Celerity: High-level C++ for Accelerator Clusters

Peter Thoman[1], Philip Salzmann[1], Biagio Cosenza[2], and Thomas Fahringer[1]

[1] University of Innsbruck, 6020 Innsbruck, Austria
{petert,psalz,tf}@dps.uibk.ac.at
[2] Technical University of Berlin, 10623 Berlin, Germany
cosenza@tu-berlin.de

Abstract. In the face of ever-slowing single-thread performance growth for CPUs, the scientific and engineering communities increasingly turn to accelerator parallelization to tackle growing application workloads. Existing means of targeting distributed memory accelerator clusters impose severe programmability barriers and maintenance burdens.
The Celerity programming environment seeks to enable developers to scale C++ applications to accelerator clusters with relative ease, while leveraging and extending the SYCL domain-specific embedded language. By having users provide minimal information about how data is accessed within compute kernels, Celerity automatically distributes work and data.
We introduce the Celerity C++ API as well as a prototype implementation, demonstrating that existing SYCL code can be brought to distributed memory clusters with only a small set of changes that follow established idioms. The Celerity prototype runtime implementation is shown to have comparable performance to more traditional approaches to distributed memory accelerator programming, such as MPI+OpenCL, with significantly lower implementation complexity.

1 Introduction

As Moore's Law is dying [5], end-users in many computational domains are turning to increasingly sophisticated parallelization methods in order to see speedups in their workloads. One particularly promising avenue is GPU computing, which leverages the high peak performance and energy efficiency of GPUs – or, more generally, *accelerators* – to implement suitable algorithms. To achieve even better performance and to tackle larger workloads, targeting a compute cluster of accelerators can be highly beneficial.

While these considerations seem straightforward from a hardware-centric, parallelism expert perspective, in practical use the *programmability* of such systems is a significant hindrance to their broader adoption in domain sciences [9]. Targeting accelerators requires an accelerator-specific API and programming model, and the most widespread vendor-agnostic option, OpenCL [12], assumes familiarity with low-level hardware details and imposes significant implementation effort and maintenance overhead. When targeting clusters, these issues are

compounded by the additional requirement of managing distributed-memory semantics, usually by leveraging MPI [10] for explicit message passing.

Not only does this type of software stack impose programmability and maintenance challenges, it also greatly reduces flexibility in optimizing and adapting a given program for current and future hardware architectures by hard-coding data and work distribution strategies.

Celerity aims to address these shortcomings, while keeping the barrier of adoption for users at a minimum. To this end, the basis of Celerity is SYCL [13], an open industry standard for programming arbitrary OpenCL devices using modern high-level C++. Celerity automates the parallelization of SYCL programs across heterogeneous computing clusters, opting to provide reasonable defaults and performance for domain scientists, while leaving room for manual tuning on a per-application basis. In this work we focus specifically on the programmability goals of Celerity, with our central contributions comprising:

- The *Celerity API* extending industry-standard SYCL programs to distributed memory with minimal programmer overhead, by introducing the concept of *custom data requirement functors* and a *virtual global queue*.
- A *prototype runtime implementation* based on a *multi-level task graph*, which is implicitly generated and distributed during the execution of a Celerity program.
- An *evaluation* of this API and prototype runtime implementation from *both programmability and performance perspectives*, compared to traditional MPI+OpenCL and state-of-the-art MPI+SYCL implementations.

2 Related Work

The increasing prevalence of parallelism in all application domains has warranted significant research into how the scheduling and partitioning of parallel codes can be automated [14]. In this section we summarize a number of languages and libraries which relate to the goals of Celerity.

Charm++ [7] is a task-based distributed runtime system and C++ language extension. Its global shared address space execution model allows executing asynchronous functions on distributed objects called *chares*, which may reside on a local or remote processor and which are transparently invoked through internally passed messages. Charm++ supports GPUs with a GPU Manager component, but is not natively designed for accelerator clusters. Furthermore, somewhat reducing programmability, a so-called *interface definition file* has to be provided for user-defined classes.

StarPU [2] is a task-based runtime system that provides data management facilities and sophisticated task scheduling algorithms for heterogeneous platforms, with the ability to easily implement custom schedulers. Its API is however still relatively low-level, and does not provide multi-node distributed memory parallelism out of the box. While it does feature facilities to integrate with MPI, even

automatically transferring data between nodes based on specified task requirements [1], the splitting and distribution of work still remains the responsibility of the user.

OmpSs [6], another task-based runtime system and compiler extension, builds on top of the well-established OpenMP standard. Using extended OpenMP `#pragma` directives to express data dependencies between tasks allows for asynchronous task parallelism. The ability to provide different implementations depending on the target device enables OmpSs to also support heterogeneous hardware. By annotating functions that wrap MPI communications with information about their data dependencies, effectively turning them into tasks as well, OmpSs can integrate them into the task graph and interleave OpenMP computations with MPI data transfers. However, the runtime itself has no explicit notion of MPI and thus again all work splitting and distribution decisions are offloaded to the user.

PHAST [11] is a heterogeneous high-level C++ library for programming multi- and many-core architectures. It features data containers for different dimensionalities and provides various STL-like parallel algorithms that can operate on said containers. Additionally, custom kernel functors are supported through a set of macros that wrap function headers and bodies. It does not feature any facilities for targeting distributed memory systems.

The Kokkos C++ library [4] allows thread parallel execution on manycore devices and attempts to provide performance portability by automatically adjusting data layouts of multidimensional arrays to fit the access patterns of a target device. It does not provide any facilities for distributed memory parallelism (i.e., everything has to be done manually), and it is again a rather low-level approach.

Legion [3] allows for even more flexibility by describing data accessed by tasks in terms of *logical regions*, while delegating decisions about how to lay them out in physical memory, alongside the decision of where to run tasks, to a (potentially user-provided) *mapper*. Logical regions and associated tasks can be partitioned and executed across heterogeneous clusters, with Legion taking care of ensuring data coherence between nodes. Again, partitioning decisions as well as the mapping of tasks to devices are delegated to the user.

While each of these approaches is well-suited to particular use cases, none of them with the exception of Legion and PHAST were natively designed for accelerator computing. Crucially, they all operate on a lower level of abstraction and thus require higher implementation effort compared to Celerity when targeting distributed memory accelerator clusters.

3 The Celerity System

Figure 1 gives a high-level overview of the entire Celerity system. At its core, the project extends the ease of use of the SYCL domain-specific embedded language to distributed clusters. While execution of shared memory parallel kernels is still handled by the SYCL runtime on each individual worker node, the Celerity

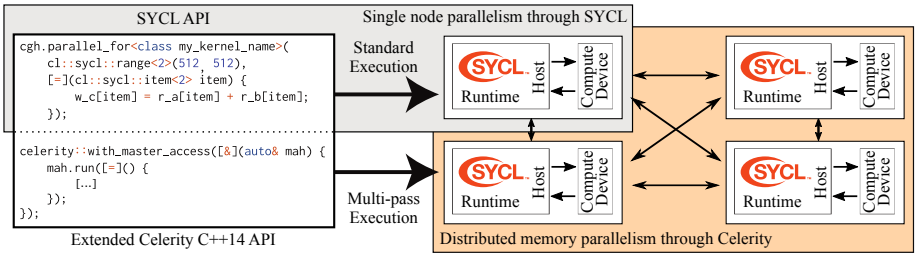


Fig. 1: A bird's-eye view of the Celerity system.

runtime acts as a wrapper around each compute process, handling inter-node communication and scheduling.

The central components making this possible are Celerity's user-facing *API*, and its *multi-pass execution* process at runtime. The latter allows the distributed system to gain a shared understanding of the program being executed and automatically distribute kernel executions while ensuring that their data requirements are fulfilled.

3.1 The Programming Interface

```
1  sycl::queue queue;
2
3  sycl::buffer<float, 2> buf_a(hst_a.data(), sycl::range<2>(512, 512));
4  sycl::buffer<float, 2> buf_b(hst_b.data(), sycl::range<2>(512, 512));
5  sycl::buffer<float, 2> buf_c(sycl::range<2>(512, 512));
6
7  queue.submit([&](sycl::handler& cgh) {
8      auto r_a = buf_a.get_access<acc::read>(cgh);
9      auto r_b = buf_b.get_access<acc::read>(cgh);
10     auto w_c = buf_c.get_access<acc::write>(cgh);
11     cgh.parallel_for<class my_kernel_name>(sycl::range<2>(512, 512),
12         [=](sycl::item<2> itm) {
13             w_c[itm] = r_a[itm] + r_b[itm];
14         });
15 });
```

Listing 1: A simple SYCL program that adds up two buffers.

Listing 1 illustrates the main portions of a simple SYCL program. Note that SYCL-related types are marked in orange, and that we assume a prologue of `namespace sycl = cl::sycl` and `using acc = sycl::access::mode` in all our examples for brevity. At its core, a SYCL program consists of a *queue* used to submit commands to a compute device, as well as data structures such as *buffers*, and the *kernels* which operate on them.

Lines 3 through 5 define three two-dimensional `float` buffers of size 512×512, the first two of which are being initialized using existing host data. On line 7, a so-called *command group* is submitted to the execution queue. Command groups serve as wrappers for device kernel calls, allowing the specification of

data access requirements as well as the kernel code that operates on said data in one place, tied together by the *command group handler* `cgh`. This handler is passed as an argument into the C++ lambda expression constituting the command group, and is used to request *device accessors* on lines 8 through 10. Accessors concisely express the intent of the subsequent operation (reading, writing, or both), allowing the SYCL runtime to determine dependencies between subsequent kernel invocations and schedule data transfers required to ensure data coherence between the host and device.

In this particular command group, read access to buffers `buf_a` and `buf_b` is requested over their entire range. Conversely, write access is requested for buffer `buf_c`. Finally, on lines 11 through 15 the actual kernel is specified: A simple sum of the two read-buffers is computed. Note that each kernel has to be invoked using a template method such as `parallel_for<class kernel_name>` which uses a unique *tag-type* to allow linking of the intermediate representation of a kernel – which potentially is generated in a separate, implementation-defined compilation step – to the kernel invocation in the host program.

```
1  celerity::distr_queue queue;
2
3  celerity::buffer<float, 2> buf_a(hst_a.data(),sycl::range<2>(512, 512));
4  celerity::buffer<float, 2> buf_b(hst_b.data(),sycl::range<2>(512, 512));
5  celerity::buffer<float, 2> buf_c(sycl::range<2>(512, 512));
6
7  queue.submit([=](celerity::handler& cgh) {
8      auto one_to_one = celerity::access::one_to_one<2>();
9      auto r_a = buf_a.get_access<acc::read>(cgh, one_to_one);
10     auto r_b = buf_b.get_access<acc::read>(cgh, one_to_one);
11     auto w_c = buf_c.get_access<acc::write>(cgh, one_to_one);
12     cgh.parallel_for<class my_kernel_name>(sycl::range<2>(512, 512),
13         [=](sycl::item<2> itm) {
14             w_c[itm] = r_a[itm] + r_b[itm];
15         });
16 });
```

Listing 2: The same program as shown in Listing 1, now using the Celerity API.

Listing 2 shows the Celerity version of the program previously seen in Listing 1. The first observation of note is that the overall structure of the two programs is quite similar. While some objects now live under the `celerity` namespace, we still have buffers, a queue and a command group containing buffer accessors as well as a kernel invocation. In fact, whenever possible, the original objects from the `sycl` namespace are used, allowing for code to be migrated to Celerity with minimal effort. Since unlike SYCL, Celerity may call command groups multiple times during a user program's execution – as will be discussed in Section 3.2 – it is recommended to capture all required buffers by value rather than reference, as can be seen on line 7.

Notice the lack of typical indicators of a distributed memory parallel program, such as the notion of a local *rank* and the total number of nodes running. Nonetheless, given this program, the Celerity runtime is able to distribute the workload associated with kernel `my_kernel_name` to any number of workers dynamically at runtime.

This is made possible in large parts by one of Celerity's most significant API additions: So-called *range mappers* specify the data access behavior of each kernel and are provided as the final parameter to Celerity's `get_access()` calls on lines 9 through 11.

Specifying Data Requirements When Celerity workers execute disjunct parts of the same logical kernel in parallel, they typically each require different portions of some input data. Distributing full buffers to each node and compute device is theoretically valid, but clearly untenable in terms of performance due to potentially redundant data transfers. Furthermore, a different part of the result is produced on each worker, a fact that needs to be taken into consideration when deciding on how to use their output in subsequent computations.

What is required is a flexible and minimally invasive method of specifying exactly what data requirements a kernel has, in a way that is independent of data distribution and work scheduling. In Celerity this is accomplished by *range mappers*, which are arbitrary functors with the following signature:

```
(celerity::chunk<KD>) -> celerity::subrange<BD>
```

Here, a `celerity::chunk<KD>` specifies an N-dimensional chunk of a kernel, containing an offset, a range, and a global size. The offset and range of a chunk depend on how the Celerity runtime decides to distribute a kernel across worker nodes, i.e., each chunk represents a portion of the execution of a kernel, each assigned to a particular worker node. A chunk is then mapped to a `celerity::subrange<BD>`, which specifies the offset and range of a data buffer the kernel chunk will operate on. `KD` and `BD` can differ: a 2D kernel may for example access data stored in a 1D buffer.

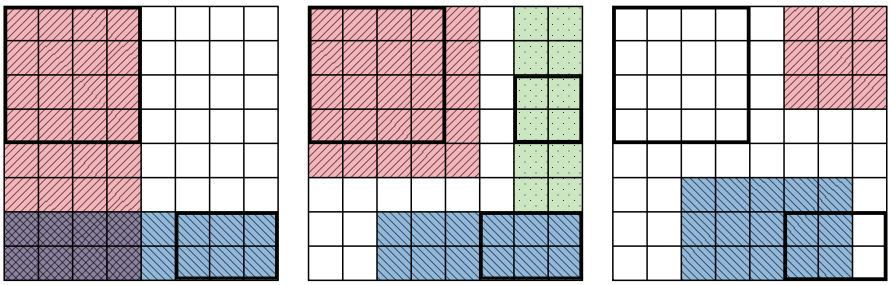
```
1  queue.submit([=](celerity::handler& cgh) {
2      auto i_r = input.get_access<acc::read>(cgh, celerity::access::
       one_to_one<2>());
3      auto o_w = output.get_access<acc::write>(cgh,
4          [](celerity::chunk<2> chnk) -> celerity::subrange<2> {
5              return { { chnk.offset[1], chnk.offset[0] },
6                       {  chnk.range[1],  chnk.range[0] } };
7          });
8      cgh.parallel_for<class transpose>(sycl::range<2>(128, 256),
9          [=](sycl::item<2> itm) {
10             auto idx = sycl::id<2>(itm[1], itm[0]);
11             o_w[idx] = i_r[itm];
12         });
13 });
```

Listing 3: Range mapper for computing a matrix transpose.

Listing 3 shows how a range mapper can be used to specify data requirements for a simple matrix transpose. Implemented on lines 4 through 7, this range mapper specifies that for any input matrix of size $n \times m$, the kernel will write to an output matrix of size $m \times n$. Crucially, it also specifies that for any given submatrix of size $p \times q$ at location (i, j) with $i + p \leq n, j + q \leq m$, it will write to the corresponding output submatrix of size $q \times p$ at location (j, i).

Note that the range mapper merely acts as a contract of how data is going to be accessed, and does not affect the actual kernel in any way. The index-reversal and subsequent assignment on lines 10 and 11 is where the actual transpose is computed. Given this range mapper, regardless of the number of workers executing the kernel (i.e., the number of chunks the kernel is split into), it is always clear where which parts of the resulting matrix are computed. Likewise, the Celerity runtime also knows exactly what parts of the input matrix each worker node requires in order to produce the transpose. While in Listing 3 a matrix of size 128×256 is transposed to a 256×128 matrix, notice how this size is not relevant to the range mapper definition itself. This allows both users and library authors to write mappers in a generic and reusable way.

Built-in Range Mappers The Celerity API provides several built-in range mappers for common data access patterns. One that is very frequently used, `celerity::access::one_to_one`, can be seen in both Listing 2 and Listing 3. This range mapper specifies that a kernel, for every individual work item, will access a buffer only at that same global index. In Listing 3 this means that to compute the "result" for work item (i, j), the kernel will access the input matrix at index (i, j) as well – while writing to the output matrix at index (j, i), as specified by the custom mapper described previously.



(a) `slice<KD>` (b) `neighborhood<KD>` (c) `fixed<KD, BD>`

Fig. 2: Example inputs and outputs for three built-in range mappers, in this case applied to 2-dimensional kernels and buffers.

Figure 2 illustrates three additional range mappers currently provided by the Celerity API. Thick lines indicate the input chunk, colored areas the associated output subrange (i.e., the accessed portion of a buffer). Each color corresponds to a different configuration of a range mapper. The `slice` range mapper allows extending the range of a chunk along one dimension indefinitely, thus selecting an entire slice of a buffer in that dimension. A common use case for this is matrix multiplication. The `neighborhood` range mapper allows selecting a given border around a chunk, a pattern that is commonly encountered in stencil codes.

Finally, the `fixed` range mapper always returns a given, fixed subrange of a buffer, regardless of the input chunk. This can be useful when each worker needs to read the same input buffer, e.g. a mask when applying a discrete convolution.

3.2 The Prototype Runtime System

The Celerity runtime system is a multi-threaded application built on top of SYCL and MPI that runs in concert with a user-defined program in a *single program multiple data* (SPMD) fashion. It uses a master/worker execution model, where the master node is responsible for scheduling all the distributed work. Worker nodes encapsulate the available accelerator hardware (i.e., one worker is spawned per accelerator). They asynchronously receive *commands* from the master node and execute them as soon as possible.

Commands are lightweight asynchronous operations such as the execution of a certain chunk of a kernel, or initiating a data transfer with another worker. The master node generates commands as part of a graph and includes dependency information within the directives sent to each worker. This allows workers to execute commands as soon as all of their dependencies are satisfied. The resources that they operate on, i.e. kernels and buffers, are identified by unique numerical IDs within the lightweight command data structure. To enable this, each worker, as well as the master node, has an implicit shared understanding of what any particular ID refers to. This is possible because each Celerity process executes the exact same user code, deterministically assigning IDs to newly created objects.

To allow for Celerity to retain the familiar SYCL syntax for specifying kernels using command groups, without performing lots of duplicated computational work on each worker node, certain parts of a Celerity program are executed twice, in a process we call *multi-pass execution*.

In the *pre-pass*, command groups are executed solely to collect their defining properties, such as buffer accesses and their range mappers, as well as the global size of a kernel and the kernel function itself. Using this information, the master node constructs a task graph that respects consumer/producer relationships and other data dependencies between subsequent kernel executions. From this, it then generates the more fine grained *command graph* which contains commands assigned to particular workers. Once a kernel execution command is received by a worker, it then executes the corresponding command group a second time. During this *live-pass*, the actual computation on the device takes place. However, instead of using the global size provided by the user, it is transparently executed on the chunk assigned by the master node.

While the pre-pass is performed immediately upon first encounter of a command group within a user program, the Celerity runtime needs to be able to defer the live-pass to a later point in time in order to schedule additional work ahead. If that were not the case, each command group would act as an implicit global barrier, which is highly undesirable. This in turn means that Celerity has to retain command groups internally to be able to execute the live-pass at a later point in time, independently of the user program's execution flow. As the

combination of C++11 lambda closures and this deferred execution can cause hard to diagnose lifetime bugs, we recommend to only capture parameters by value. While this is currently being enforced through static assertions, more sophisticated diagnostics enabled by compiler extensions might be explored in the future.

It is crucial to note that pre-pass and live-pass execution, as well as task and command graph generation all occur asynchronously – and, in fact, at the same time in any non-trivial program. Most importantly, this means that worker nodes can execute their local command graphs, performing computations and peer-to-peer data transfers, completely independently of the main task generation process. In practice, this system ensures that Celerity imposes no bandwidth overhead compared to a fully decentralized approach, and no latency overhead outside of a startup phase during which the initial commands are generated.

4 Evaluation

This section evaluates Celerity as a framework for writing distributed memory accelerator applications. To this end, we compare the Celerity implementation of three programs with more traditional implementations.

The example programs highlighted in this chapter are: i) *MatMul*, a sequence of dense matrix-matrix multiplications, ii) *Pendulum*, which simulates the behavior of a pendulum swinging across a board with an arrangement of magnets, and iii) *WaveSim*, a simulation of the 2D wave equation. For each of these programs, we compare a Celerity version with a MPI+SYCL version representing the current state of the art. For *MatMul*, we additionally consider a traditional MPI+OpenCL variant, to verify that SYCL performs equally to this baseline.

4.1 Programmability

To estimate and compare the differences of each implementation from a programmer's point of view, we present two different metrics. First, the widely employed *cyclomatic complexity* [8] measures code complexity in terms of the number of linearly independent paths of execution a program could take. It is computed using the `pmccabe` command-line utility, which is available for many Linux distributions. As a somewhat simpler – but perhaps more immediately apprehensible – metric, the number of non-comment lines of code (NLOC) is provided. Note that all non-essential code, such as selection of compute devices, instrumentation, and result verification is excluded from these metrics. The former two need to be performed manually for the classical implementations while they are included in Celerity, and the latter is the same across all versions.

Figure 3 summarizes the programmability metrics for all three applications. Note that across all programs and both metrics, there is a very significant decrease in implementation complexity of about factor 2 going from MPI+SYCL to Celerity. This is primarily caused by eliminating the need for most traditional trappings associated with distributed memory programming, including manual

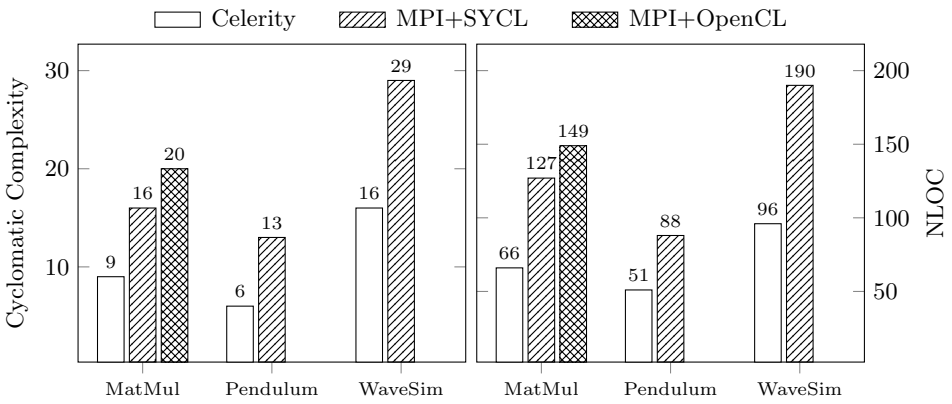


Fig. 3: Comparison of programmability metrics.

work and data distribution as well as data synchronization. The smallest difference is observed in NLOC for *Pendulum*, which is due to the fact that there is no data redistribution required in this algorithm outside of the initial conditions and final aggregation.

When considering the MPI+OpenCL version for *MatMul*, we see that there is a further increase in implementation effort associated with the lower-level OpenCL API compared to the high-level SYCL, although a less significant one than what is required for distributed memory.

4.2 Performance

While Section 4.1 demonstrates the significant programmability advantages conferred by Celerity compared to state-of-the-art methods, these advantages would be relatively meaningless if they came at a large general loss in performance potential. Therefore, although the current Celerity runtime implementation is still only a prototype, we provide some initial benchmarks in this section which demonstrate its performance.

Host:	AMD Ryzen Threadripper 2920X 12-Core, 32 GB DDR4 RAM
GPUs:	4x Nvidia RTX 2070
Interconnect:	10 Gigabit Ethernet
Software:	Ubuntu 18.04; OpenMPI 4.0.0; GPU driver 410.79; hipSYCL 0.7.9

Table 1: Per-node specification for the benchmarking system.

All benchmarks were executed on a small cluster comprised of 8 GPUs, situated in two distinct but otherwise identical machines with 4 GPUs each. Thus, all runs using 4 GPUs or less are on a single machine, while runs with 8 GPUs

utilize both. Table 1 summarizes the hardware of each machine, as well as the software stack used for this evaluation. For each benchmark, the workload is statically distributed in a uniform fashion, i.e., no load-balancing strategies are employed. Figure 4 illustrates the speedup achieved by each application scaling from 1 to 8 GPUs (corresponding to 2304 to 18432 CUDA cores). The results presented are based on the median of 5 benchmark runs for each configuration.

Before discussing the individual results, note that we do not include the MPI+OpenCL version of *MatMul* in this chart. Its performance is exactly equivalent to the MPI+SYCL version and is therefore omitted for clarity.

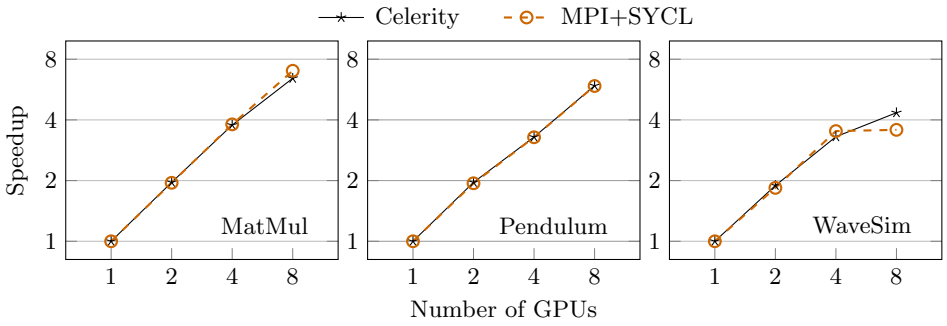


Fig. 4: Speedup for 1 to 8 GPUs of Celerity compared to manual MPI+SYCL.

Evidently, Celerity offers performance comparable to the manual distributed memory accelerator implementation in all three applications benchmarked. This is most apparent in *Pendulum*, which shows the exact same speedup for both variants. This is a result of the absence of intermediate data transfers resulting in a relative lack of network transmission impact on the overall execution time.

For *MatMul*, Celerity shows equivalent behavior up to 4 GPUs, but slightly worse scaling to 8 GPUs. We have examined this drop in efficiency and determined that it is due to the manual MPI version leveraging collective communication for data transfers in between individual matrix multiplications, while the Celerity prototype currently performs point-to-point communication. This is a quality-of-implementation issue rather than an inherent feature of our approach, and we intend to improve on this behavior in future work.

Finally, *WaveSim* actually demonstrates better scaling from 4 to 8 GPUs in its Celerity variant than it does in MPI+SYCL. This stencil-like code is relatively latency-sensitive, as ghost cells need to be exchanged after every time step. The Celerity version benefits from the fact that all (automatic) data distribution is inherently implemented asynchronously in our runtime system. While the MPI+SYCL version could also be made entirely asynchronous, likely resulting in similar performance, this would further increase its implementation complexity in the metrics discussed in Section 4.1.

5 Conclusion

In this work we have introduced the Celerity API for programming distributed memory accelerator clusters. It builds on the SYCL industry standard, and allows extending existing single-GPU programs to GPU clusters with a minimal set of changes, while shielding the user from much of the complexity associated with work and data distribution on clusters.

This is achieved by i) a concise API extension focusing on flexible, reusable *range mapper* functors, ii) a multi-pass runtime execution model which builds an implicit, shared understanding of the data and work primitives – buffers and kernel invocations – involved in the computation at runtime, and iii) a fully asynchronous execution environment implementation for this model.

In concrete terms, programmability metrics show significant ease of implementation advantages for our approach compared to state-of-the-art MPI+SYCL combinations, with improvements around factor 2 in both cyclomatic complexity as well as lines of code. This advantage is even more pronounced when comparing against a more traditional MPI+OpenCL implementation version.

Crucially, these programmability advances do not come at a significant performance overhead. Execution times for the Celerity implementation versions are comparable to their respective manual distributed memory accelerator versions in all programs tested, with minor advantages and disadvantages in individual benchmarks.

The approach introduced in Celerity enables a broad spectrum of future research. On the API level, even more concise or domain-specific abstractions can be introduced to further improve ease of use for domain scientists. Independently – and without requiring any change to the input programs – the efficiency of the runtime system can be increased, by e.g. introducing command graph optimizations which gather individual transfers into collective operations, or by improving scheduling for kernels with non-uniform workloads.

Acknowledgments

This research has been partially funded by the FWF (I 3388) and DFG (CO 1544/1-1, project number 360291326) as part of the CELERITY project.

References

1. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.P.: Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE Transactions on Parallel and Distributed Systems (2017)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)

3. Bauer, M., Treichler, S., Slaugther, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE (2012)
4. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216 (2014)
5. Courtland, R.: Gordon Moore: The Man Whose Name Means Progress (2015), https://spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress
6. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(02), 173–193 (2011)
7. Kale, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. vol. 10, pp. 91–108 (1993)
8. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **SE-2**(4), 308–320 (1976)
9. Meade, A., Deeptimahanti, D.K., Buckley, J., Collins, J.J.: An empirical study of data decomposition for software parallelization. Journal of Systems and Software **125**, 401–416 (2017)
10. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.1 (2015), https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
11. Peccerillo, B., Bartolini, S.: PHAST Library — Enabling Single-Source and High Performance Code for GPUs and Multi-cores. In: Smari, W.W., Simulation, I.C.o.H.P.C.&. (eds.) 2017 International Conference on High Performance Computing & Simulation. pp. 715–718. IEEE, Piscataway, NJ (2017)
12. The Khronos Group: The OpenCL Specification, Version 1.2 Revision 19 (2012), https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
13. The Khronos Group: SYCL Specification, Version 1.2.1 Revision 3 (2018), https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
14. Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing **74**(4), 1422–1434 (2018)