpenCL Working Group, "SYCL Specification 1.2.1," Khronos OpenCL Working Group, Tech. Rep., 2017.
[7] D. Huber, R. Kissmann, and O. Reimer, "Relativistic fluid modelling of gamma-ray binaries. II. Application to LS 5039," *arXiv e-prints*, p. arXiv:2103.00995, Mar. 2021.
[8] R. Kissmann, J. Kleimann, H. Fichtner, and R. Grauer, "Local turbulence simulations for the multiphase ISM," MNRAS, vol. 391, pp. 1577–1588, Dec. 2008.
[9] M. Flaig, W. Kley, and R. Kissmann, "Vertical structure and turbulent saturation level in fully radiative protoplanetary disc models," MNRAS, vol. 409, pp. 1297–1306, Dec. 2010.
[10] T. Wiengarten, J. Kleimann, H. Fichtner, P. Kühl, A. Kopp, B. Heber, and R. Kissmann, "Cosmic Ray Transport in Heliospheric Magnetic Structures. I. Modeling Background Solar Wind Using the CRONOS Magnetohydrodynamic Code," ApJ, vol. 788, p. 80, Jun. 2014.

[11] T. Wiengarten, J. Kleimann, H. Fichtner, R. Cameron, J. Jiang, R. Kissmann, and K. Scherer, "MHD simulation of the inner-heliospheric magnetic field," *Journal of Geophysical Research (Space Physics)*, vol. 118, pp. 29–44, Jan. 2013.

[12] R. Kissmann, K. Reitberger, O. Reimer, A. Reimer, and E. Grimaldo, "Colliding-wind Binaries with Strong Magnetic Fields," ApJ, vol. 831, p. 121, Dec. 2016.

[13] K. Reitberger, R. Kissmann, A. Reimer, O. Reimer, and G. Dubus, "High-energy Particle Transport in Three-dimensional Hydrodynamic Models of Colliding-wind Binaries," ApJ, vol. 782, p. 96, Feb. 2014.

[14] D. Huber, R. Kissmann, A. Reimer, and O. Reimer, "Relativistic fluid modelling of gamma-ray binaries. I. The model," A&A, vol. 646, p. A91, Feb. 2021.

[15] K. Reitberger, R. Kissmann, A. Reimer, and O. Reimer, "3D magneto-hydrodynamic models of non-thermal photon emission in the binary system γ^2 Velorum," ApJ, vol. 847, p. 40, Sep. 2017.

[16] S. K. Godunov, "Finite difference method for numerical computation of discontinuous solution of the equations of fluid dynamics," *Mat. Sb.*, vol. 47, p. 271, 1959.

[17] C.-W. Shu, "Total-variation-diminishing time discretizations," *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 6, pp. 1073–1084, 1988.

[18] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes, II," J. Chem. Phys., vol. 83, pp. 32–78, 1989.

[19] R. Courant, K. Friedrichs, and H. Lewy, "Über die partiellen Differenzengleichungen der mathematischen Physik," *Math. Ann.*, vol. 100, pp. 32–74, 1928.

[20] PRACE, "HPC Systems," https://prace-ri.eu/hpc-access/hpc-systems/#GENCI, 2021.