International Journal of Parallel Programming (2022) 50:341–359
https://doi.org/10.1007/s10766-022-00731-8

The Celerity High-level API: C++20 for Accelerator Clusters

Peter Thoman[1] · Florian Tischler[1] · Philip Salzmann[1] · Thomas Fahringer[1]

Received: 5 October 2021 / Accepted: 16 March 2022 / Published online: 22 April 2022

Abstract
Providing convenient APIs and notations for data parallelism which remain accessible for programmers while still providing good performance has been a long-term goal of researchers as well as language and library designers. C++20 introduces ranges and views, as well as the composition of operations on them using a concise syntax, but the efficient implementation of these library features is restricted to CPUs. We present the Celerity High-level API, which makes similarly concise mechanisms applicable to GPUs and accelerators, and even distributed memory clusters of GPUs. Crucially, we achieve this very high level of abstraction without a significant negative impact on performance compared to a lower-level implementation, and without introducing any non-standard toolchain components or compilers, by implementing a C++ library infrastructure on top of the Celerity system. This is made possible by two central API design and implementation strategies, which form the core of our contribution. Firstly, gathering as much information as possible at compile-time and using metaprogramming techniques to automatically fuse several distinctly formulated processing steps into a single accelerator kernel invocation. And secondly, leveraging C++20 "Concepts" in order to avoid type erasure, allowing for highly efficient code generation. We have evaluated our approach quantitatively in a comparison to lower-level manual implementations of several benchmarks, demonstrating its low overhead. Additionally, we investigated the individual performance impact of our specific optimizations and design choices, illustrating the advantages afforded by a Concepts-based approach.

Keywords GPU computing · Compile-time expressions · API design · SYCL · C++20

✉ Peter Thoman
peter.thoman@uibk.ac.at

[1] Distributed and Parallel Systems, University of Innsbruck, Technikerstaße 21a, 6020 Innsbruck, Tirol, Austria

1 Introduction

With the end of Dennard scaling [1], the hardware complexity of today's high-performance computing (HPC) systems is now—by necessity—growing along with their computational power. The TOP 500 list [2] shows that many of the most powerful current HPC systems are highly parallel and heterogeneous, consisting of a combination of multi-core CPUs, GPUs and/or accelerators in clusters of interconnected nodes.

Writing efficient applications for such systems is often labor-intensive and prone to error, as it requires the use of specific low-level parallel programming paradigms at node level (e.g., OpenMP [3] or OpenCL [4]), while leaving inter-node communication to libraries such as MPI [5]. Although all of these technologies have evolved over time, their use still limits productivity as the application programmer is responsible for the complexity of scheduling computational tasks and moving data as required.

To deliver higher productivity for scientists and other end-users, a number of high-level, abstract programming models have been proposed. Most of these programming models require the user to locate and specify parallelism or explicitly require user-placed synchronization; examples include UPC [6], Cilk [7], and Chapel [8]. Although static, user-specified schedules and partitionings are common, the increasing complexity of contemporary and future systems encourages automatic tuning support to dynamically optimize the utilization of resources through runtime systems; examples of such dynamic systems supporting heterogeneous distributed memory architectures are *StarPU* [9] and *OmpSs* [10].

A promising HPC programming approach leverages C++ template libraries, which hide the details of the underlying infrastructure from application experts. Implementations of this principle include Kokkos [11] from Sandia National Laboratories and the RAJA portability layer [12]. A lower level of abstraction, which is based on similar technology, is provided by the OCCA library [13].

A particularly interesting option in a similar category is SYCL [14], an industry standard supported by the Khronos group. SYCL provides a higher-level C++ interface to accelerators with broad industry support, but remains constrained to a single node. To overcome this limitation, the Celerity project [15] extends the SYCL approach to accelerator clusters.

However, while Celerity obviates the need for explicit message passing and data decomposition, its SYCL-like interface still assumes that programmers are familiar with concepts such as GPU kernels, task graphs and asynchronous work queues. Modern high-level languages commonly allow the abstract application of sequences of operands and their composition on data, in a manner which specifies only the operations required but does not constrain the implementation of these operations. C++20 *range adaptors* and *views* are one example of this principle.

With the *Celerity high-level API* (Celerity HLA), we introduce a programming interface which provides a very high level of abstraction, inspired by C++ ranges and views, while targeting not just single accelerators or shared memory nodes, but clusters of accelerators. Crucially, we achieve this goal without introducing significant overhead compared to a manual implementation in many benchmark scenarios. Our concrete contributions are as follows:

- The design and implementation of the Celerity High-level API, providing a very high level of abstraction and a functional programming style for targeting clusters of GPUs, which was previously only available for CPUs on single shared-memory nodes.
- Details on our research into various potential sources of overhead and their mitigation using state-of-the-art API design and programming techniques. This includes concept-based typing to eliminate the need for type erasure, as well as compile-time metaprogramming to maximize automatic kernel fusion opportunities.
- A performance evaluation of the Celerity HLA compared to direct low-level implementations, demonstrating the overall applicability of our approach as well as the impact of our overhead mitigation strategies across a set of benchmarks.

The remainder of this paper is structured as follows. In Sect. 2, we provide some background information on SYCL and Celerity, as well as the new C++20 standard library features for working with collections, which served as an inspiration for the Celerity HLA. Section 3 describes our core contributions, including the Celerity HLA API design and implementation featuring concept-based typing and compile-time fusion of operations into single kernels. In Sect. 4 we evaluate our implementation and demonstrate its low performance overhead compared to more verbose options. We also investigate the impact of some specific implementation choices and optimizations. Finally, Sect. 5 discusses some related work and Sect. 6 concludes the paper.

2 Background

2.1 SYCL & Celerity

SYCL is a single-source programming model for heterogeneous computing that builds on pure C++. Unlike other GPU computing options such as CUDA, the language is not extended syntactically, and SYCL programs are valid C++ programs. When accelerators are targeted, a SYCL implementation requires a dedicated compiler that identifies kernels, extracts them, and compiles them into a representation—such as SPIR-V [16]—suitable for a given accelerator. As both kernel and host code are stored in the same source file and have access to the same data structures, SYCL therefore enables C++ features such as templates to work seamlessly and in a type-safe manner *across boundaries of host and device code*. This property is of particular importance for the Celerity HLA.

In SYCL, the execution of data parallel kernels is organized by an implicitly constructed task graph, which is based on data access specifications that a programmer associates with a kernel by constructing *accessor* objects.

Celerity [15] builds upon and extends SYCL—which can be seen as a domain-specific embedded language—to enable execution on distributed memory clusters.

While shared memory parallel kernels are still handled by SYCL on each individual worker node, the Celerity runtime acts as a wrapper around each compute process, transparently handling inter-node communication and scheduling. This is made possible by an asynchronous *multi-pass execution* process, which 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.

Listing 1 Simple Celerity program adding two matrices

```
 1  celerity::distr_queue queue;
 2  const auto extents = sycl::range<2>(512, 512);
 3
 4  celerity::buffer<float, 2> buf_a(hst_a.data(), extents);
 5  celerity::buffer<float, 2> buf_b(hst_b.data(), extents);
 6  celerity::buffer<float, 2> buf_c(extents);
 7
 8  queue.submit([=](celerity::handler& cgh) {
 9      auto one_to_one = celerity::access::one_to_one<2>();
10      auto r_a = buf_a.get_access<acc::read>(cgh, one_to_one);
11      auto r_b = buf_b.get_access<acc::read>(cgh, one_to_one);
12      auto w_c = buf_c.get_access<acc::write>(cgh, one_to_one);
13      cgh.parallel_for<class my_kernel_name>(extents,
14          [=](sycl::item<2> itm) {
15              w_c[itm] = r_a[itm] + r_b[itm];
16          });
17  });
```

Listing 1 illustrates a simple example which adds two matrices in Celerity. Note the use of the simple `one_to_one` access pattern specifier in lines 10 to 12. This information enables the Celerity runtime sytem to correctly associate data with parts of the implicitly distributed kernel [15]. While this code is quite succinct when considering that it is capable of executing on a cluster of GPUs and handling all related distributed memory and accelerator complexities, it is still very verbose compared to a functional specification of the algorithm. This gap in expressiveness motivates our work on the Celerity HLA.

2.2 C++20 Ranges and Views

With C++20, the C++ standard library starts supporting pipelines of operations for a more concise programming style inspired by functional programming. This pattern is widely used in other languages such as C# (LINQ [17]) or Python and allows for more concise viewing and manipulation of collections of elements. Consider an example of discarding all odd values from an integer collection and converting the remaining values to their string representation.

Listing 2 C++ sample, imperative style

```cpp
const std::vector<int> in{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<std::string> out{};

for(auto i : in) {
    if(i % 2 == 1)
        continue;
    out.emplace_back(std::to_string(i));
}
```

Listing 3 C++ sample, imperative style using algorithms

```cpp
std::vector<int> in{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<std::string> out{};

std::ranges::remove_if(in, [](int i){ return i % 2 == 1;});
std::ranges::transform(in, std::back_inserter(out),
                       [](int i){ return std::to_string(i); });
```

Listing 4 C++ sample, functional style in C++20

```cpp
const std::vector<int> in{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto out = in | views::remove_if([](int i){ return i % 2 == 1;})
              | views::transform([](int i){ return to_string(i); });
```

Listing 2 shows a traditional, imperative way of solving this problem. While straightforward to grasp in this simple example, this approach quickly becomes hard to comprehend for more complicated sequences of operations. It also loses semantic information regarding potential parallelism. Therefore, it is usually preferable to use named algorithms from the standard library, making it easier for the reader to follow the individual steps performed and clarifying their semantics.

This solution is implemented in listing 3, demonstrating the effect of using named algorithms on readability and clarity. However, since the computation has been divided into two separate steps, those are performed sequentially, each of them fully iterating over their respective input sequence. Depending on the use case and run time requirements, this may not be a viable approach. These cases then require pulling the condition which filters the odd numbers into the transform functor, which in turn reduces readability again. In the case of a distributed, accelerator-based platform like Celerity, splitting computations in this manner is especially costly—as queuing operations to accelerators on different nodes is relatively expensive—and should be reduced to a minimum.

The functional approach in Listing 4 achieves the optimal run time properties of Listing 2 while being even more concise than Listing 3 by allowing range adaptions to be *composed* and evaluated lazily upon iteration in a single pass. This functional style of processing ranges of elements is yet to be adopted widely in the C++

landscape, but with the standardization in C++20 it is very likely to gain broad adoption in the future. It has been chosen as the model for designing the celerity high-level API because it fits our requirements very well: it provides optimal run time complexity while still being clear and descriptive.

3 Method

The Celerity High-Level API is implemented as an open source C++20 library on top of the Celerity Runtime system.[1] Figure 1 provides an overview of the software stack of an application using the Celerity HLA.

Application code is based on the HLA C++ library, which transparently turns a functional processing pipeline formulation into buffers, kernels and accessor operations understood by the Celerity runtime system.

It in turn manages the launching of kernels and provisioning of memory using an underlying SYCL implementation, and implements communication between nodes via MPI. In this way, the Celerity HLA program is capable of executing on multiple GPUs in a distributed memory system.

Given the additional constraints imposed on GPU kernel code, as well as the requirements for dealing with distributed memory, achieving performance comparable to a low level implementation in Celerity HLA while maintaining programmability and functional composition requires very specific API design choices and researching various implementation alternatives and optimizations.

3.1 API Design

Listing 5 shows a simple Celerity HLA example. In all HLA examples, in the interest of readability, a preamble including `using namespace celerity;` and `distr_queue q;` is assumed. Note that the code closely resembles the C++ ranges and views API illustrated in listing 4 with two additions: first, as with every SYCL kernel submission, a kernel name has to be specified. Therefore, `transform` is passed an explicit template type argument (`<class _1>`). Note that this ceases to be required with SYCL 2020 [18], which further simplifies Celerity HLA usage (i.e. eliminating `<class _1>` in the example). We have chosen to include these classes in the example code presented in this paper to maintain its compatibility with existing SYCL 1.2 implementations.

Listing 5 Celerity HLA: computing squares

```
buffer<int, 1> in{src.data(), {src.size()}};

auto out = in | hla::transform<class _1>([](int i){ return i * i; }
              | hla::submit_to(q);
```

[1] The Celerity HLA is available on GitHub: https://github.com/celerity/celerity-hla.

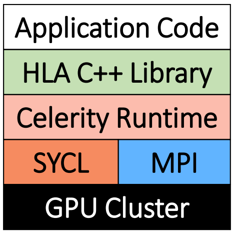

Fig. 1 Overview of the Celerity HLA software stack

As a second addition compared to standard C++, to trigger kernel submission to the Celerity queue, the operation sequence is terminated with `submit_to(q)`. Note that since the lambda expression passed to `transform` takes a simple `int` as parameter, the kernel may only access the current element.

Selecting another Celerity access pattern (i.e. builtin range mapper) is done by changing the parameter type of the kernel functor. This is similar to dependency injection, a technique used in managed languages like Java to request certain services to be passed to a method or constructor by a framework. In Celerity HLA we use the same pattern to tell the run time which portion of the buffer we want to access: a single element, a slice, a block of adjacent elements or all elements of the corresponding input buffer.

The buffer type of `out` is automatically deduced from the submission of the operations sequence, in this case `celerity::buffer<int, 1>`..

Listing 6 shows an example of using `hla::slice` to request slice range mapping of the corresponding input buffers to compute the product of two square matrices. The template arguments of `slice` define the data type on which to operate and the desired dimension of the slice, as illustrated in Fig. 2. Since `hla::accessors` like `hla::slice` provide iterators for traversing their respective ranges, the column-row inner product can be conveniently computed using the `inner_product` algorithm from the standard library.

Listing 6 Celerity HLA: $\alpha(A * B)$

```
 1  buffer<int, 2> in_a{{16, 16}}, in_b{{16, 16}};
 2  constexpr float alpha = 2.0f;
 3
 4  const auto multiply = [](const hla::slice<int, 1>& a,
 5                           const hla::slice<int, 0>& b) {
 6     return std::inner_product(begin(a), end(a), begin(b), 0);
 7  };
 8
 9  auto out = in_a
10           | hla::transform<class _1>(multiply) << in_b
11           | hla::transform<class _2>([=](int i){ return alpha*i; })
12           | hla::submit_to(q);
```

Note how the second buffer is piped in using the stream operator in line 9. This code will result in only a single kernel submission as those two operations are

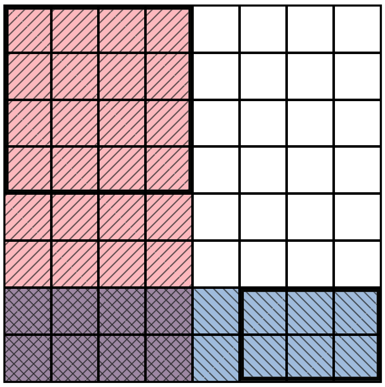
Fig. 2 “Slice” chunk examples on a 2D buffer. Blue: dimension 1, Red: dimension 0 (Color figure online)

suitable for kernel fusion, merging them into one single kernel. We will now elaborate on how this is achieved internally.

3.2 Kernel Fusion and Implementation Optimization

Listing 7 shows an example of an operation sequence with fusible kernels—i.e. kernels which can be merged into a single one, resulting in a single task submission—and non-fusible kernels. Kernels `scale` and `square` can be fused as the latter only reads a single element per work item. However, `square` and `multiply` can not be fused, as `multiply` reads a *slice* of input data, which needs the transformations of `square` (and `scale`) to have already been applied.

Listing 7 Celerity HLA: fusible and non-fusible kernels combined

```
buffer<int, 2> in_a{{16, 16}};
buffer<int, 2> in_b{{16, 16}};
constexpr float alpha = 2.0f;

const auto scale = [=](int i){ return alpha*i; };
const auto square = [](int i){ return i*i; };

const auto multiply = [](const hla::slice<int, 1>& a,
                         const hla::slice<int, 0>& b) {
   return std::inner_product(begin(a), end(a), begin(b), 0);
};

auto out = in_a
         | hla::transform<class _1>(scale)
         | hla::transform<class _2>(square)
         | hla::transform<class _3>(multiply) << in_b
         | hla::submit_to(q);
```

Figure 3 illustrates the process of building the final, partially fused operation sequence. The first step is to package up the kernel functors and all available

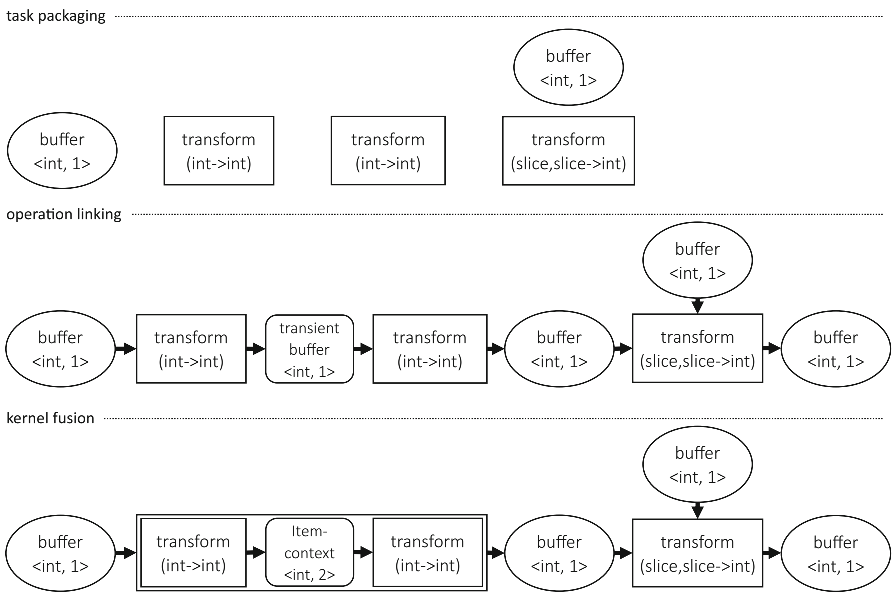

Fig. 3 Processing steps for the operation sequence in listing 7

compile-time information. These are crucial for the following stages and serve as a decision-making basis. After the tasks have been built, they are sequenced by using the pipe operator | and secondary input buffers are streamed in using <<.

When the full sequence is built, the *operation linking* phase starts. Here the decision whether two kernels are eligible for fusion is made. This decision depends on the access pattern of the latter task. Thus, the decision boils down to the question if the second task has loop-carried dependencies or not. If it does, those tasks can not be fused, otherwise they can be. Now, it might seem difficult to extract this information from the kernel with a library-based approach which does not allow analysing kernels on a source level. However, interestingly, this information is already available to the Celerity runtime, although in a different form. Recall that the Celerity runtime requires users to specify which data ranges they plan to access in the form of range mappers. This is needed to determine which parts of buffers need to be available on each distributed node. The `one-to-one` range mapper indicates that in every iteration step only the current element is accessed which means loop-independent dependence, while all other range mappers allow accessing elements other than the currently iterated one and thus describe loop-carried dependencies. Consequently, knowing which range mappers are used inside a kernel suffices to decide fusibility. Conveniently, the Celerity high-level API handles the selection of range mappers and thus already has that information.

In short, fusibility of two unary kernels is decided by considering the requested range mapper of the second kernel - if and only if it is a one-to-one mapping, the kernel can be fused with its predecessor.

For binary kernels (which have two predecessors) the decision of fusing them or not has to be evaluated for both inputs in isolation. If the first input is accessed in a loop-independent manner then the respective predecessor can be merged into the binary kernel. The same applies to the the second input, which results in the following final decision algorithm: for each input of the kernel, examine its range mapper and if and only if it is an one-to-one mapping, merge the corresponding predecessor kernel into the one in question while preserving the sequence of operations.

In case two adjacent kernels can be fused, they are linked through a *transient buffer* which indicates that there is no actual buffer-like storage required. Otherwise, the operations are connected by creating a temporary Celerity buffer acting as the sink for the first kernel and source for the second. At the end of this phase, the final buffer which will hold the result of the sequence is created and linked as a sink to the last operation.

Finally, *kernel fusion* is performed. In this pass, for each task pair which is connected via a transient buffer a new task is created containing the merged kernel functor of those two tasks. Internally, the two kernel functors communicate through an `item_context` which acts as a kernel-local, shared storage. In the case depicted in Fig. 3, the first kernel retrieves its input from the first input buffer and writes its result to the `item_context`. Subsequently, the `item_context` serves as input to the second kernel which in turn writes its output to the temporary Celerity buffer created earlier (see Listing 8).

Listing 8 Celerity HLA: unary kernel fusion scheme

```cpp
template<typename ValueType, int BufferRank>
class item_context { ... };

const auto kernel_a = [](item_context<int, 1>& ctx){ ... };
const auto kernel_b = [](item_context<int, 1>& ctx){ ... };

const auto kernel_ab = [=](item_context<int, 1>& ctx)
{
    item_context<int, 1> ctx_a;
    ctx_a.set_input(ctx.get_input());
    kernel_a(ctx_a);

    item_context<int, 1> ctx_b;
    ctx_b.set_input(ctx_a.get_output());
    kernel_b(ctx_b);

    ctx.set_output(ctx_b.get_output());
};
```

3.2.1 From Type Erasure to Concepts

Task packaging presented implementation challenges, mainly because of the limited possibilities of type erasure required to implement HLA accessors. Type erasure is needed to hide the actual Celerity accessor behind the user-visible HLA accessor classes. Since virtual functions and function pointers are disallowed inside SYCL kernels, the only possible way—while keeping a succinct interface—was to turn the compile-time type information of the Celerity accessor not present in the HLA accessor (e.g. `cl::sycl::access::mode` and `cl::sycl::access::target`) into run time information and reconstruct the type using multiple switch statements. Our initial hope was that the compiler could eliminate the related branching in most cases, but as detailed in Sect. 4.3, and confirmed by inspection of the generated PTX code, this was not the case.

Consequently, with C++20 compiler support maturing, a completely different approach leveraging concepts was explored. Instead of defining a concrete type as parameter for the kernel functor, a concept is specified which constraints the set of types which are accepted. Conceptually, this is the same as having a template function and using `std::enable_if` to impose restrictions on the template parameter.

Listing 9 C++17 `std::enable_if` vs C++20 concept

```
// C++17
template<class T, typename = std::enable_if_t<std::is_integral_v<T
    >>>
void foo(T x) {}

// C++20
void moo(std::integral auto x) {}
```

In Listing 9 we can see two functions, each constrained to accept only integral types (e.g. `int`). The first one uses the traditional way of employing `std::enable_if_t` in conjunction with `std::is_integral_v`. In this case, `std::enable_if` is a template meta-function which has the effect of removing the function `foo` from the overload set at a given call site if the template parameter `T` is instantiated with a concrete type which does not fulfill the stated condition (`std::is_integral_v<T>`). Detailing the mechanisms involved in this process goes beyond the scope of this work, and we would like to refer readers to the existing literature on template metaprogramming [19].

The second function uses C++20 *concepts* and the standard-supplied concept `std::integral` to achieve the same goal. These concepts can be thought of as named sets of constraints. In this case, the parameter has only a single constraint but even such a simple example clearly illustrates how concepts can improve readability and conciseness. Additionally, since concepts are a language construct, these requirements can be checked more easily by the compiler which in turn can generate clearer error messages. The most significant advantage of concepts for our work in this paper, however, comes from the fact that they can be succinctly used in lambda expressions (see listing 10). By constraining the parameter types of the kernel functor, we can specify how we want to access data. For example, we can create a *slice* concept to signal that we need a type that allows us to access a slice of data.

Listing 10 Constrained lambda expressions: `std::enable_if` vs concepts

```
// constrained lambda using std::enable_if
auto foo =
  []<class T,
    typename = std::enable_if_t<std::is_integral_v<T>>>(T x) {};

// C++20 constrained lambda using concepts
auto moo = [](std::integral auto x) {};
```

Listing 11 Celerity HLA: $\alpha(A * B)$ using the concepts API

```
distr_queue q;
buffer<int, 2> in_a{{16, 16}};
buffer<int, 2> in_b{{16, 16}};
constexpr float alpha = 2.0f;

const auto multiply = [](hla::Slice auto a, hla::Slice auto b){
    a.configure(1);
    b.configure(0);
    return std::inner_product(begin(a), end(a), begin(b), 0);
};

auto out = in_a
         | hla::zip<class _1>(multiply) << in_b
         | hla::transform<class _2>([=](int i){ return alpha*i; })
         | hla::submit_to(q);
```

Listing 11 shows the concepts variant of listing 6. Here, the multiply kernel takes two `auto` parameters which are both constrained using the `hla::Slice` concept. Passing this functor to an algorithm basically tells the runtime the following: "Provide me with some type that satisfies the `Slice` concept and thus contains a Celerity accessor with slice range-mapping.". So, rather than requesting an explicit type we specify what features the requested type has to support. This allows the runtime to provide the kernel with a type that holds the required Celerity accessor as-is, without any type erasure involved. Performance-wise this means no branching and hence no diverging instruction paths. However, from runtime perspective, detecting which concept was specified is significantly more involved than with the type erasure approach. With type erasure, detecting the accessor type boils down to checking if the parameter type equals a certain type. With concepts, the library now has to probe the kernel functor with a fixed set of types. Depending on the compile-time information for which of those types the kernel functor is invocable, it can infer the access concept and create the matching Celerity accessor. This approach

Table 1 Specification of the benchmarking system

Host	AMD Ryzen Threadripper 2920X 12-Core, 32 GB DDR4 RAM
GPUs	4× Nvidia RTX 2070
Software	Ubuntu 20.04; OpenMPI 4.0.0; GPU driver 460.32.03; hipSYCL 0.9; Celerity 0.2.1

simplifies the function signature of the kernel functor even further by discarding the data type from the accessor template parameters. However, since there is no way of specifying the dimension of the slice in the concept type, this configuration has to be provided via a `configure(dimension)` method inside the kernel functor.

Note how in listing 11 the range adaptor which takes two input buffers is now created using `hla::zip` instead of `hla::transform`.. This is necessary because it is not possible to determine the arity of a function template without having valid input argument types. Since the function call operator of the kernel functor with its concept-constrained input types is a function template, this also applies here. This in turn means that at the time the operation is packaged there is no way of telling if the kernel functor takes one or two arguments. Thus we need to discern between transformation (one input) and zipping (two inputs) at the call site using two distinct function templates.

4 Evaluation

The significant advantages in terms of code succinctness, readability, and composition conferred by the Celerity HLA would be relatively meaningless if they came at a large general loss in performance potential. Therefore, we provide a set of benchmarks in this section demonstrating the performance of our approach, and analyzing the impact of our design choices and optimizations.

Table 1 summarizes the hardware of our benchmarking machine, as well as the software stack used for this evaluation. We used various benchmarks from the PolyBench GPU suite, which was previously ported to low-level Celerity,[2] and which we ported to idiomatic Celerity HLA for this work.

The results presented are based on the median of 5 benchmark runs for each configuration, and there was no significant cross-run variance.

4.1 General Performance

Figure 4 illustrates the relative execution time of the Celerity HLA version of each benchmark compared to a low-level baseline—that is, the existing Celerity versions with explicit kernel invocations and accessor management by the application programmer. Three categories of results are visible: for the largest group including the *atax*, *bicg*, *covar*, *gesumv* and *mvt* benchmarks, the relative difference in execution time between both versions is 1% or less. We can consider these results

[2] https://github.com/bcosenza/celerity-bench/tree/master/polybench.

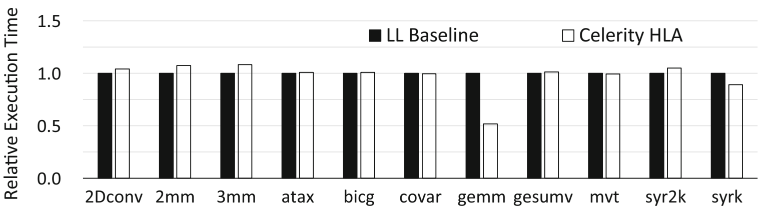

Fig. 4 Performance comparison between Celerity HLA and a low-level baseline

practically identical, indicating that for these benchmarks the higher level of abstraction and succinctness of the Celerity HLA comes at no performance cost.

The *2DConv*, *2mm*, *3 mm* and *syr2k* benchmarks form the second group. Here, the implementations based on the Celerity HLA are measurably and consistently slower, between 4% for *2DConv* and 8% for *3 mm*. These differences can be explained by backend code generation overheads—related to the additional level of indirection introduced—which we were not able to fully eliminate in the current version of the Celerity HLA. However, this overhead remains below 10% in all cases.

Finally, *syrk* and *gemm* both show significantly better performance in their HLA versions than in the baseline implementation. After investigating the unusually large performance differential, we discovered a performance bug in the baseline: it specifies the output buffer access as `read_write` even though it could be rewritten to only require `discard_write`.

Of course, as is the nature of low-level APIs, the baseline versions of both of these benchmarks could be rewritten to perform as well or better than the HLA version. However, we believe it is interesting and noteworthy that the automatic mechanisms in the Celerity HLA can help uncover inefficiencies in manual, lower-level implementations in this fashion, without requiring manual intervention by developers.

Overall, general performance remains within an acceptable margin across the entire set of benchmarks, given the significantly higher level of abstraction provided by Celerity HLA.

4.2 Kernel Fusion

The performance impact of kernel fusion varies with the pipeline structure of any given application—operator pipelines which have no fusible kernels are unaffected—as well as its problem size and memory access behaviour. To illustrate the dependence on problem size, Fig. 5 shows a comparison between the standard Celerity HLA implementation, and a version in which kernel fusion was manually disabled, for the *gemm* benchmark containing two fusible kernels.

At smaller problem sizes, performance is not significantly affected by kernel fusion, but at the largest tested problem size fusion improves the throughput achieved by 26%.

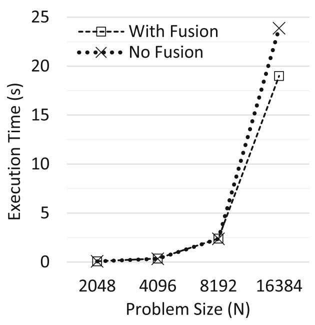

Fig. 5 *gemm* kernel fusion

4.3 Concepts-based Design vs. Type Erasure

In Sect. 3.2 we outlined the reasoning, implementation and consequences of the final, C++20 concepts-based, HLA implementation. Figure 6 provides a quantitative perspective on this issue.

The first fundamental result visible is that the concepts-based version outperforms the version based on type erasure for all benchmarks. However, the actual performance difference induced varies widely based on the properties of a given benchmark. Some, such as *2DConv*, are primarily memory bandwidth limited, and as such the execution time impact associated with type erasure is relatively small. For the specific *covar* case, the backend compiler can actually statically detect the branches taken, resulting in almost identical performance.

Finally, for *2* and *3 mm*, the compiler is incapable of moving the additional control flow for dealing with type erasure out of the innermost computational loops, which also prevents any further optimizations, and results in a massive performance differential. Overall, these results illustrate that a concepts-based API which makes dispatch decisions at compile time is essential for broad applicability.

5 Related Work

The introduction already covers some related work necessary to establish the larger context of the Celerity HLA, including low-level accelerator and networking APIs such as CUDA, OpenCL, OpenMP and MPI, as well as popular runtime systems and abstraction layers such as StarPU, OmpSs, Kokkos and RAJA. In this section, we

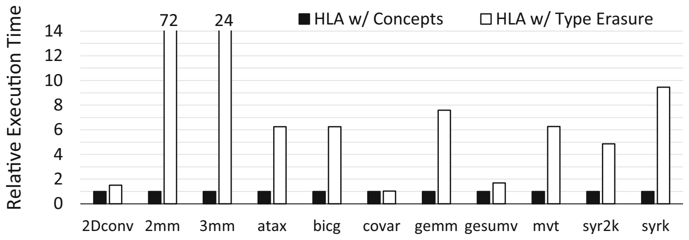

Fig. 6 Comparing HLA implementation versions based on concepts and type erasure

want to focus on three additional types of related work: those which introduce techniques leveraged in the design and implementation of the Celerity HLA, skeleton-based libraries and frameworks, and some additional projects which are less widely used but closer in nature to our work than the popular APIs covered in Sect. 1.

In the first category, the history of C++ expression templates is of particular note. Introduced by Haney et al. [20], they were developed into one of the most powerful tools for providing very high levels of abstraction with minimal overhead [21], with Chen et al. initially applying these methods to GPU computing using CUDA [22].

Esterie et al. [23] introduced a numerical template toolbox based on an architecture-aware domain engineering method for reusable algorithmic libraries. Their library NT^2 follows a substantially different interface design philosophy compared to Celerity HLA, with the goal being relatively simple porting from Matlab code, while our API design seeks to match the expectations of and appear familiar to C++20 programmers.

Expression templates are frequently used in concert with skeleton-based APIs in order to provide a high level of abstraction in C++ EDSLs. Early implementations of this general concept include the Quaff library by Falcou et al. [24], as well as the Orléans Skeleton Library developed by Noman and Loulergue [25].

Matsuzaki et al. [26] developed a parallel skeleton library for distributed memory environments which supports kernel fusion for list skeletons via an expression template mechanism, but this work was not targeting GPUs or accelerators. Shigeyuki et al. [27] provided an early application of algorithmic skeletons to GPU computing. As was common at the time of this work, only CUDA-based GPU platforms are supported, and the API is still lower level than more recent approaches. More recently, Ernstsson et al. [28] developed an extension to the SkePU skeleton programming model which lazily records the lineage of skeleton invocations, and applies tiling once partial results are actually required by the program.

Several of these approaches feature optimizations which are similar in principle to the kernel fusion performed by Celerity HLA, with one difference being that our heterogeneous SYCL target presents some complications—as outlined previously—for a pure runtime system. Additionally, our work also builds upon modern advances in C++ in terms of API design as well as implementation, and applies to distributed memory clusters of multiple GPUs using a vendor-agnostic interface rather than only targeting CPUs or individual GPUs.

Looking beyond the rich history of expression templates and skeleton-based APIs, we will now discuss some projects which are closer to Celerity HLA in terms of target platforms and underlying GPU computing technologies.

Huynh et al. [29] presented a high-level framework targeting streaming applications which maps these applications to multiple GPUs. Their work focuses specifically on the partitioning and communication challenges involved in this type of application. Crucially, they focus on a set of GPUs connected to a single host, while the Celerity HLA system is designed for leveraging distributed memory clusters of GPUs using an underlying MPI layer.

PHAST [30] is a wrapper library which, at first glance, seems to be positioned very closely to the Celerity HLA. It also seeks to provide a C++ interface to GPUs using STL-like algorithms and iterators, and maps to multiple lower-level backends. However, there are several significant differences: from an API perspective, PHAST—like most mainstream technologies covered in the introduction, and unlike skeleton-based approaches—is based on a procedural formulation of algorithms, while our work introduces a functional pipeline composition approach; in terms of implementation, we focus particularly on enabling automatic kernel fusion; and in terms of target platforms, PHAST is limited to single GPUs while the Celerity HLA can transparently target multiple GPUs.

The same comparative analysis holds concerning other existing STL-on-GPUs implementations, such as those based on SYCL [31]. As we demonstrate in our benchmarks, providing a composition-based functional API necessitates careful type treatment to achieve performance comparable to lower-level procedural implementations.

6 Conclusion and Future Work

We have presented the Celerity High-level API, which, for the first time, enables the development of applications for accelerator clusters using composable functional operator pipelines. As demonstrated in our evaluation, we achieve this very high level of abstraction without substantially compromising performance: most benchmarks perform almost identically to a lower-level implementation, and the worst cases of overhead remain below 10%.

These results are enabled by two key features: a concept-based API design allowing for concise syntax while maintaining compile-time dispatch, and an automatic kernel fusion procedure which eliminates temporary buffers and merges separate kernel invocations whenever possible.

We are aware of two main avenues for future work. Firstly, kernel code is highly sensitive to register assignment by the compiler, which currently limits kernel fusion performance gains to larger problem sizes. Consequently, there is ongoing work to simplify the `item_context` portion of the fusion algorithm which should reduce run-time overhead further. Secondly, HLA accessor iterators induce some overhead when iterating multi-dimenstional ranges such as 2D stencils or 3D voxels. Improvements in the iterator facilities could help bring Celerity HLA to performance parity with manual low-level implementations in even more cases.

Funding Open access funding provided by University of Innsbruck and Medical University of Innsbruck.

References

1. Bohr, M.: A 30 year retrospective on Dennard's mosfet scaling paper. IEEE Solid-State Circuits Soc. Newslett. **12**(1), 11–13 (2007)
2. The Top 500 List (2020). http://www.top500.org
3. Dagum, L., Menon, R.: Openmp: an industry standard API for shared-memory programming. Comput. Sci. Eng., IEEE **5**(1), 46–55 (1998)
4. Group, K.O.W.: The opencl specification, version 2.0. Technical report (2014)
5. Forum, M.P.I.: MPI: a message-passing interface standard. Technical Report, Knoxville, TN, USA (1994)
6. Carlson, W.W., Draper, J.M., Culler, D.E.: Introduction to UPC and language specification. Technical Report (1999). https://www.researchgate.net/publication/228734882_Introduction_to_UPC_and_language_specification
7. Blumofe, R.D., Joerg, C.F.: Cilk: an efficient multithreaded runtime system. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP), pp. 207–216 (1995)
8. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
9. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In: Euro-Par Parallel Processing (2009)
10. 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 Process. Lett. **21**(2), 173–193 (2011)
11. Edwards, H.C., Trott, C.R.: Kokkos: enabling performance portability across manycore architectures. In: Extreme Scaling Workshop (XSW), pp. 18–24. IEEE (2013)
12. Hornung, R., Keasler, J., et al.: The Raja Portability Layer: Overview and Status. Lawrence Livermore National Laboratory, Livermore (2014)
13. Medina, D.S., St-Cyr, A., Warburton, T.: Occa: a unified approach to multi-threading languages. arXiv preprint arXiv:1403.0968 (2014)
14. Group, T.K.: SYCL Specification, Version 1.2.1 Revision 3. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf. Accessed 05 Feb 2019
15. Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: high-level c++ for accelerator clusters. In: European Conference on Parallel Processing, pp. 291–303. Springer (2019)
16. Kessenich, J., Ouriel, B., Krisch, R.: Spir-v specification. Khronos Group **3**, 17 (2018)
17. Marguerie, F., Eicherte, S., Wooley, J.: LINQ in Action. Manning Publications Co, New York City (2008)
18. Reyes, R., Brown, G., Burns, R., Wong, M.: Sycl 2020: More than meets the eye. In: Proceedings of the International Workshop on OpenCL, p. 1. Association for Computing Machinery, New York (2020)
19. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Pearson Education, London (2004)
20. Haney, S.W., Dubois, P.F.: Beating the abstraction penalty in C++ using expression templates. Comput. Phys. **10**(6), 552–557 (1996)
21. Iglberger, K., Hager, G., Treibig, J., Rüde, U.: Expression templates revisited: a performance analysis of current methodologies. SIAM J. Sci. Comput. **34**(2), 42–69 (2012)
22. Chen, J., Joo, B.: Automatic offloading C++ expression templates to CUDA enabled GPUS. In: IEEE 26th International Parallel and Distributed Processing Symposium Workshops, pp. 2359–2368. IEEE (2012)
23. Esterie, P., Falcou, J., Gaunard, M., Lapresté, J.-T., Lacassagne, L.: The numerical template toolbox: a modern C++ design for scientific computing. J. Parallel Distrib. Comput. **74**(12), 3240–3253 (2014)
24. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Comput. 32(7), 604–615 (2006). Algorithmic Skeletons

25. Javed, N., Loulergue, F.: Parallel programming and performance predictability with Orléans skeleton library. In: 2011 International Conference on High Performance Computing and Simulation, pp. 257–263 (2011)
26. Matsuzaki, K., Emoto, K.: Implementing fusion-equipped parallel skeletons by expression templates. In: Morazán, M.T., Scholz, S.-B. (eds.) Implementation and Application of Functional Languages, pp. 72–89. Springer, Berlin, Heidelberg (2010)
27. Sato, S., Iwasaki, H.: A skeletal parallel framework with fusion optimizer for GPGPU programming. In: Hu, Z. (ed.) Programming Languages and Systems, pp. 79–94. Springer, Berlin, Heidelberg (2009)
28. Ernstsson, A., Kessler, C.: Extending smart containers for data locality-aware skeleton programming. Concurr. Comput.: Pract. Exp. **31**(5), 5003 (2019)
29. Huynh, H.P., Hagiescu, A., Wong, W.-F., Goh, R.S.M.: Scalable framework for mapping streaming applications onto multi-GPU systems. SIGPLAN Not. **47**(8), 1–10 (2012)
30. Peccerillo, B., Bartolini, S.: Phast-a portable high-level modern C++ programming library for GPUS and multi-cores. IEEE Trans. Parallel Distrib. Syst. **30**(1), 174–189 (2018)
31. Copik, M., Kaiser, H.: Using sycl as an implementation framework for hpx compute. In: Proceedings of the 5th International Workshop on OpenCL (2017)