An Asynchronous Dataflow-Driven Execution Model For Distributed Accelerator Computing

Philip Salzmann†, Fabian Knorr†, Peter Thoman†,
Philipp Gschwandtner†, Biagio Cosenza‡ and Thomas Fahringer†
†Distributed and Parallel Systems Group
University of Innsbruck, Austria
Email: first.last@uibk.ac.at
‡University of Salerno, Italy
Email: bcosenza@unisa.it

***Abstract*—While domain-specific HPC software packages continue to thrive and are vital to many scientific communities, a general purpose high-productivity GPU cluster programming model that facilitates experimentation for non-experts remains elusive.**

We demonstrate how Celerity, a high-level C++ programming model for distributed accelerator computing based on the open SYCL standard, allows for the quick development of – and experimentation with – distributed applications. To achieve scalability on large machines, we replace Celerity's existing master/worker scheduling model with a fully distributed scheme that reduces the worst-case scheduling complexity from quadratic to linear while maintaining the existing programming interface. We then show how this declarative, data-flow based API paired with a point-to-point communication model with *eager data pushing* can effectively expose and leverage opportunities for latency hiding and computation/communication overlapping with minimal or no manual guidance. We demonstrate how Celerity exhibits very good scalability on multiple benchmarks from several scientific domains and up to 128 GPUs.

***Index Terms*—Accelerator Computing, GPGPU, Cluster Computing, Runtime System, SYCL**

I. Introduction

High performance computing (HPC) is a complicated endeavor: Memory hierarchies, heterogeneous architectures, network topologies and work and data distribution introduce levels of complexity that are hard to manage even for experts. Yet, many modern fields of research either heavily depend on, or are outright built upon the ability to perform number crunching at extreme scales.

Development of new software for HPC is typically left to the select few, while cutting edge research is often performed using well-known domain specific software packages, such as GROMACS [1] for biomolecular simulations.

Those that are not lucky enough to find their domain having a well-maintained and highly optimized software package are frequently forced to use legacy application codes which have been passed down through generations of researchers. These codes, after years of organic growth, are often difficult to extend and experiment with when trying to target novel hardware.

These problems are only compounded as we are on the verge of entering the "Exascale Era", where even larger cluster configurations consisting of *fat nodes* containing multiple accelerators are likely to further complicate programmability.

Today's *de-facto* standard approach to developing distributed HPC applications is still "MPI + X", where the Message Passing Interface (MPI) [2] is combined with a data parallel programming model such as OpenMP, CUDA or OpenCL. While this approach is certainly viable and can be used to produce programs achieving peak performance, it can be likened to how assembly is still used nowadays: Once all tradeoffs and performance characteristics are known, and when targeting specific hardware systems for execution, with a high degree of expertise, peak performance can be achieved. However, as a consequence, the experimentation phase becomes increasingly inflexible over time, as for example changing the distribution of work and data in a hand-written MPI + X application can be extremely labor intensive.

Meanwhile, there has been no shortage of research projects and software frameworks to facilitate distributed accelerator computing. We can roughly group them into three categories:

Several early works have attempted to bring OpenCL or CUDA to distributed environments by abstracting remote devices into a local-looking API [3], [4], [5]. While simplifying the interaction with remote devices across clusters, projects of this first category typically still require the manual assignment of work to remote devices, ultimately leaving complex decisions about scheduling and data distribution to the user.

The second category comprises full-blown runtime systems, which typically introduce a broad API and custom terminology, as well as enabling ecosystems of tooling and derived software projects. These projects offer great flexibility and impressive performance results, at the cost of having to adopt a new API, and rely on the sustainability of their associated development ecosystems.

A notable example is StarPU [6], an extensible runtime system for programming heterogeneous systems. It offers a wide array of scheduling approaches, from simple FCFS policies, over work-stealing and heuristics based on HEFT [7], to dedicated schedulers for dense linear algebra on heterogeneous architectures [8], [9]. Nevertheless, StarPU's C API is rather low level and requires the explicit handling of data distribution when executing in cluster environments.

Legion [10] is a runtime system designed to make efficient use of heterogeneous hardware through highly configurable and efficient work splitting and mapping to the available resources. Its C++API is intricate and precise, with the explicit intent of putting performance first, before any programmability considerations, making it unsuitable for non-expert users.

Another notable entry in the second category is PaRSEC [11], which uses a custom graph representation language called JDF to describe the dataflow of an application [12]. Either automatically generated through a custom compilation step, or written by hand, this representation then enables a fully decentralized scheduling model and automatic handling of data dependencies across a distributed system. The initial distribution of data needs to be provided by the user.

The Kokkos Programming Model [13] is a popular library-based framework for programming different CPU and GPU architectures. It promises performance portability through a set of abstractions around both parallel execution and data structures. While not aimed at distributed computing directly, the experimental *remote spaces* extension adds multi-node capabilities to the project.

The third category comprises those projects that extend existing programming languages, for example the pragma-based OmpSs [14], or introduce entirely new languages altogether, such as Chapel [15] or Regent [16]. While these projects can greatly simplify working with parallel and distributed systems, by building the appropriate semantics directly into a language, the often limited tooling support (IDEs, debugging) can hamper their wide-spread adoption.

A common theme across many of the aforementioned works is that they rely on *dataflow* information to infer dependency relationships between tasks. Dataflow graphs have proven to be a good abstraction for distributed computing at scale, as demonstrated for example by TensorFlow [17] for machine learning applications. Here, general purpose programming models with user-defined kernels face the additional challenge of determining *how* data flows through the application. While coarse-grained dataflow information can be obtained by means of replacing raw pointer arithmetic with *accessor* constructs, for example as is done in SYCL [18], enabling efficient execution in distributed memory contexts requires more fine-grained (i.e., on the level of a single data element) information. Some works have explored the possibility of inferring fine-grained data access patterns automatically using compiler-based techniques [19], [20], [21]. Challenges faced by these approaches include that detectable patterns are constrained to certain affine functions, while keeping a compiler component up to date with evolving language standards is a maintenance burden that is difficult to sustain. A pure library-level approach is to rely on users to augment tasks with data access annotations [22], [23], which is also the approach taken by this work (which will be discussed in Section II-B).

In order to fuel future research and innovation at scale, ways to quickly develop distributed applications and efficiently experiment with different work and data distribution patterns, in a way that is manageable by non-experts, are required. We

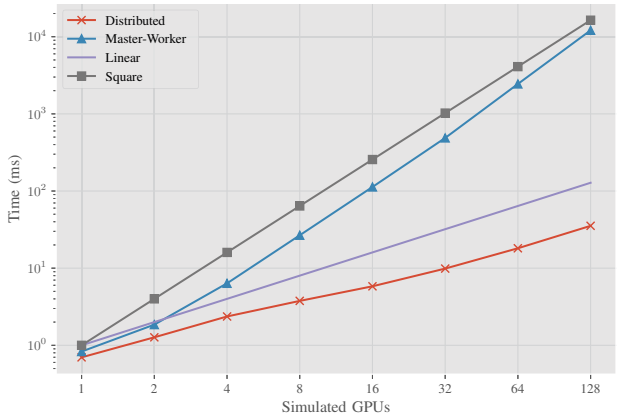

Fig. 1. *Dry run* comparison of master/worker versus distributed scheduling model performance for up to 128 GPUs. Reported times are for 100 simulated time steps of a task with an *allgather* in between each iteration.

contrast the aforementioned works with Celerity, a high-level programming model focused on programmability, ease of use and experimentation. Celerity's approach differs to existing solutions by the following main points:

1) Celerity is designed for execution on distributed memory GPU clusters from the ground up. No special integration with a communications layer such as MPI or an accelerator API such as CUDA is required.
2) It compiles as plain C++, no language extensions are introduced, facilitating the use of existing tools.
3) It is designed as a thin layer on top of SYCL [18], an industry standard for accelerator programming, allowing for existing SYCL applications to be easily converted to Celerity — and back.

The Celerity API was first proposed by Thoman et al. in [24]. Thereafter, [25] introduced a proof-of-concept implementation focusing on the programmability benefits of Celerity compared to plain MPI + X. While its API surface has remained relatively stable ever since, apart from adopting changes introduced in SYCL 2020 [18], the runtime implementation has undergone several iterations before settling on the current design.

None of this prior work focused on the design and implementation of the Celerity runtime system, and as such we will give a comprehensive overview in Section III. The most significant change to date is the focus of Section IV: The initial prototype implementation used in [25] relied on a master–worker scheduling model, where a centralized node – the master node – is responsible for the generation of *commands* for each worker. While this approach has the advantage of complete knowledge about the distributed system state within a single node, it faces difficulties when scaling to larger numbers of GPUs.

In this work, we propose an alternative, fully distributed scheduling model. Its latency advantage can be seen from the experiment in Fig. 1, where we schedule 100 iterations of a

simple task on up to 128 GPUs. In order to focus solely on the scheduler's performance, no actual kernel launches or data transfers take place (we refer to this as a *dry run*). Between each iteration, the task requires an *allgather* step, that is, data written by each GPU in one time step will be read by every GPU in the next times step. This represents a worst-case scheduling scenario for the centralized model, as for N nodes, N^2 transfers need to be issued. The resulting numbers confirm this, with scheduling for 128 GPUs requiring an excessive 100 milliseconds per iteration. The distributed model on the other hand, which will be explained in detail in Section IV, scales linearly with the number of GPUs.

We summarize the contributions of this article as follows:

- A detailed description of the Celerity runtime system's current design.
- Presenting a novel, fully distributed scheduling scheme replacing the existing master/worker model.
- Introduction of a new *task hints* API that enables users to guide Celerity's task splitting behavior.
- Enabling multi-GPU support for individual workers.
- Experimental evaluation of the runtime system performance for application codes on up to 128 GPUs.

II. The Celerity Programming Model

The Celerity API and runtime system is a modern C++ framework for distributed GPU computing [25]. Built on the open SYCL standard [18] published by the Khronos Group, it aims to bring SYCL to clusters of GPUs with a minimal set of API extensions. While Celerity inherits the ability to support a variety of accelerator devices including FPGAs and ASICs from SYCL, it is currently focused on a GPU-based accelerator use case, which is also the most common scenario in high performance computing.

Before we introduce the Celerity API, let us briefly reiterate the most important concepts of the SYCL programming model: A typical SYCL program is centered around *buffers* of data and *kernels* which manipulate them. Kernels are expressed as plain C++ functors within the host source code, making SYCL a *single-source* programming model. An implementation-defined compilation step is then used to extract the kernel code and translate it into a representation that can be understood by the target hardware platform. As is common in modern accelerator programming models, kernel functions are not called directly within the context of the host application. Instead, kernels are wrapped in so-called *command groups* and submitted to a *queue*, which is then processed asynchronously with respect to the host process. *Events* can be used to query a kernel's execution status and wait for its completion.

One aspect of SYCL crucial to Celerity's design is that data buffers are more than simple pointers returned by a `malloc`-esque API, as for example in CUDA: they are an evolution of OpenCL's memory objects, opaque handles to memory that cannot be manipulated directly. Instead, buffers are accessed through so-called *accessors*, which are declared within a command group before a kernel is launched and provided to the kernel as arguments (either through implicit lambda captures, or explicitly)[1].

Upon creating buffer accessors, the user additionally has to declare *how* a buffer will be accessed, i.e., for reading, writing or both. Not only does this enable certain optimizations, it also allows the SYCL runtime to construct a *task graph* based on the dataflow of buffers through kernels: If a kernel reads from a buffer that is written by an earlier kernel, it implicitly depends on said earlier kernel to have finished its execution. Celerity uses and builds upon this mechanism to allow for tasks to be executed across a distributed cluster of nodes, while ensuring data coherence through implicit asynchronous transfers.

From a high-level perspective, the Celerity API design can be considered a natural evolution and generalization of an overarching trend in languages and APIs for highly parallel architectures. Several successful modern programming models such as CUDA, OpenCL and SYCL abstract the concept of a hardware thread in a way that lets users express their programs in terms of linear looking kernels, which are invoked over an N-dimensional range of work items. Celerity attempts to abstract the concept of distributed computation in a similar way: kernels are written in the same way as in SYCL, however they can be executed across multiple devices, with all resulting data transfers handled completely transparently to the user.

The Celerity API is thus designed around three levels of parallelism:

1) Task-parallelism at the outer level: Tasks without mutual dependencies can be executed concurrently.
2) Per-worker parallelism: The execution of a single task can potentially be split across many different workers.
3) Data-parallelism within a task.

As previously mentioned, the Celerity API tries to stay as close as possible to SYCL. The most notable and fundamental extension to SYCL introduced by Celerity is the concept of *range mappers*, functions that provide additional information about how buffers are being accessed from within a kernel. From user-provided range mappers, the Celerity runtime system can infer, which parts of a buffer will be read, and which ones will be written – at arbitrary granularity. This information is in turn used to make scheduling decisions, and to ensure that all data dependencies are satisfied before a particular task is executed. Listing 1 shows an example of a simple matrix addition implemented in Celerity.

A. Tasks

As mentioned previously, to transparently enable asynchronous execution, all compute operations in a Celerity program are invoked by means of a queue object. In the second line of Listing 1, this queue of type `celerity::distr_queue` is created. Subsequently, three two-dimensional buffer objects are created, two of which are

[1]SYCL has recently introduced additional lower-level APIs for pointer-based memory management which do not convey dependency information to the runtime and are thus omitted from Celerity.

Listing 1 Computing a matrix addition in Celerity.

```
using namespace celerity;
distr_queue queue;

auto rg = range<2>(512, 512);
buffer<float, 2> buf_a(hst_a.data(), rg);
buffer<float, 2> buf_b(hst_b.data(), rg);
buffer<float, 2> buf_c(rg);

queue.submit([=](handler& cgh) {
  auto o2o = access::one_to_one{};
  accessor a{buf_a, cgh, o2o, read_only};
  accessor b{buf_b, cgh, o2o, read_only};
  accessor c{buf_c, cgh, o2o, write_only};
  cgh.parallel_for(rg, [=](item<2> itm) {
    c[itm] = a[itm] + b[itm];
  });
});
```

initialized from some host data (not included in the code snippet).

The central call to `distr_queue::submit` creates a new *task*, which will later be scheduled onto one or more GPUs across the given cluster. Celerity offers different types of tasks for different activities, with the most common one being a device kernel execution. Other types of tasks can be used for example to perform host-side computations, I/O operations, or to interface with third-party libraries. Most Celerity tasks can be executed over a 1-, 2- or 3-dimensional index space. This space is then split into multiple *chunks* that can be executed by different workers. In Listing 1, a two-dimensional device kernel is submitted through the call to `parallel_for`. The provided callback (the kernel code) is subsequently invoked with an index object (`itm`) of corresponding dimensionality, which is used to uniquely identify each kernel thread.

B. Range Mappers

Except for namespace changes, this program closely resembles a canonical SYCL program, with one important difference: Each constructor for `celerity::accessor` is provided with a *range mapper*, in this case a two-dimensional instance of the `one_to_one` mapper.

This particular range mapper informs the runtime that every work item (thread) of the 512×512 global iteration space accesses exactly one element from `buf_a`, `buf_b` and `buf_c` each — precisely at the index of the work item, which is provided to the kernel through the `itm` parameter.

In general, range mappers can be (almost) arbitrary functions, allowing for a high degree of flexibility. Celerity provides several built-in range mappers for common access patterns (such as *neighborhoods* for stencil codes), while users may implement custom range mappers for specific use cases.

There is, however, one limitation on how accessors can be associated with buffers: Celerity currently only supports output partitioning. From a theoretical point of view (in practice, custom acceleration data structures are employed), the runtime system has to track the state of each individual element, including in particular its *last writer*, in order to build a data dependence graph and construct the necessary transfer operations. As such, to maintain a consistent global view, only one accessor may write to a given element of an output buffer at a time.

C. Summary

While Celerity can be considered a *task-based runtime system*, its default mode of operation differs significantly from the more common approach taken, particularly in distributed memory settings. Instead of leaving the choice of how to split a computation fully to the user (by submitting, for example, one task for each block in a matrix multiplication), the Celerity approach is to consider the entire computation as a single *splittable* task.

The rest of this paper focuses on the runtime design, distributed scheduling and their performance characteristics. For more information on the Celerity API, we refer the reader to [25], [26].

III. Runtime Design

The Celerity runtime system is built on top of SYCL and MPI and runs in concert with a user provided program. While the underlying SYCL runtime is responsible for interfacing with the compute device, Celerity manages the distribution of work and data throughout a cluster. We often refer to individual Celerity processes as *worker nodes*, where a worker corresponds to an MPI *rank*. While in the earlier implementation presented in [25] it was required to spawn a separate worker for each GPU, in its current form, a single worker can manage all available GPUs on a host.

As discussed in the previous section, the Celerity API closely resembles SYCL, in that it is centered around buffers and asynchronous tasks that perform computations on them. However unlike in SYCL — and most other task-based runtime systems for that matter — a task may optionally be executed across more than a single device.

A visual overview of the runtime is given in Fig. 2: After first being submitted, a task is analyzed regarding its requirements and incorporated into a *task graph* for later execution. Each task is then asynchronously processed into a set of *commands* that describe concrete operations to be performed by individual worker nodes. Commands may instruct a worker to execute parts of a task, or to transfer data to another worker, for example. Given a set of commands corresponding to a particular task, each worker is then individually responsible for deciding when to execute these commands, based on the execution status of preceding commands, and the availability of required data.

A. Asynchronous Execution

The entire Celerity runtime system is designed with a strong focus on asynchronicity. For this reason, a task is not directly executed once it has been submitted. Instead, the call to `distr_queue::submit` returns immediately after insertion into the task graph, and the execution flow of the user program continues. The *command function* provided as part of a submitted task is then later asynchronously invoked,

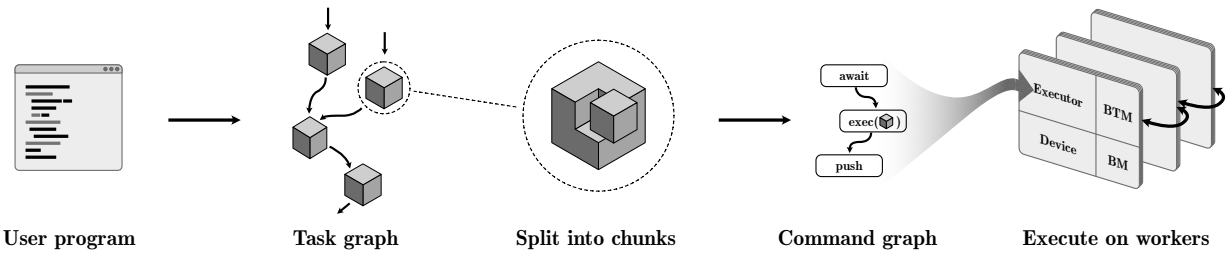

Fig. 2. Overview of the Celerity runtime: A user program submits tasks which are analyzed for their dataflow and inserted into the high-level task graph. They are then split into executable chunks and assigned to worker nodes. Data transfer commands are inserted between the execution of chunks to ensure coherence across the cluster. The per worker executor loop interfaces with the compute device, buffer manager and buffer transfer manager to run kernels and exchange data with other nodes. All stages are executed concurrently as part of the asynchronous application pipeline.

once the time comes to actually perform computational work. All Celerity workers spawned for a particular run execute the same user program in *single program multiple data* (SPMD) fashion, not unlike a typical MPI program. Upon submission, tasks are analyzed regarding their requirements: Which buffers they access and how, as well as what type of operation they contain (for example, running a GPU kernel). At a later point in time, once all data requirements of a task (or more precisely, *parts* of a task) have been satisfied, the command function provided as the inner C++lambda function (lines 14 through 16 in in Listing 1) is executed. There are two main benefits to this execution model:

1) By having all worker nodes execute the exact same user program, no explicit communication about the distributed primitives such as tasks and buffers involved in a computation is necessary. Each node deterministically assigns unique IDs to resources the first time they are encountered. This requires that the control flow does not diverge between workers.
2) If a task were to be executed right away, each task submission would act as a global synchronization point, in which the runtime system has to determine where to execute which parts of the task, as well as the set of data transfers that would be required to do so. This would make the entire execution model much more latency sensitive, and would prevent many of the benefits emerging from asynchronous execution that will be discussed in later sections.

Furthermore, by keeping task submission a lightweight operation, the runtime is enabled to "look ahead" within the user program. We plan to explore using this property for more informed scheduling decision in future work.

B. Task Graph Construction

The runtime uses the information obtained by analyzing tasks upon their submission to build a *task graph*, a directed acylic graph (DAG) that encodes the dependency relationships between tasks. Importantly, due to the additional information provided by the range mappers, the task graph is — unlike in SYCL — not constructed based on buffer accesses alone, but also considers the accessed buffer *locations*. This means that two or more tasks that operate on the same buffer, but in different regions, may be executed concurrently.

As the task graph is being constructed, the set of currently eligible tasks (that is, the set of tasks that have all their dependencies satisfied) is processed further, by means of *scheduling* them. Selecting the next task for scheduling currently follows a simple first-come-first-serve policy, however more sophisticated heuristics could be implemented here, which we consider for future research.

The selected task is then converted from an abstract description of an execution into a set of concrete commands that instruct individual Celerity workers on what to do. This step includes deciding on whether and how to split a task into *chunks*. The default behavior of Celerity is to split each task into one chunk per GPU.

The runtime then creates one *execution command* for each chunk. These commands include information about what kind of operation to perform (e.g., running a device kernel or invoking a host-side callback), and over which part of the global index space of the corresponding task they operate. Commands are again, based on their data requirements, generated as part of a DAG, the *command graph*, which also includes commands for data transfers and synchronization, among other things [26]. Generation of commands is at the core of Celerity's scheduling loop, which will be explained in Section IV, where we detail how we replaced Celerity's centralized master/worker model with a fully distributed scheme.

C. Per-Worker Executor

Once a command for a particular worker has been generated, it enters said worker's *executor loop*. This loop is responsible for all task executions as well as data transfers on a given node. Upon entering the loop, commands become *jobs*, lightweight wrappers around asynchronous operations such as data transfers and kernel executions, which can be cheaply polled for their completion status. The entire executor loop is built around the idea of maintaining cheap handles to potentially expensive operations that concurrently execute in other threads or devices. It is thus very important that the executor loop is not stalled by blocking operations for longer

periods of time, as this will impact all other active jobs. The executor imposes a limit on the total number of active jobs, in order to both maintain the performance of the loop itself, as well as not to oversubscribe the available hardware resources (such as GPU cores or network bandwidth). Once a slot becomes available, a choice has to be made which of the *ready jobs* to start next. Currently a simple first-come-first-serve policy is implemented.

Readiness of a job is determined based on its dependencies, i.e., predecessors in the command graph. Upon arrival, a command's predecessors are examined and compared against the list of enqueued and active jobs. For each matching job, its count of *unsatisfied dependencies* is increased by one. Completion of a predecessor decreases the count, and reaching zero means that the command is ready to be executed. If a predecessor cannot be found in the list of enqueued or active jobs, it can be assumed to have already completed (thus not increasing the count). This assumption is valid given that the order of arrival of commands in the executor is a topological ordering of the command graph. The implication of this is that the execution of commands is not constrained by the order of their generation. Instead, the local command graph can be processed dynamically, for example by beginning to push data to a peer as soon as it is available, and not only once the task that prompted the generation of this command is being executed.

D. Buffer Management

In the SYCL programming model, a buffer does not directly correspond to any particular memory allocation. Instead, the runtime can freely migrate buffers between the host and potentially multiple devices, in a completely transparent fashion. However once a computation operating on a SYCL buffer is launched, a full memory allocation is made available to the kernel, and users can access the entire range[2] of the buffer.

For Celerity's distributed execution model a more sophisticated way of managing buffers is required. After all, it is not only wasteful to fully allocate every buffer on every GPU, using up precious device memory for parts of a buffer that may never be touched on a particular worker: such an approach would also prevent Celerity applications from processing more data than can fit in any one GPU.

For this reason, Celerity distinguishes between two types of buffers: *virtual buffers* and *backing buffers*. Virtual buffers correspond to the conceptual idea of a distributed buffer that is created by the user through the `celerity::buffer` class. They can span an essentially arbitrary n-dimensional range (only limited by 64-bit integers), and are used to declare data dependencies for tasks. However, importantly, virtual buffers are nothing more than abstract handles, and do not incur any actual bulk memory allocations on the host or device. Backing buffers on the other hand correspond to the actual data allocations on each worker node. Every Celerity worker maintains multiple backing buffers for each virtual buffer, one on the host, and one for each local device, ensuring coherence between them as needed, by means of fine-grained spatial versioning. Crucially, backing buffers are only as large as needed, spanning an arbitrary contiguous subregion of their corresponding virtual buffer, and are lazily resized on demand.

To the user, the dimensions of a backing buffer are of no concern. Within kernels, buffers are accessed using their virtual coordinates, and the `celerity::accessor` internally performs the necessary coordinate transformations.

A problem that can arise from the lazy resizing of backing buffers is that programs that access buffers in erratic patterns can cause frequent re-allocations and coherence copy operations. While strategies to mitigate this problem are currently in development, a pragmatic and effective method to avoiding this overhead can be to overestimate the range mapper specifications in the early stages of a program, such that later accesses are already contained in the allocated backing buffer.

E. Transferring Buffers

Buffer transfers in Celerity are managed as part of the executor loop. All transfers are implemented using the non-blocking point-to-point MPI routines `MPI_Isend` and `MPI_Imrecv`. In each iteration of the executor loop, the *buffer transfer manager* polls the state of existing MPI transfers as well as newly incoming ones. The current design relies solely on point-to-point communication as it best matches the asynchronous nature of the runtime system. We note that use of MPI collective operations, while representing a synchronization point across all participating worker nodes, can nevertheless allow for certain communication patterns to be implemented more efficiently, and as such their use will be explored in future work.

Buffers are transferred together with a header. The header contains, among other information, a description of which part of the virtual buffer (i.e., an offset and range) is being transmitted.

Transfers are performed on dense copies of the source data, which are created on demand. To avoid frequent re-allocation of these data packages, a pool of pinned host memory is used, which also allows for more efficient asynchronous transfers between GPUs and the host.

Buffer transfers again benefit from Celerity's fine-grained access management. For example, an incoming data transfer can be written into a target buffer even if kernels are operating on that same buffer (but in a different location) at the same time.

IV. Distributed Scheduling

Celerity uses a static scheduling model, where parts of a task are assigned to worker nodes in a predetermined (but configurable) fashion. Nevertheless, the fine-grained dependency information generated as part of the scheduling loop, together with the executor model described in Section III-C allow Celerity applications to effectively leverage automatic computation/communication overlapping.

[2]Certain restrictions may apply, depending on how the corresponding buffer accessor is created.

Unlike in other distributed memory programming models, Celerity does not require the user to settle on a data partitioning a-priori. While this increases usability and scheduling flexibility, it requires the runtime to keep track of data distribution down to the individual buffer element level. In this section, we will describe how Celerity leverages this fine-grained buffer management to turn tasks into commands, and how we ported this mechanism from the centralized master/worker model to a fully distributed implementation.

A. Task Hints

The Celerity programming model operates on a high level of abstraction, trading precise control for convenience, ease of use, and performance portability. In some situations it can however still be beneficial to enable users to provide context-specific semantic information about their program that can help Celerity achieve better performance. This is achieved through the novel *task hints* mechanism, which can be passed to the runtime alongside buffer accessors and kernel submissions inside command group functions:

```
queue.submit([=](celerity::handler& cgh) {
  cgh.hint(celerity::hint::tiled_split{});
  // create accessors, provide kernel code etc.
});
```

For example, hints allow the user to swap out Celerity's default one-dimensional for a 2D *tiled split*, or a split can be *oversubscribed* by generating more than one chunk per GPU, which might enable the runtime system to leverage computation/communication overlapping.

Hints as an API mechanism are beneficial to both the development of Celerity as well as users of Celerity. From the perspective of Celerity runtime developers, hints provide a way of quickly and easily experiment with new functionality, without introducing breaking changes or unnecessarily bloating the core API. If a particular hint's functionality proves useful across different applications, it may influence the development of heuristics to automatically provide their behavior to a larger user base. They also offer a simple deprecation path, with hints that become superseded by heuristics or other mechanisms simply turning into no-ops. From a user's perspective, hints provide a way for incorporating their knowledge about domain specific characteristics of their application, or about the hardware platform into the runtime behavior.

Crucially, these hints – together with the high level of abstraction and consequently semantic knowledge available to the runtime system – allow users to experiment with various data distribution patterns and scheduling approaches without spending a significant amount of time on engineering or maintaining different implementation options. In Section V we show how the combination of two hints can substantially influence the performance of two stencil codes.

B. Turning Tasks into Commands

On a conceptual level, scheduling in Celerity means turning abstract tasks from the task graph (as described in Section III-B) into commands; concrete operations that can be executed by a worker node. Other than the execution of chunks (i.e., parts of a task) themselves, commands most commonly describe data transfers that must be performed in order to ensure all buffer contents are available and up to date before a computation can begin. To express these relationships, commands are again arranged in a directed, acyclic graph.

In the earlier design presented by Thoman et al. [25], the Celerity runtime designated a *master node* that is responsible for the generation of commands for all workers. We will use this approach as a starting point, first describing scheduling from a centralized perspective, and subsequently extend it to a fully distributed scheme.

Let the index space I of an n-dimensional task be a set of points in $\mathbb{I}^n$, over which a parallel computation is to be performed. In practice the index space is constrained to be a dense, rectangular grid with $p \geq 0^n \forall p \in I$ and $1 \leq n \leq 3$. For example, the kernel in Listing 1 spans over a two-dimensional index space containing all points in $[0, 512)^2$. A *split* S of I is then a set of disjoint chunks $C \in S$ such that $\bigcup_{C \in S} C = I$.

A range mapper is a function $f : \mathbb{I}^n \rightarrow \mathbb{I}^m$ that maps an n-dimensional chunk to an m-dimensional set of buffer elements that are accessed (for reading, writing, or both) by that chunk. In practice both sets are again constrained to be dense and of rectangular shape. Importantly, range mappers must be *monotonic*, that is $A \subset B \Rightarrow r(A) \subset r(B)$. We use r and w to denote range mappers for reading and writing buffer accesses, respectively.

Scheduling begins by splitting a task into several chunks, by default one per GPU. Tasks can be split in several different ways, for example along a single dimension (1D split) or into tiles (2D split); this can be controlled by the user. For each chunk a corresponding *execution command* is generated, and assigned to a worker node in a deterministic way. For the scheduling algorithm the shape of a split is not of importance, only how many chunks there are, and to which worker nodes they are assigned.

Given a task t_a with index space I_a that writes to a buffer x with range mapper $w_{x,a}$, we call t_a the *last writer* of the region $w_{x,a}(I_a)$ of buffer x iff there exists no other task[3] t_b with $b > a$, index space I_b and range mapper $w_{x,b}$ such that $w_{x,b}(I_b) \cap w_{x,a}(I_a) \neq \emptyset$.

Given tasks t_a and t_b and an index space I_b, t_b is said to be a *successor* of t_a iff there exists at least one buffer x that is written by t_a and read by t_b, and $X = w_{x,a}(I_a) \cap r_{x,b}(I_b) \neq \emptyset$, where task t_a is the last writer of region X. Conversely, task t_a is then said to be a *predecessor* of t_b. While this property over tasks and index spaces is used to construct the task graph, the same analogously applies to commands and their respective chunks for constructing the command graph.

To produce a valid scheduling of a program, Celerity needs to determine all predecessors for a given command. Note that in practice a command can have many predecessors, as it may access multiple buffers, and each of the accessed buffer regions could again have several last writers. If command c_a

[3]We assume tasks to be numbered according to their submission order.

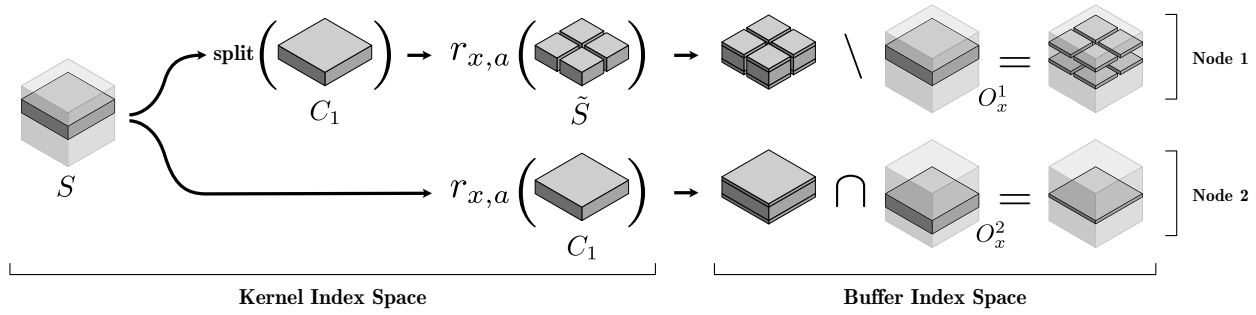

Fig. 3. Processing of chunks in the distributed scheduler: Two worker nodes process the same chunk C_1 as part of the split S of task t_a. Since the chunk is local to node 1, it is further split into $\tilde{S}$, containing four chunks. Node 2 continues to work with the original chunk. Both nodes apply the range mapper $r_{x,a}$ to compute the read requirements of the chunk(s). Node 1 subtracts its owned data O_x^1 to compute which parts need to be received from other nodes, while node 2 intersects with O_x^2 to compute what data – in this case, a plane of ghost cells – needs to be pushed to node 1.

is a predecessor of command c_b, and the commands were assigned to different worker nodes, a data transfer needs to be inserted: On the side of the last writer c_a a `push` command is generated, while on the side of c_b an `await-push` command is generated.

In order to keep track of last writers, Celerity needs to maintain a spatial data structure containing this information for each buffer[4]. After generating a pair of transfer commands, the corresponding buffer's last writer data structure is updated to include the recipient node, reflecting the fact that this region of the buffer is now replicated. This way, future accesses to the same region (or subsets thereof) by the recipient node won't generate redundant transfers (instead, a simple local dependency on the `await-push` command is created).

Since lookups and updates into/of the last writer data structure need to be performed often – usually several times for each chunk – the algorithmic complexity for these operations becomes a concern. The current implementation of Celerity employs a customized R-Tree [27] which yields sufficient performance for real world data access patterns.

To summarize, in the centralized model, the master node not only has to generate commands for all workers, but also precisely keep track of data movement within the distributed system over time, incurring considerable bookkeeping overhead. A worst case scenario for this approach then is an *allgather*-operation across N nodes, which requires the generation of N^2 `push`/`await-push` pairs. Figure 1 demonstrates how for such a pattern, the master/worker scheduling model starts to exhibits scalability issues relatively quickly, with command generation for a single task and 16 GPUs already requiring approximately 1 millisecond, and for 128 GPUs reaching an infeasible 100 milliseconds per task. Clearly, a different approach is required to scale to larger clusters configurations.

[4]In fact, last writers need to be tracked on both the distributed level ("which worker last wrote a particular region?"), as well as on a per-worker level, to enable the generation of *anti-dependencies*, the discussion of which we will omit for brevity.

C. Distributed Command Generation

A straightforward way of moving from a centralized master/worker model to a distributed one would be to replicate the same algorithm as detailed in the previous section across all worker nodes. This way, each node would maintain a complete view of the global data distribution in the system, while only having to generate commands for itself. However in practice, maintaining the last writer data structures makes up for the bulk of the overall time spent on scheduling, and should therefore be avoided as much as possible. The key to enabling this is the fact that for an individual worker, the exact location of buffer data at any given point in time is of little importance. Instead, the only information required is whether the newest data is currently[5] available locally or not.

Given this insight, we modify the scheduling algorithm in the following way: Instead of maintaining per-buffer last writers, each node now only keeps track of its *owned regions*, that is, parts of a buffer that *it* is the last writer of. Each node splits a given task t_a with index space I_a and begins to process each of the resulting chunks, with the assignment of chunks to nodes again following a deterministic but configurable pattern. We distinguish between *local* and *remote* chunks, that is, chunks that are to be executed on the current node, versus those that are executed on other nodes. For a remote chunk C_r, we apply the reading range mappers for each buffer x and check whether the resulting region $X = r_{x,r}(C_r)$ intersects with the owned region O_x. If it does, we generate a `push` command to transfer $X \cap O_x$ to the node that is executing the chunk. For a local chunk C_l we generate an execution command and again apply all range mappers to find the required buffer regions. For owned regions, we simply generate a dependency onto the last writer. For all other regions, we generate `await-push` commands. Unlike in the centralized model however, we don't await incoming data from a particular node (as we don't know which node currently owns said data). Instead, `await-push` commands

[5]Here, *currently* refers to an anticipated state of the distributed system within the context of the command graph.

now simply express the fact that a certain data region of a particular buffer is expected, but not from where. In practice this means that multiple incoming transfers can be required to satisfy a single `await-push` job. To prevent mixups between transfers across different tasks, `push` commands are matched with `await-push` commands using a unique *transfer ID* that is generated for each combination of task ID and chunk. This is unambiguous as all nodes process chunks in the same order.

After processing all local and remote chunks of task t_a, the final step is to update the locally owned regions. To this end, for each buffer y we compute the new owned region as $O'_y = (O_y \setminus w_{y,a}(I_a)) \cup \bigcup_{C_l} w_{y,a}(C_l)$, that is, we remove parts written by remote chunks of task t_a, while adding parts written by local chunks. The same mechanism is used to determine whether data that has been received from other nodes is still valid or needs to be received again.

Finally, to prevent the generation of redundant `push` commands, each node also maintains information on which of its owned buffer regions have been replicated, and to which nodes. In practice this is implemented as a collection of bit sets stored within an R-Tree, with an active bit signifying the data having been replicated to a particular node.

Given this scheme, the amount of scheduling work performed on each node is drastically reduced, as only locally available data regions need to be tracked. This can be seen in Fig. 1, where the complexity of scheduling the worst-case allgather-operation is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

D. Multi-GPU and Oversubscription Support

The master/worker design does not lend itself well to support multiple GPUs on a single worker: Either the master node would have to generate commands for each individual device, while keeping track of data movements on a per-worker level, thus increasing scheduling complexity. Alternatively, workers could be required to further split the commands they receive, recomputing their dependencies and distributing them across local devices, thus substantially increasing complexity on the workers. For this reason, in the centralized model, a separate worker process needs to be spawned for each GPU in a system, which can lead to unnecessary intra-node data transfers and wastes host-side memory with duplicate data, among other problems.

The distributed scheduling model on the other hand can easily be extended to support multiple GPUs on a single worker: Thanks to the range mapper's monotonicity requirement, local chunks can be split further arbitrarily, without affecting the data requirements as seen from the outside by another node. We therefore extend the algorithm from the previous section to recursively split local chunks into one chunk per GPU. In fact, we can generate even more chunks than that, thus *oversubscribing* each GPU with multiple commands, which can enable computation/communication overlapping for certain applications and data access patterns. While each of these chunks can generate separate `await-push` commands, their respective transfer IDs are still generated based on the original chunk, which again matches what is seen from the outside. Figure 3 illustrates how the same chunk is handled differently by two worker nodes.

Notably, supporting multiple GPUs within a worker node only affects the number of generated chunks, while owned buffer regions are still tracked on a per node level, which further reduces the scheduling cost. Coherence between device buffers is then instead ensured during actual execution, with the buffer manager (described in Section III-D) keeping track of last-writers as commands are being processed, issuing device-to-device transfers as appropriate.

We currently generate one `push` command for each last writer. As explained previously, from the recipients perspective it does not matter how many incoming transfers there are, but from a performance standpoint it may seem preferable to consolidate multiple pushes into one to reduce the overall messages transmitted in the distributed system. However, there are two advantages to this approach: On the sender side, a single consolidated `push` command represents a partial synchronization point in the execution of the local commands; only once all participating writers have finished, the transfer can start. Fine-granular pushes on the other hand can start immediately once their data is ready. Conversely, on the receiving side, execution commands can potentially start early instead of having to wait until data required by other chunks has arrived. This is particularly important for oversubscription, where these fine-granular transfers are essential to enabling computation/communication overlapping.

We note that the choice of how many commands to generate represents an interesting tradeoff, and sending data based on last writers is only one possible heuristic, with other patterns left to explore in future work.

V. Experimental Evaluation

We present experimental weak scaling results for five different application benchmarks, obtained for up to 128 GPUs on the Marconi-100 supercomputer at CINECA in Bologna, Italy, which places 24th in the TOP500 at the time of writing[6] Table I lists the configuration of a single cluster node and our software setup. The following applications were benchmarked, representing 4 common HPC dwarf classes [28]:

- **Black-Scholes** from FinanceBench [29] and HeCBench [30] computes European option pricing using the Black-Scholes-Merton process using single precision floating point values. The total number of options computed for each configuration is $N_{\texttt{GPU}} \times 4.5 \cdot 10^8$.
- **N-Body** performs a distributed all-pairs N-body simulation. From a distributed memory benchmarking perspective, the most critical aspect of this benchmark is that an all-to-all update of the body states is necessary in each time step, before the subsequent all-pairs update. The number of simulated bodies for each configuration is $\approx \sqrt{N_{\texttt{GPU}}} \times 5.24 \cdot 10^5$. Note that, in order to achieve weak compute workload scaling, the ratio of memory access to compute and transfer operations must by necessity change

[6] https://www.top500.org/system/179845/

across the experiment, as a linear growth in compute necessitates a square-root growth in simulated bodies.

- **Unstructured Mesh** This benchmark performs a finite element simulation on an unstructured domain of elements, with per-element connectivity information. The domain is pre-split into an appropriate number of subdomains depending on the degree of (distributed memory node) parallelism. The number of elements for each configuration is $\approx N_{\mathtt{GPU}} \times 9.44 \cdot 10^6$. As is common with unstructured mesh simulations, the single-GPU performance of this benchmark is primarily limited by the available memory bandwidth, as the arithmetic intensity is below what would be required to saturate the GPU compute units. A single update operation on the mesh also features no significant temporal or spatial data reuse, and therefore caches are not particularly effective.
- **WaveSim** simulates the 2D wave equation over a series of time steps, using finite differences to approximate the solution to the differential equation at each time step. It is implemented as a classic 5-point stencil code operating on two buffers representing the simulated domain at time steps t and t_{-1}. The side length of the simulated domain for each configuration is $\approx \sqrt{N_{\mathtt{GPU}}} \times 2.45 \cdot 10^4$.
- **Cahn-Hilliard** from HecBench [30] and [31] is a phase-field simulation of spinodal decomposition. It is implemented as a three-dimensional 7-point stencil and uses three alternating kernels. The side length of the simulated domain for each configuration is $\approx \sqrt[3]{N_{\mathtt{GPU}}} \times 768$.

Note that the choice of problem size for each benchmark involves a tradeoff between maximizing per-GPU memory usage and avoiding other limiting factors, such as available host memory, for larger configurations. All benchmarks were run ten times for each GPU configuration and the presented results are the median value of those runs. Most benchmarks include warmup iterations to ensure all kernels are compiled and buffers are resized to their final dimensions. Figure 4 shows the results. All results are presented in terms of domain-specific throughput metrics, with the y-axis indicating the throughput achieved *per GPU* This is in principle similar to *parallel efficiency* (a horizontal line would indicate perfect scaling), while also showing absolute differences in achieved throughput between different variants of the application. We include comparisons with baseline MPI + SYCL implementations for four of the benchmarks. These baselines have been optimized to a reasonable degree and use MPI primitives best suited to the particular use case (for example collective *allgather* operations for N-Body). For the two stencil codes we include variants that make use of CUDA-aware MPI RDMA through SYCL's backend interoperability features.

Black-Scholes: Being a compute intensive benchmark without communication, it acts as a baseline for the other benchmarks. We can see that for up to 128 GPUs, Celerity tracks closely with the MPI implementation, maintaining very high parallel efficiency of 95%. We attribute the slight downwards trend to synchronization overhead involved in timing of the application, as run times are very short in absolute terms (tens of milliseconds).

N-Body: In absolute terms, on a single GPU, the N-Body benchmark achieves approx. 5.2 TFLOPs/s in double precision arithmetic, or nearly 75% of the theoretical hardware peak of 7.0 TFLOPs/s.

The multi-GPU scaling of the N-Body benchmark with Celerity is generally good, achieving more than 70% efficiency on 128 GPUs, despite the required all-to-all communication. The shape of this chart appears irregular with slightly superlinear performance at 2 and 8 GPUs and a dip in performance for the 16 GPU case. These artifacts can be attributed to a large variance in kernel run times that stem from caching effects that become visible when adjusting the body count for this weak-scaling experiment.

This is further backed by the MPI implementation exhibiting the same dip at 16 GPUs, while otherwise tracking closely with Celerity. We see that Celerity slightly outperforms MPI for most configurations; we attribute this to the larger overhead incurred by using one rank per GPU in the MPI implementation. For 128 GPUs MPI gains the lead due to the use of collective communication operations, a feature that will be explored in future work.

Unstructured Mesh: For the Unstructured Mesh finite element simulation, as mentioned previously, the absolute performance level is best investigated by observing the effective memory bandwidth achieved during a run. On a single GPU, the benchmark achieves a throughput of 1.8 billion elements per second, or an estimated 441 GB/s, which equals 49% of the 900 GB/s theoretical hardware peak. Given the – per necessity – data-dependent and unstructured memory accesses required by a simulation on an unstructured mesh, this is a good result.

In terms of multi-GPU scaling, the unstructured mesh benchmark shows no particular anomalies, achieving an excellent 90% parallel efficiency.

A very interesting algorithmic property of this code is that finite elements on the borders between subdomains are updated first, and independently of the bulk of inner elements. Without any additional work on behalf of the application developer, the Celerity runtime system is able to leverage the pattern of data accesses induced by this behavior in order to overlap transfers with computation. In practice, updates to the border elements are transmitted while the current update on inner elements is being performed.

WaveSim and *Cahn-Hilliard*: For the two stencil codes, we compare four different split configurations, enabled by means of task hints (as described in Section IV-A). 1D refers to a

TABLE I
PER-NODE SPECIFICATION FOR THE BENCHMARKING SYSTEM.

Host:	IBM POWER9 16C 3 GHz, 256 GB RAM
GPUs:	4x NVIDIA V100 GPUs 16 GB, NVLink 2.0
Interconnect:	Mellanox InfiniBand EDR DragonFly+
Software:	RHEL 8.1; Spectrum MPI 10.4.0.03rtm4; GPU driver 450.51.06; CUDA 11.0, hipSYCL v0.9.1

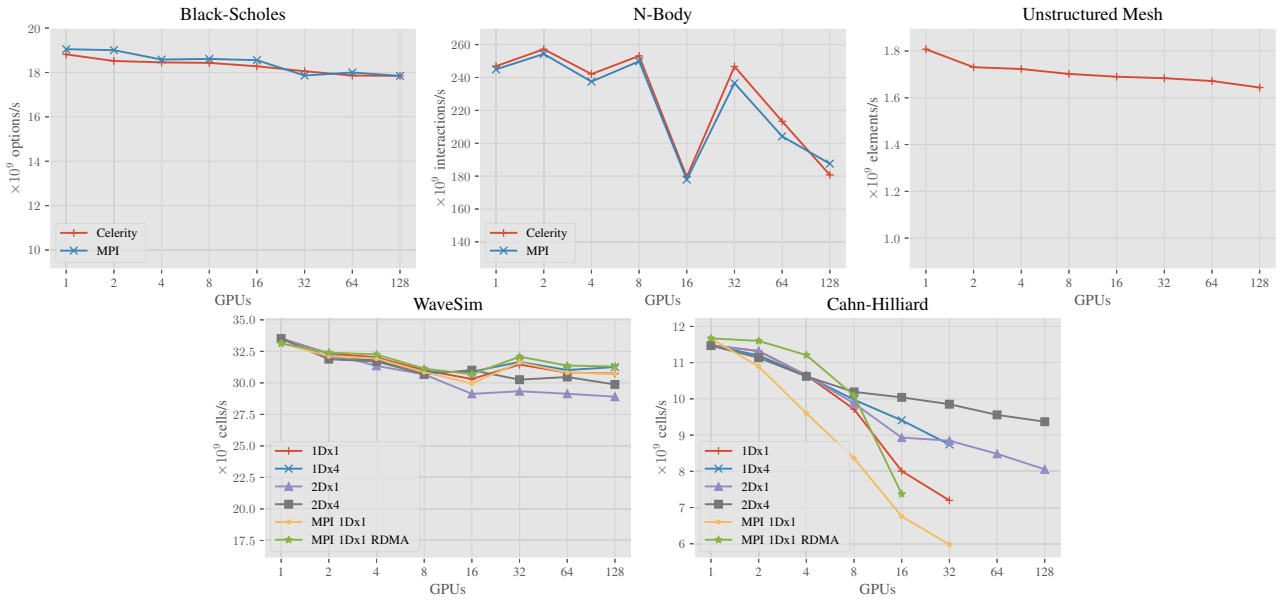

Fig. 4. *Per-GPU* throughput of weak scaling experiments performed on Marconi-100 (higher is better). Note the cropped y-axes.

split along the slowest dimension 0, whereas 2D refers to a tiled split in dimensions 0 and 1. $\times N$ is the oversubscription factor, where $\times 4$ means that four chunks are generated for each GPU. We can see that for *WaveSim*, all split types perform well, achieving at least 85% efficiency. The oversubscribed 1-dimensional split performs best, achieving an excellent 93% parallel efficiency. By using optimized CUDA-aware RDMA communication (a feature not yet supported by Celerity) the baseline MPI implementation compensates for the lack of oversubscription, achieving the same throughput as Celerity.

We conclude that *WaveSim* is mostly limited by latency rather than communication bandwidth, and overlapping works better in a 1-dimensional split, as there are fewer communications to hide. *Cahn-Hilliard* on the other hand shows the opposite behavior: With it being a 3-dimensional stencil, the amount of data transferred between each iteration is much larger than for *WaveSim*, making the choice of split and effective computation/communication overlapping much more important. Beyond 32 GPUs (or 16 in the case of MPI RDMA, due to scratch buffers), the size of the 2D slabs created by the 1-dimensional split exceeds the memory available on a single GPU. The 2D$\times 4$ split continues to scale however, achieving a solid 81% parallel efficiency on 128 GPUs. We note that the switch from a 1D split without oversubscription to 2D$\times 4$ amounts to adding two lines of code.

We summarize that all of the presented benchmarks achieve good scalability, and track closely with the MPI + SYCL baseline implementations where available. While all of the discussed applications can be considered *mini-apps*, we note that they represent a selection of communication patterns that form the basis of many larger applications. In fact, several of the improvements discussed in this work, in particular distributed scheduling, were implemented out of necessity for achieving scalability in real world production applications, results for which will be published in future work.

VI. Conclusion

In this work, we have given a detailed review of the Celerity programming model and distributed runtime design. We replaced the centralized master/worker scheduling model by a fully decentralized scheme which improves the scheduling complexity for a worst-case scenario from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, while maintaining the existing high-productivity API surface.

We evaluated the feasibility of the proposed design in five application benchmarks from different scientific domains on up to 128 GPUs. The results generally show good scalability, with four out of five benchmarks achieving over 80% parallel efficiency on 128 GPUs. This can at least in part be attributed to Celerity's innate ability to perform automatic computation/communication overlapping. Furthermore, we demonstrated how the performance of two stencil codes can be considerably improved by means of a novel task hinting API, which allows quick experimentation with different splitting patterns while requiring only minimal code changes.

Acknowledgment

This project has received funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement No 956137 as well from the Austrian Research Promotion Agency under grant agreement No 879201.

REFERENCES

[1] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen, "GROMACS: A message-passing parallel molecular dynamics implementation," *Computer Physics Communications*, vol. 91, no. 1, pp. 43–56, Sep. 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/001046559500042E

[2] Message Passing Interface Forum. (2015) MPI: A Message-Passing Interface Standard, Version 3.1. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[3] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, Jun. 2012, pp. 341–352. [Online]. Available: https://doi.org/10.1145/2304576.2304623

[4] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, Jun. 2010, pp. 224–231.

[5] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "LibWater: heterogeneous distributed computing made easy," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 161–172. [Online]. Available: https://doi.org/10.1145/2464996.2465008

[6] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware Task Scheduling on Multi-accelerator Based Platforms," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, 2010.

[7] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, Mar. 2002, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[8] S. Thibault, "On Runtime Systems for Task-based Programming on Heterogeneous Platforms," Thesis, Université de Bordeaux, Dec. 2018. [Online]. Available: https://hal.inria.fr/tel-01959127

[9] S. Kumar, "Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources," PhD Thesis, Université de Bordeaux, Apr. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01538516

[10] M. Bauer, S. Treichler, E. Slaugther, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 1151–1158, iSSN: 1530-2075.

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

[14] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[15] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, Aug. 2007, publisher: SAGE Publications Ltd STM. [Online]. Available: https://doi.org/10.1177/1094342007078442

[16] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2015, pp. 1–12, iSSN: 2167-4337.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Mar. 2016, arXiv:1603.04467 [cs]. [Online]. Available: http://arxiv.org/abs/1603.04467

[18] The Khronos Group. (2022) SYCL Specification, Version 2020 Revision 5. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[19] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 277–288, Feb. 2011. [Online]. Available: https://doi.org/10.1145/2038037.1941591

[20] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2013, pp. 245–255, iSSN: 1089-795X.

[21] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating Multiple Data-Parallel Kernels on Multiple Devices," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015, pp. 355–366, iSSN: 1089-795X.

[22] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger, "DistCL: A Framework for the Distributed Execution of OpenCL Kernels," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug. 2013, pp. 556–566, iSSN: 2375-0227.

[23] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, and R. V. v. Nieuwpoort, "Lightning: Scaling the GPU Programming Model Beyond a Single GPU." IEEE Computer Society, May 2022, pp. 492–503. [Online]. Available: https://www.computer.org/csdl/proceedings-article/ipdps/2022/810600a492/1F1VVLx00JG

[24] P. Thoman, H. Jordan, P. Gschwandtner, T. Fahringer, B. Cosenza, and B. Juurlink, "CELERITY: Towards an Effective Programming Interface for GPU Clusters," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2018.

[25] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-Level C++ for Accelerator Clusters," in *Euro-Par 2019: Parallel Processing*. Springer International Publishing, 2019, vol. 11725, pp. 291–303, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-29400-7_21

[26] F. Knorr, P. Thoman, and T. Fahringer, "Declarative data flow in a graph-based distributed memory runtime system," in *International Symposium on High-level Parallel Programming and Applications (HLPP 2022)*, 2022.

[27] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *ACM SIGMOD Record*, vol. 14, no. 2, pp. 47–57, Jun. 1984. [Online]. Available: https://doi.org/10.1145/971697.602266

[28] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[29] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 127–136.

[30] Z. Jin. HeCBench. [Online]. Available: https://github.com/zjin-lcf/HeCBench

[31] M. Yousefi. Cahn Hilliard Phase-Field Simulation by using CUDA GPU: Exascale Computational Materials Science. [Online]. Available: https://github.com/myousefi2016/Cahn-Hilliard-CUDA