

Kokkos::Graph: A Performance-Portable Abstraction for Directed Acyclic Graphs of Asynchronous Operations

Romin Tomasetti^a, Damien Lebrun-Grandié^b, Christian Trott^c,
Maarten Arnst^{a,*}

^a*University of Liège, allée de la Découverte 9, 4000, Liège, Belgium*

^b*Los Alamos National Laboratory, Bikini Atoll Rd, 87544, Los Alamos, New Mexico, US*

^c*Sandia National Laboratories, PO Box 5800/MS 1318, 87185, Albuquerque, New Mexico, US*

Abstract

Workloads structured as directed acyclic graphs (DAGs) of asynchronous operations arise in many high-performance computing applications. This paper presents `Kokkos::Graph`, the graph abstraction of the `Kokkos` performance portability library for expressing and orchestrating the execution of such DAG-structured workloads. It represents a static DAG defined ahead of submission and maps to corresponding vendor graph implementations on the CUDA, HIP and SYCL backends. Although their definition and instantiation incur an upfront cost, these backend graphs are designed so that they may be repeatedly submitted for execution with lower scheduling overhead than enqueueing the operations individually onto streams or queues, thus allowing the one-time costs to be amortised over multiple submissions. After contrasting the graph abstraction with the well-established execution space abstraction, this paper describes recent improvements and extensions to `Kokkos::Graph` that resolve underspecification issues in its execution and data-lifetime semantics and broaden its feature set. Then, the performance is compared across backends using two metrics: the asymptotic speedup in the limit as the one-time costs are fully amortised, and the break-even number of submissions. The asymptotic speedup is consistently greater than one across the full set of benchmarks only on the CUDA

*Corresponding author

backend, although multiple submissions are still needed to reach the break-even point. On CUDA, HIP and SYCL alike, the asymptotic speedup tends to one as the work per operation grows and the scheduling overhead thus becomes negligible relative to the operation execution time. Finally, a concrete application to finite element assembly highlights several code-design benefits of adopting `Kokkos::Graph`.

Keywords: Kokkos, performance portability, directed acyclic graph, asynchronous execution, graph API, GPU

1. Introduction

Many application codes in high-performance computing (HPC) execute asynchronous operations that can be organised as a directed acyclic graph (DAG): operations are nodes, while directed edges encode precedence constraints, *i.e.*, ordering requirements that subsume, but are not limited to, data dependencies. Application developers are therefore faced with two distinct challenges: on the one hand, writing efficient kernels, and, on the other hand, determining an efficient scheduling strategy for the DAG. While kernel optimisation is inherently application-specific, achieving an optimal orchestration of the execution of a DAG can be challenging [1, 2, 3]. Importantly, conflating application-specific code with mechanisms for orchestrating the execution may hinder code readability and maintainability. Describing the DAG through a dedicated abstraction that delegates the orchestration of the execution to a specialised software component provides an appropriate separation of concerns.

Developers are faced with a third challenge: the modern HPC landscape has evolved towards heterogeneous computing platforms, whose hardware and software stacks evolve rapidly. Delivering and maintaining high performance across platforms can be achieved with portability abstractions, such as those of RAJA [4], SYCL [5], alpaka [6], or Kokkos [7, 8], which allow developers to write single-source vendor-agnostic library and application code. This work uses Kokkos, a performance portability library for on-node parallelism written

in modern C++. It provides abstractions for both parallel execution and data structures, leveraging template specialisations to map to a user-chosen lower-level vendor programming model, referred to as the *backend*.

Kokkos provides two abstractions for scheduling asynchronous operations: the *execution space* abstraction and the *graph* abstraction. The execution space abstraction is the primary abstraction in **Kokkos**. Operations are submitted to a device-bound queue and executed eagerly in first-in first-out (FIFO) order. Independent operations can be submitted to distinct execution space instances, and precedence constraints beyond queue ordering can be implemented by introducing explicit synchronisation points. An execution space instance maps to a corresponding vendor queue, for instance, a CUDA stream on the CUDA backend. The graph abstraction was introduced in **Kokkos** 3 [8] and remains experimental as of **Kokkos** 5.1 [9]. It represents a significant shift: rather than dispatching operations individually, the entire DAG is described upfront, *ahead of submission*. The resulting graph is then instantiated into an executable graph, which may be submitted repeatedly. The responsibility for orchestrating the execution of the DAG is thereby transferred to the underlying graph implementation. The main computational benefits currently targeted by the graph abstraction are reduced kernel launch overhead and improved device utilisation.

Kokkos::Graph maps to vendor implementations on the CUDA, HIP and SYCL backends (see Table 1). On the CUDA backend, it leverages CUDA graph, introduced by NVIDIA as part of CUDA 10 to address CPU launch overhead issues in applications that repeatedly launch many small GPU operations from the CPU. By bundling a group of GPU operations into a single CUDA graph launch, such applications benefit from reduced CPU launch overhead and from CPU resources being freed to perform overlapping work. The resulting performance gains provided by CUDA graph have been extensively studied [10, 11] and have motivated its adoption in machine learning frameworks, such as PyTorch [12, 13], as well as in computational physics codes, like GROMACS [14]. AMD and Intel later introduced their own graph implementations, HIP graph and SYCL command graph, respectively. CUDA graph continues to evolve,

with ongoing developments towards system-wide multi-GPU graphs, as well as new types of spatial and temporal execution guarantees [15]. Surfacing the capabilities of the vendor graph implementations in a performance-portable way within `Kokkos` is thus an important development direction.

Although the graph abstraction was introduced in `Kokkos 3` [8], it has not previously been described in detail in the literature, nor exercised extensively in an application. The contributions of this paper are as follows. First, this paper provides an in-depth description of the `Kokkos::Graph` programming model. Second, it brings a series of improvements and extensions to `Kokkos::Graph`, which have been integrated into `Kokkos` during the `Kokkos 3` to `Kokkos 5.1` release cycle: it resolves several underspecification issues in the API semantics across the graph object lifecycle, such as the interaction with the execution space abstraction, user data lifetime and device selection; and it introduces new features that broaden the class of DAGs expressible within the graph abstraction, such as support for graph capture and the *then* node. Third, it demonstrates the usability of `Kokkos::Graph` through a portable single-source implementation of several inherently iterative workflows, such as a conjugate gradient (CG) solver. This single-source code is then used to provide insight into the performance across multiple backends. Finally, this paper reports on a use of `Kokkos::Graph` in a representative finite element (FE) assembly application.

This paper is organised as follows. Section 2 describes the execution space and graph abstractions. Then, Section 3 describes the resolution of the underspecification issues and Section 4 describes the new features. Next, Section 5 provides the performance evaluation. Subsequently, Section 6 discusses the benefits and challenges of using `Kokkos::Graph` in a real-world codebase, drawing from the authors’ experience integrating it into an in-house FE code. Finally, Section 7 and Section 8 provide concluding discussion.

2. Asynchronous execution in `Kokkos`

Harnessing modern heterogeneous platforms requires asynchronous programming models to fully exploit available execution resources. Launching an oper-

ation asynchronously allows the caller to perform overlapping work while the operation is in flight. Such asynchronous launches allow launch latency to be hidden and independent operations, such as computations and memory transfers, to potentially run concurrently.

Kokkos typically expresses computational operations as triplets of a parallel pattern, an execution policy and a functor (the actual work) [8]. These triplets can be submitted either to an *execution space instance* or as part of a *graph*. The following section describes these two asynchronous programming model abstractions in more detail.

[Table 1 about here.]

2.1. Execution space

The execution space programming model abstracts an asynchronous device-bound FIFO queue into which operations are enqueued. It encapsulates both an execution resource and its associated lower-level programming model [7, 8] (see Table 1).

An execution space instance *eagerly enqueues* operations: any call to a **Kokkos** API function that creates an asynchronous operation immediately enqueues it into the backend queue and returns. An instance may be synchronised (*fenced*), blocking the caller until all enqueued operations complete (see Listing 1). Enqueueing operations into distinct instances enables, but does not guarantee, concurrent execution. The actual degree of concurrency is determined by the execution resources needed for the enqueued operations and the hardware and software capabilities [16, 17].

[Listing 1 about here.]

To account for precedence constraints, the execution space abstraction in **Kokkos** offers two mechanisms: (i) implicit serialisation through enqueueing into the same instance, and (ii) explicit fences to synchronise outstanding operations. Here, it should be noted that CUDA, HIP and SYCL also provide

an event-based API to express cross-queue dependencies: events act as markers that allow operations in one queue to wait for completion of operations in another, without involving the caller. `Kokkos` does not currently surface a corresponding abstraction; however, one has been proposed and is under development for inclusion in a future release [18]. Therefore, cross-queue dependencies must currently be expressed via fences.

Listing 2 shows how the execution space abstraction can express a diamond-shaped dependency pattern (see Fig. 1). Two instances are created by partitioning the default instance, so that operations B and C may potentially execute concurrently. Since B and C both depend on A, instance `exec_0` must be fenced after enqueueing A, ensuring A has completed before B and C are enqueued into their respective instances. Without this fence, the diamond dependency is not preserved. Figure 2 shows a possible execution timeline with launch overheads and concurrent execution of B and C. The fences entail additional overhead and occupy the caller.

[Figure 1 about here.]

[Listing 2 about here.]

[Figure 2 about here.]

Although the involvement of the caller in orchestrating the execution may hinder the potential for overlapping host/device work and reduce code readability and maintainability, certain coding patterns common in user code rely on this involvement. Indeed, the eager enqueueing semantics of the execution space programming model and the ability to synchronise outstanding operations through fencing allow the caller to observe the result of an operation before enqueueing a dependent operation, and, more generally, to interleave arbitrary host work between device operations. For instance, a `parallel_reduce` may count elements satisfying a predicate, with the count then providing the extent of the iteration space of a subsequent `parallel_for`. Likewise, a `Kokkos::View` may be resized between two device operations. Such coding patterns do not

readily extend to the graph abstraction, in which execution is deferred and the operations, together with their execution parameters, must be defined ahead of submission, as discussed below.

2.2. Graph

The graph programming model abstracts an ensemble of operations organised as a DAG that can be submitted as a whole. The graph lifecycle is structured around three stages:

1. The **definition** stage consists in constructing a `Kokkos::Graph` object, whose nodes and edges describe the operations and their precedence constraints.
2. The **instantiation** stage locks the graph: no further nodes or edges can be added. During this phase, the graph is validated and preparatory work for submission may be carried out.
3. The **submission** stage enqueues the whole graph as a single operation into an execution space instance for asynchronous execution. This stage may be repeated multiple times.

`Kokkos::Graph` is a templated type that, when specialised for a given backend, maps to the corresponding vendor graph implementation (see Table 1) or falls back to a default implementation. Its three-stage lifecycle mirrors that of CUDA, HIP and SYCL graphs: definition maps to inserting nodes into the vendor graph, instantiation produces a vendor executable graph object, and submission executes it.

During the definition stage, each new operation is added to the graph from a single predecessor (Listing 3). An operation with multiple predecessors can be expressed by first joining the predecessors via the `when_all` primitive, which produces a single aggregate node that can then be used as the predecessor. If supported by the vendor graph, `Kokkos::Graph` may span multiple devices. A device handle may be used to select for each node the device where the node operation must be executed (Section 3.2).

Upon submission, `Kokkos::Graph` enqueues the execution of the graph as a whole into the execution space instance passed to `submit`, thus ordering graph execution with respect to other operations on that instance. The underlying graph implementation is free in orchestrating the execution of nodes, provided that it observes this execution space ordering for the graph as a whole, the precedence constraints for the nodes, and the per-node device selection; conversely, the graph abstraction leaves unspecified which underlying queues, if any, are used to execute the nodes. In particular, the vendor graph implementations expose no control over the queues, if any, that they may order the execution of the nodes into. This consideration shaped the evolution of the API toward using a device handle (Section 3.2).

The definition and instantiation stages of the graph are one-time costs. In return, once these stages are complete, the scheduling overhead of submitting the executable graph can be expected to be lower than the cumulative overhead of scheduling the operations individually with the execution space abstraction. This benefit may stem from two factors. On the one hand, submitting a vendor graph can be expected to incur less scheduling overhead than enqueueing individual operations into one or more vendor queues; it should be noted, however, that Section 5 shows that such a speedup is currently consistently attained only on the CUDA backend. On the other hand, `Kokkos` adds overhead: whereas submitting a `Kokkos::Graph` is only a thin wrapper around the launch of the underlying vendor executable graph, enqueueing an operation into an execution space instance costs more than enqueueing it directly into the vendor queue (*e.g.*, acquiring locks, launch parameter calculation).

`Kokkos::Graph` currently supports computational operations corresponding to the `parallel_for` and `parallel_reduce` constructs, as well as the `then` primitive, both on host and on device (Sections 4.1 and 4.2). As of `Kokkos` 5.1, restrictions still apply on the use of `parallel_reduce` in `Kokkos::Graph`: indeed, because of implementation limitations, `Kokkos::Graph` does not currently support reduction operations that may interfere, *i.e.*, neither is a topological ancestor of the other [19]. Moreover, `Kokkos::Graph` does not currently support

the `parallel_scan` construct.

`Kokkos::Graph` targets a common subset of vendor graph capabilities, prioritising portability. For instance, among the vendor graph implementations, only CUDA graph provides conditional `while` nodes; `Kokkos::Graph` therefore does not surface this capability. `Kokkos::Graph` provides interoperability with vendor graph API to allow vendor-specific features to be used (Section 4.4).

On backends without a vendor graph implementation, `Kokkos::Graph` falls back to a default implementation built on the execution space abstraction. This implementation is currently deliberately simple and may evolve. Instantiation performs no preparatory work. Upon each submission, the DAG is traversed from its sinks, *i.e.*, nodes without successors, with each node recursively dispatching its predecessors before its own operation. Fences are inserted to observe precedence constraints between operations enqueued into distinct execution space instances.

Listing 3 illustrates how the graph abstraction can express the diamond-shaped dependency pattern from Fig. 1. The definition stage highlights how `Kokkos::Graph` enables an explicit description of the DAG. Figure 3 shows a possible execution timeline. The definition, instantiation and submission occupy the caller. Once the graph is submitted, the caller may perform overlapping work while the graph executes asynchronously.

[Listing 3 about here.]

[Figure 3 about here.]

Certain coding patterns that can be expressed in the execution space programming model are not currently supported by `Kokkos::Graph`. Indeed, the deferred execution semantics, and, in particular, the requirement that the operations, together with their execution parameters, be defined ahead of submission, preclude the coding pattern mentioned at the end of Section 2.1, whereby the result of an operation determines the execution parameters of a dependent operation. Further, although the `then_host` node allows host work to be interleaved (Section 4.2), vendor restrictions may apply to the kind of host work

that may be performed; on CUDA, for instance, the host work may not make CUDA API calls [20]. Such restrictions prevent portable code from, for instance, resizing a `Kokkos::View` within a host node. The graph instantiation fixing the execution parameters is a limitation of the portable API rather than of the vendor graphs: the latter do provide mechanisms for updating the execution parameters of the nodes of an executable graph between submissions, and CUDA graph even allows certain updates to be issued from device code within a submission [21]. These vendor-specific features are accessible through the interoperability of `Kokkos::Graph` with the vendor graph API (Section 4.4), at the price of portability.

3. Resolving underspecified `Kokkos::Graph` semantics

`Kokkos::Graph` was introduced in Kokkos 3 [8]. However, key aspects of its API semantics across the three-stage graph object lifecycle were underspecified, such as the interaction with the execution space abstraction, user data lifetime and device selection. This section describes the changes integrated during the Kokkos 3 to Kokkos 5.1 release cycle, seeking to resolve these issues.

3.1. API as of Kokkos 3

Listing 4 shows a typical use of `Kokkos::Graph` as of Kokkos 3 [8]. The API provided an explicit definition stage through the `create_graph` function. It took two parameters: an execution space instance and a closure, whose semantic intent was to clearly indicate the scope wherein the graph could be defined. The execution space instance `exec_graph` was stored by the graph, but used only later to enqueue the graph submission. An execution space instance (`exec_node`) could be passed to the node execution policy, with the effect that the node operation would execute on the device to which this instance is bound (device selection). The first call to `submit` would instantiate the graph, and then submit the executable graph.

This API had several limitations and semantic shortcomings. First, the execution space instance used for submission had to be provided during the

definition stage, and was therefore missing at the submission site. Moreover, the submission or any subsequent resubmission could not be enqueued in any other instance than `exec_graph`. Second, there was no user-facing API for explicitly controlling when the instantiation stage happens. Third, the execution of the graph raises the question of data lifetime. Under the graph abstraction, execution is deferred and may be repeated, so any data the graph operations rely upon must be kept alive until the last submission completes. In Kokkos 3, it was not specified whether the user was responsible for keeping the data alive, or whether the graph object extended the lifetime of the data. Fourth, the closure enforced a scoped definition of the graph, thus imposing significant code design restrictions. The nodes were not intended to be used outside the closure. Lastly, an execution space instance (`exec_node`) could be passed to a node execution policy, but it was used solely to select the device. As such, it would not necessarily be used by the underlying graph implementation for enqueueing the node operation, contrary to what users familiar with the execution space abstraction semantics would expect. The sole actual guarantee was that the execution of the graph as a whole was ordered on `exec_graph`.

[Listing 4 about here.]

3.2. Evolution of the API

Enqueueing submission into an execution space instance. The `submit` API was extended to allow an execution space instance to be optionally passed, thus allowing users to choose where the submission is enqueued [22].

As per Kokkos conventions, API calls that enqueue operations must use the default execution space instance when no user-specified instance is provided. Accordingly, `submit` was modified in [23] to fall back to the default execution space instance when none is given. This change also enforces a clean separation between the definition and submission stages: the execution space instance stored by the graph at definition time is no longer used for submission.

Explicit instantiation stage. As of Kokkos 3, graph instantiation was triggered implicitly upon the first call to `submit`, leaving users with no control over when

this potentially costly stage of the graph lifecycle would occur. Therefore, an `instantiate` method was added to `Kokkos::Graph` [24].

Data lifetime. Given the deferred execution nature of the graph programming model, the data needed for executing graph operations must be kept alive until the last submission completes.

Each node object keeps the data used by its operation alive. For instance, a `then_parallel_for` node object stores its execution policy and functor by value; the functor, in turn, can be expected to hold the `Kokkos::View` objects that share ownership of the data that the call operator operates on.

The absence of a specified guarantee in Kokkos 3 had allowed backend specialisations to diverge: some did not take ownership of their node objects, so that operation data could be freed before graph destruction, leading to use-after-free crashes upon submission.

The resolution, integrated in [25], establishes the semantic guarantee that operation data remain alive at least for the lifetime of the graph object; accordingly, the graph object takes shared ownership of all node objects.

Relaxing the scoped definition. In the original design, the graph was constructed within a closure, which clearly delimited the scope of the definition stage. The closure thereby separated the definition of the graph, whose execution is deferred, from the surrounding code, which may enqueue work eagerly. The intended usage pattern was to capture by copy, without declaring the lambda `mutable`, so that the captured state was `const` within the closure. The design intent was that mistakes arising from conflating the definition of the operations with their deferred execution, such as attempts at directly transcribing the coding patterns mentioned at the end of Section 2.1, would be rejected at compile time. In practice, the protection afforded specifically by the closure proved modest: capture by copy was a convention that `create_graph` could not enforce; in addition, the type requirements of the node API already provided relevant compile-time checks. Notably, `then_parallel_reduce` requires the reduction target to be provided as a `Kokkos::View` or a reducer rather than as a host

scalar, so that a direct transcription of the reduce-then-for coding pattern fails to compile with or without the closure.

The scoped definition entailed significant code restrictions. It prevented the graph from being built incrementally, as required for instance by the FE assembly application of Section 6. Moreover, because capturing by copy was the intended usage pattern, developers were discouraged from retaining node handles beyond the definition stage. Consequently, the original design precluded extensions of the graph API towards deeper interoperability with the vendor graph API, such as selectively disabling a node with `cudaGraphNodeSetEnabled` or updating node parameters between submissions (Section 4.4.2).

The closure-based definition was relaxed in [26] by allowing the graph object to be constructed standalone; the end of the definition stage, previously delimited by the closure, is now marked by the call to `instantiate` (Section 2.2).

The relaxation facilitates storing node handles and thus heightens the risk of invalid code patterns in which nodes outlive the graph. However, this risk was already latent in the original design through by-reference capturing lambdas, and the precondition checks implemented for the original design catch such violations.

Execution policy and device handle. The graph abstraction reuses the execution policies of the execution space abstraction for defining its computational nodes. In particular, these execution policies own an execution space instance. In the execution space abstraction, the established semantic is that an operation is enqueued in the execution space instance owned by the execution policy. However, as the underlying graph implementation is free to choose how node operations are executed, the graph abstraction cannot preserve this enqueueing semantic. In Kokkos 3, users could nonetheless specify an execution space instance in the node execution policy. It provided a mechanism to select the device on which the node operation would execute, allowing the definition of a graph spanning multiple devices. This was semantically inconsistent: users familiar with the execution space abstraction would expect the instance to govern enqueueing,

while it was in fact used only for device selection.

A dedicated device handle was introduced for explicit device selection during the graph definition stage [27]. It expresses the device on which the node operation executes, without referring to an execution space instance. The device handle can be passed through a new node properties construct that was also introduced in [27]. If no device handle is given, the device handle held by the graph object itself is used as a default. Accordingly, the execution policy must be built without providing an execution space instance. An example is shown in Listing 5.

Since `Kokkos::Graph` still reuses the execution policies of the execution space abstraction that own an execution space instance, a runtime check was introduced in [27]. This runtime check aborts node creation if the execution policy owns a user-provided non-default execution space instance. This provides a partial safeguard; detecting the case where the user passed a default instance to the execution policy constructor remains a further improvement still to be addressed in future work.

The device handle internally stores an execution space instance that the `Kokkos::Graph` internals may use as a default queue for the selected device if the node requires preparatory work. For instance, when creating a node whose computational work is defined by a large functor, certain `Kokkos` backends trigger an asynchronous copy of this large functor to the selected device ahead of submission; the internally stored execution space instance may thereby be used for such an asynchronous copy. Since the user only passes a device handle, the responsibility for proper synchronisation of such preparatory work before graph submission semantically falls to `Kokkos`.

3.3. API as of *Kokkos* 5.1

Listing 5 shows a typical use of `Kokkos::Graph` incorporating the above API changes. The three-stage graph object lifecycle is now explicit in the public API. The user is given more freedom in defining the graph since no closure is required any more. Device selection and interaction with execution space instances are

now semantically clearer.

[Listing 5 about here.]

4. Extending Kokkos::Graph capabilities

This section covers features added to Kokkos::Graph during the Kokkos 3 [8] to Kokkos 5.1 [9] release cycle.

4.1. *then* node

The **then** node was introduced in [28]. It represents an operation that invokes a functor whose call operator takes no index argument and must be decorated with KOKKOS_FUNCTION, as illustrated in Listing 6. The **then** node is intended for an operation that requires no iteration space, such as a scalar update or an operation that interacts with a CUDA conditional node handle.

Such an operation can also be expressed with a **then_parallel_for** node, using the half-open range $[0, 1[$ as the iteration space, but this approach unnecessarily introduces loop machinery that the compiler cannot fully eliminate. For instance, even with Kokkos::LaunchBounds<1>, the CUDA backend can only assume there is one thread per block, but cannot optimise out grid-size-related logic. Beyond better expressing semantic intent, the **then** node preserves the information that only a single thread of execution is needed, which an implementation-specific code path for **then** could leverage.

However, as of Kokkos 5.1 [9], the underlying implementation of the **then** node still maps to a **then_parallel_for** node with Kokkos::LaunchBounds<1> [29]. A dedicated implementation is still a direction for future work.

[Listing 6 about here.]

4.2. *then_host* node

The **then_host** node was introduced in [30]. It represents an operation that invokes a functor on the host whose call operator takes no index argument, as illustrated in Listing 7. The **then_host** node is intended for operations that must occur on the host, such as I/O or host-side MPI calls. Unlike **then**, the

`then_host` operation does not accept a device handle. Indeed, vendor graph implementations do not expose control over how the host functor is invoked. As already mentioned, vendor restrictions may apply to the operations that may be carried out. Internally, `Kokkos` maps `then_host` to a callback with user-specified data on CUDA and HIP, a `host_task` on SYCL [5, 31], and an inline call for the default implementation.

[Listing 7 about here.]

4.3. Capture node

The capture node was introduced in [32]. It is an interoperability feature that allows operations not directly expressed with `Kokkos` to be added to a `Kokkos::Graph`. A key use case arises from operations provided by external libraries, such as cuBLAS or cuSPARSE. Another notable use case concerns operations deliberately not expressed with `Kokkos`, for instance to preserve a legacy or fine-tuned implementation. The capture node is supported on the CUDA and HIP backends via stream capture, and on the SYCL backend via queue recording.

The capture node takes two arguments: a functor describing the operations to capture, and an execution space instance whose underlying vendor queue is used for the capture, as illustrated in Listing 8. Internally, upon adding a capture node, `Kokkos` puts the vendor queue in capture mode, invokes the functor with the execution space instance, and then resets the vendor queue to normal mode. Operations enqueued into the vendor queue during the functor invocation are captured as a sequence of nodes in the vendor graph rather than executed, collectively forming the `Kokkos` capture node within the `Kokkos::Graph`.

The capture node deliberately takes an execution space instance rather than a device handle. Indeed, the underlying vendor queue is put in capture mode during the functor invocation, thus implying that any other operations enqueued into it from any other host thread are captured as well, *i.e.*, not just those from the functor invocation itself. Explicitly exposing the execution space instance in the API gives the user control over which queue is involved.

The functor also serves to extend data lifetime: any data required by the captured operations can be stored by value in the functor. The capture node stores its own copy of the functor, so that such data are kept alive until the graph object is destroyed.

Capture is an advanced interoperability feature that broadens the set of use cases that can be mapped to `Kokkos::Graph`. However, vendor restrictions apply to the operations that may be captured. Moreover, `Kokkos` cannot provide any guarantee on what is being captured and mandates that no `Kokkos`-based operation be captured. The user should only capture backend operations explicitly documented as supporting capture.

[Listing 8 about here.]

4.4. Interoperability with the vendor graph API

Each backend specialisation may provide interoperability with the underlying vendor graph objects. Such interoperability enables two important use cases. First, it provides access to vendor-specific features not currently surfaced due to lack of portability. Second, it allows forward-looking developments that go beyond what `Kokkos::Graph` currently implements. Any such interaction must be exercised with care: manipulating the graph through the vendor API must not violate any `Kokkos::Graph` invariant, as `Kokkos` does not attempt to detect inconsistencies introduced through the vendor API.

On CUDA, HIP and SYCL, `Kokkos::Graph` exposes the following handles: graph and executable graph (*e.g.*, `cudaGraph_t` and `cudaGraphExec_t`), and nodes (*e.g.*, `cudaGraphNode_t`).

4.4.1. Graphs

The interoperability with the vendor graph API was introduced in [33, 34, 35]. It is two-way for the graph object itself, but one-way for the executable graph. A `Kokkos::Graph` can be constructed from an existing vendor graph handle (*e.g.*, `cudaGraph_t`). Listing 9 illustrates such a use case, whereby a CUDA `while` node creates a child `cudaGraph_t` for the loop body, and wrapping

it into a `Kokkos::Graph` allows populating it through `Kokkos` constructs. Both the underlying vendor graph handle and the executable graph handle (*e.g.*, `cudaGraphExec_t`) held by a `Kokkos::Graph` can be accessed. For instance, it allows the user to use `cudaGraphUpload`, which has no counterpart in SYCL and as such is not portable.

One of the invariants that govern the `Kokkos::Graph` object is that the underlying vendor graph handle may not be replaced. Accordingly, the accessor returns a handle that does not permit reassignment.

[Listing 9 about here.]

4.4.2. Nodes

The interoperability with the vendor node handles was introduced in [36, 37]. On the one hand, `Kokkos::Graph` nodes have accessors to retrieve the underlying vendor node handle. For instance, they can be used to enable or disable the underlying CUDA node with `cudaGraphNodeSetEnabled`. On the other hand, a `Kokkos::Graph` node can be constructed by using vendor graph API. For instance, it can be useful to integrate a CUDA `if` node into a `Kokkos::Graph`.

5. Performance evaluation across backends

This section compares the execution space abstraction with the graph abstraction across the CUDA, HIP and SYCL backends, using a single-source code for each case study. The section first addresses two canonical topologies and then moves to iterative solvers, which illustrate use cases involving repeated DAG submission that are expected to benefit from the graph abstraction.

No comparison is provided with implementations realised directly in terms of the vendor API. Indeed, apart from the use of `Kokkos` internals to define node operations, `Kokkos::Graph` is a thin wrapper that forwards to the underlying vendor graph API, with negligible overhead.

5.1. Experimental setup

For CUDA, the machine is a GH200 Grace Hopper; the CUDA version is 13.0.0, and the driver version 580.65.06. For HIP, the machine has a Xeon

Platinum 8462Y+ CPU and an MI300X GPU; the ROCm version is 7.2.4, and the driver version 6.16.13. For SYCL, the machine has a Xeon Silver 4410T CPU and an Intel Data Center Max 1100 GPU; the Intel compiler version is 2026.0.0, and the driver version 23.5.19. The Kokkos version is 5.1.

The Google Benchmark 1.9.5 framework is used to drive the experiments; the reported values are the mean over a number of repetitions determined automatically by the framework, following a warm-up phase.

The code used for the benchmarks is available at <https://github.com/uliegecsm/graph-dispatching>. The version used for the benchmarks corresponds to the tag `kokkos_graph_paper`.

5.2. Reported timings and metrics

For each case study, two implementations are compared: the first uses the execution space abstraction and is the baseline; the second uses `Kokkos::Graph`.

The reported elapsed time is defined as follows for the execution space implementation:

- T^E : from the call that enqueues the first operation to the completion of all enqueued operations.

The creation of a pool of execution space instances is thereby never included in the reported elapsed times.

The elapsed times are defined as follows for the graph implementation:

- T_d^G : the definition stage,
- T_i^G : the instantiation stage,
- T_{s0}^G : from the call that submits the graph to its completion for the first submission,
- T_{sr}^G : from the call that submits the graph to its completion for a repeat submission.

The first submission is distinguished from a repeat submission because CUDA performs an implicit graph upload step upon the first submission, unless it has

been explicitly triggered beforehand through `cudaGraphUpload`. Therefore, T_{s0}^G can be expected to be larger than T_{sr}^G . HIP exposes `hipGraphUpload` [38, 39]; however, examination of the source code indicates that it is not called upon graph submission, and it currently performs only a few verifications protected by a scope guard upon being called explicitly. SYCL does not describe whether or when such an upload happens, nor does it expose one in its API.

The speedup of adopting a graph after n submissions is defined as:

$$S_n = \frac{n \times T^E}{T_d^G + T_i^G + T_{s0}^G + (n - 1) \times T_{sr}^G}.$$

The asymptotic speedup is defined as follows:

$$S = \frac{T^E}{T_{sr}^G}.$$

It represents the achievable speedup in the limit where the cost of the graph definition, instantiation and upload is fully amortised. Provided that $S > 1$, the break-even point of the graph implementation is achieved after n_{be} submissions:

$$n \geq n_{be} = \left\lceil \frac{T_d^G + T_i^G + T_{s0}^G - T_{sr}^G}{T^E - T_{sr}^G} \right\rceil \implies S_n \geq 1.$$

In this expression, $\lceil \cdot \rceil$ is the ceiling function that rounds a number up to the nearest integer, and the contribution $T_{s0}^G - T_{sr}^G$ to the numerator accounts for the overhead that a potential graph upload step may incur.

5.3. Canonical graphs

Two canonical topologies are studied: a straight-line graph and a fork graph, both populated with nodes that deliberately execute small operations. The overall execution time of these topologies is thus dominated by the scheduling overhead, much like the use cases that CUDA graphs initially addressed.

5.3.1. Straight line

The straight-line graph consists of a sequence of m operations forming a linear chain, as illustrated in Fig. 4. In this case study, each operation executes the same kernel consisting of a single thread incrementing a 4-byte counter in device global memory. The graph implementation uses `then` nodes.

[Figure 4 about here.]

Table 2 provides results for $m = 10$ operations. For this DAG, the elapsed times are such that $T^E > T_{\text{sr}}^G$ only on CUDA; hence, there is a break-even number of submissions n_{be} for which adopting the graph abstraction is beneficial only on CUDA. Notably, this break-even point is attained strikingly rapidly on CUDA.

[Table 2 about here.]

Figure 5 shows how S and n_{be} vary with the number of operations m . It can be observed that S increases with m , indicating that the marginal cost of executing an additional operation is lower using the graph abstraction than using the execution space abstraction. On CUDA and HIP, n_{be} decreases with m : the break-even point is reached more rapidly as the number of operations grows.

[Figure 5 about here.]

5.3.2. Fork graph

The fork graph consists of b parallel branches of m operations each, all connected to a common root, as illustrated in Fig. 6. Each operation is the same as the one used for the straight-line graph.

[Figure 6 about here.]

The execution space implementation assigns one execution space instance per branch from a pool of b instances, and submits operations according to the sequence $\{N_1^1, \dots, N_1^b, \dots, N_m^1, \dots, N_m^b\}$, thus creating the opportunity for concurrent execution and launch latency hiding. The graph implementation uses **then** nodes, defined according to the sequence $\{N_1^1, \dots, N_m^1, \dots, N_1^b, \dots, N_m^b\}$. The node definition order may influence the scheduling decisions of the underlying vendor implementation [40].

Table 3 shows that $T^E > T_{\text{sr}}^G$ on all backends for this DAG for $b = 4$ and $m = 10$. As for the straight-line graph, the break-even point is attained strikingly rapidly on CUDA.

The asymptotic speedup S on CUDA is about 4, but the numerical similarity to the number of branches is coincidental: S depends on both b and m . As noted in [41], the cost of defining a CUDA graph scales approximately linearly with the number of nodes and edges, while the instantiation cost grows sub-linearly. This trend is confirmed here by comparing the values of T_d^G in Table 2 and Table 3.

On HIP, the instantiation cost is significantly larger for this fork graph than for the straight-line graph.

[Table 3 about here.]

5.4. Iterative solvers

This section evaluates three iterative solvers: conjugate gradient (CG), preconditioned conjugate gradient (PCG) and Newton. The implementations follow reference algorithms and use linear algebra kernels from `Kokkos Kernels`, because the focus of this section is on demonstrating the feasibility and providing insight into the performance benefits of `Kokkos::Graph`. More sophisticated algorithms also exist [42, 43], often relying on specialised kernels, enabling optimisations that may benefit the execution space abstraction or the graph abstraction or both, but such algorithms are beyond the focus of this paper.

5.4.1. Conjugate gradient

The linear problem to be solved is a tri-diagonal complex-valued system $[A]\mathbf{x} = \mathbf{b}$ derived from an FE discretisation of a 1D Laplace equation with a homogeneous Dirichlet boundary condition on the left and a non-homogeneous complex-valued Neumann boundary condition on the right. The system matrix $[A]$ is Hermitian positive definite.

The implementation follows the standard algorithm described in [43, Section 6.7], and is depicted in Fig. 7. For each iteration k , the algorithm begins with computing the step size α_k :

$$\alpha_k = \frac{\mathbf{r}_k \cdot \mathbf{r}_k}{\mathbf{p}_k \cdot [A]\mathbf{p}_k}, \quad (1)$$

using the result of the operations `spmvAp` and `dotpAp`, which compute the sparse matrix-vector product $[A]\mathbf{p}_k$ and the dot product $[A]\mathbf{p}_k \cdot \mathbf{p}_k$, respectively. Once

α_k is computed, the algorithm splits into two branches. The left branch computes the updated solution vector $\mathbf{x}_{k+1}$ in the operation `axpyx`:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k. \quad (2)$$

The right branch starts with computing the updated residual $\mathbf{r}_{k+1}$ in the operation `axpyr`:

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k [A] \mathbf{p}_k, \quad (3)$$

and then computes $\mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}$ in the operation `dotrr`, needed to compute β_k :

$$\beta_k = \frac{\mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}}{\mathbf{r}_k \cdot \mathbf{r}_k}. \quad (4)$$

To ensure the solution update has completed, the branches are joined before updating the search direction in the operation `axpyp`:

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k. \quad (5)$$

The implementation loops until either a maximum number of iterations is reached or the norm $\|\mathbf{r}_{k+1}\|_2$ is below a threshold.

The execution space implementation uses a pool of two execution space instances to allow the two branches to execute concurrently. The reduction results of `dotpAp` and `dotrr` are stored in a zero-rank `Kokkos::View` allocated in host-pinned memory. The coefficients α_k and β_k are host scalars and computed on host. They are copied into the `axpyr`, `axpyx` and `axpyp` operation functors upon launching these operations from the host.

In the graph implementation, the operations are defined ahead of submission. Hence, values such as α_k and β_k must be stored in dedicated memory, allocated during the definition stage and accessible to the operations that use them.

The computations of α_k and β_k could be integrated as epilogues in the `dotpAp` and `dotrr` reduction operations. However, such an approach would require a fused operation and may hinder the use of external libraries that provide the dot product. Therefore, the considered implementation computes α_k and β_k in dedicated nodes. The use of `then` nodes implies device operations, thus allowing the reduction results and the coefficients to be stored in

memory that need only be device-accessible. The use of `then_host` nodes, involving host operations, is an alternative. However, because `Kokkos::Graph` does not yet provide a portable native memory-copy node and because vendor API calls are generally prohibited in host nodes (Section 2.2 and Section 4.2), the `then_host` node requires both the reduction results and the coefficients to reside in host-and-device-accessible memory, *i.e.*, in `Kokkos::SharedSpace` or host-pinned memory.

On CUDA, host-and-device-accessible memory entails performance penalties. These penalties may be incurred multiple times: whereas a single device thread writes the reduction result in `dotpAp` and `dotrr`, many device threads read the coefficients in `axpyr`, `axpyx` and `axpyp`. Software-coherent systems resort to page faults and entire page migration [20, Section 4.1.1.2.1.2] [44, 45]. Hardware-coherent systems do not resort to page faults and migrate data at cache line granularity, resulting in more efficient memory transfers, though still relying on two separate physical memory spaces [46]. For host-pinned memory, all GPU accesses must go through the connection between the CPU and GPU [20, Section 2.4.3.1.3].

An implementation using `then_host` nodes has been tested, but it was significantly slower than an implementation using `then` nodes on several software-coherent systems with discrete NVIDIA GPUs. Results reported hereafter thus use the device-node-based implementation with `then` nodes, with the reduction results and the coefficients α_k and β_k stored in device memory. In this implementation, the `then` node computing β_k first uses the reduction result of `dotrr` in its computation and then copies it into a zero-rank `Kokkos::View` allocated in host-pinned memory, so that it can be used on the host to evaluate the convergence criterion.

[Figure 7 about here.]

Figure 8 shows how S and n_{be} vary with the size of the linear system. The values of T^E , T_{s0}^G and T_{sr}^G used to obtain these results are elapsed times measured per submission of the CG graph of Fig. 7, which corresponds to a single CG

iteration. A beneficial asymptotic speedup S is achieved only on the CUDA backend. It reaches about 1.27 for small linear systems, for which scheduling overhead is a significant fraction of the overall execution time. On CUDA, HIP and SYCL alike, the asymptotic speedup S tends to one as the work per operation grows and the scheduling overhead thus becomes negligible relative to the operation execution time.

[Figure 8 about here.]

5.4.2. Preconditioned conjugate gradient

The preconditioned conjugate gradient (PCG) algorithm extends CG by introducing a preconditioner into the iteration loop, as shown in Fig. 9. The implementation follows the standard PCG algorithm described in [43, Section 9.2]. The implemented preconditioner consists of m sweeps of Jacobi stationary iteration:

$$\mathbf{z}_k^{j+1} = [D]^{-1}(\mathbf{r}_k - [L + U]\mathbf{z}_k^j), \quad 0 \leq j < m, \quad (6)$$

started from $\mathbf{z}_k^0 = \mathbf{0}$, with $[A] = [L + D + U]$ the decomposition of the system matrix into its lower triangular, diagonal and upper triangular parts. The first iteration collapses to $\mathbf{z}_k^1 = [D]^{-1}\mathbf{r}_k$. The iterates $\mathbf{z}_k^j$ and $\mathbf{z}_k^{j+1}$ are stored in separate views and swapped between successive sweeps to avoid an additional deep copy operation.

The preconditioner is a single-entry single-exit region in the algorithm graph. The number of execution space instances that must be provided to the execution space implementation to maximise the opportunity for concurrent in-flight operations depends on the width of the preconditioner DAG. In this case, the preconditioner is a straight-line DAG, and the full PCG DAG has a width of three, so that three execution space instances suffice.

In the graph implementation, the node `axpyr` is passed as the predecessor to the preconditioner, which adds the sweep operations followed by a deep copy operation if m is odd, and returns the last node to the caller.

The number of sweeps m is a runtime parameter, thus requiring the addition of the sweep operations to be implemented as a `for` loop with type-erased

nodes. The additional precedence constraint between nodes `dotrr` and `dotzr` is needed because of the `Kokkos::Graph` restriction mentioned in Section 2.2: both nodes are reductions, and, due to the aforementioned implementation limitations, one must be the topological ancestor of the other to ensure correctness. The `deep_copyz` node has been implemented using the graph capture node (Section 4.3) to ensure that performance is comparable with that of the `deep_copy` operation in the execution space implementation. On CUDA, it captures a `cudaMemcpyAsync` call.

Both implementations keep the number of sweeps constant across iterations. The execution space implementation can be readily adapted to support a varying sweep count via a `while` loop. However, `Kokkos::Graph` does not yet surface either a `while` node or a node enable/disable feature. Therefore, the graph implementation currently can be adapted only per backend via the interoperability features (Section 4.4). For CUDA, it would entail using the `while` node.

[Figure 9 about here.]

Figure 10 shows how S varies with the size of the linear system for $m = 4$, 8 and 12 sweeps of the preconditioner; n_{be} is shown only for the $m = 8$ cases where $S > 1$. As in the case of the CG solver, the values of T^E , T_{s0}^G and T_{sr}^G used to obtain these results are elapsed times measured per submission of the PCG graph of Fig. 9. A beneficial asymptotic speedup S is achieved consistently only on the CUDA backend, where it reaches about 1.52 for small linear systems for the smallest number of sweeps, $m = 4$. Each additional Jacobi sweep adds kernel-execution work whose cost is similar under both abstractions. This additional work dilutes the relative difference in scheduling overhead, driving S towards one as m grows. Accordingly, on CUDA, where $S > 1$, the speedup is larger for smaller m , whereas on HIP and SYCL, especially for small systems where $S < 1$, S is closer to one for larger m .

[Figure 10 about here.]

5.4.3. Newton solver

The non-linear problem to be solved is derived from an FE discretisation of the following non-linear 1D heat transfer equation for the temperature $T :]0, L[\rightarrow \mathbb{R}$:

$$\frac{d^2 T}{dx^2} - kT^2 = 0, \quad (7)$$

with k, L in $\mathbb{R}$ and boundary conditions:

$$\begin{cases} T(0) = 1, \\ \frac{dT}{dx}(L) = 0. \end{cases} \quad (8)$$

The implementation follows the standard Newton algorithm with a full step size described in [47, Section 5.1.1]. Each outer iteration starts with the assembly of the Jacobian matrix and the residual vector, followed by the solution of the linearised system. As depicted in Fig. 11, the assembly consists in filling the local elemental Jacobian matrices and residual vectors in the operation `fill`, which are then scatter-contributed into the global Jacobian matrix and residual vector in the operation `scatterA,r` [48, 49]. The inner linear solver is the PCG from Section 5.4.2.

The execution space implementation uses a pool of three instances, as needed to maximise the opportunity for concurrent in-flight operations, because, like the PCG DAG, the problem assembly DAG has a width of three.

The graph implementation uses two distinct `Kokkos::Graph` objects for the problem assembly and the PCG that are defined and instantiated only once during the first iteration. A natural extension would be to accommodate both the problem assembly and the inner linear solver into a single monolithic graph, using a `while` node to drive the inner loop. However, as already mentioned, `Kokkos::Graph` does not yet surface a `while` node.

[Figure 11 about here.]

[Table 4 about here.]

Table 4 shows the overall times to solution for the Newton solver for two sizes of the system of non-linear equations. An overall computational benefit is achieved only on the CUDA backend.

6. Application to FE assembly

This section reports on using `Kokkos::Graph` in HELM, an experimental, multiphysics FE code developed in the computational and stochastic modelling research group at the University of Liège. HELM integrates several packages from the Trilinos library [50], such as Tpetra for distributed linear algebra, Kokkos for on-node parallelism, Amesos2 for linear solvers and Intrepid2 for FE discretisation tools. On top of these packages, HELM implements its own FE assembly based on a per-rank gather-fill-scatter pattern [48], supporting research directions such as GPU-resident assembly and the use of special uncertainty-quantification scalar types in the assembly. HELM provides a core of common functionalities, and applications based on HELM need only provide functors that encode the element-level discretised physics.

The gather-fill-scatter pattern decomposes the assembly into generic gather and scatter operations, managed by HELM, and application-specific fill and boundary condition operations. These operations are specialised for subregions of the FE mesh: elements are grouped into worksets [48] according to shared characteristics such as element topology, FE discretisation, governing equations, or boundary condition types. Specialising operations exposes performance optimisation opportunities, such as avoiding warp divergence induced by a mixing of element topologies, eliding lookups of physical coefficients known to be spatially constant over a subregion, or encoding coefficient values known at compile time to enable compiler optimisations. The FE assembly therefore comprises many operations that must be scheduled efficiently, and often repeatedly, as in iterative solvers for non-linear problems (Section 5.4.3).

Figure 12 (left) shows an example of a frequency-domain electromagnetic wave scattering problem, motivated by [51, 52]. The computational domain is subdivided into several subregions: the nanoline, where the scattered-field

governing equations involve a source term; the air, spacers and substrate, where the material properties are spatially homogeneous; the perfectly matched layers, where the material properties are spatially heterogeneous. Periodic boundary conditions are imposed on the lateral boundaries, and homogeneous Dirichlet boundary conditions are imposed on the top and bottom. Each subregion uses its own specialised operation. Figure 12 (right) shows a schematic overview of the FE assembly operations and their precedence constraints.

HELM builds the entire FE assembly graph from the generic operations and those provided by the application. HELM is responsible for orchestrating the execution of this DAG appropriately and efficiently and must therefore store the application-specific functors. A common pattern would be for HELM to provide an interface as a base class with virtual functions, which the application then implements, with HELM storing an array of base pointers. However, virtual polymorphism at the functor level implies virtual calls on device, and requires special care to construct the *vtable* on device; it implies additional indirections and degrades performance [53]. An alternative is a type-erasure pattern that moves the polymorphism to the operation invocation on the host [54]. An execution-space-based scheduling would require HELM to analyse the FE assembly DAG, provision a suitably sized pool of instances, and devise a traversal strategy that inserts synchronisation points to observe dependencies.

Instead, HELM uses `Kokkos::Graph` to store the operations and orchestrate their execution. `Kokkos::Graph` handles both functor storage and efficient scheduling [41], relieving the core of HELM of both burdens. The graph node owns the functor, so the lifetime of application-specific data is extended naturally (Section 3.2). The application needs only supply functors modelling the fill or boundary condition concepts; no base class is required. HELM also lets an application insert a subgraph instead of a single operation, provided that the subgraph forms a single-entry single-exit region.

`Kokkos::Graph` thus provides HELM with a software component that handles the storage of the functors and the orchestration of the execution of the operations under their precedence constraints, in a manner that exposes the

concurrency between them. The code-design benefits of delegating these responsibilities to a dedicated software component hold regardless of the use case. By contrast, whether the exposed concurrency translates into a speedup is case-dependent. In the application described hereinabove (Fig. 12), the problem being two-dimensional, the individual operations are too small to saturate a modern scientific-computing GPU by themselves, so that exposing the concurrency between them is likely to be beneficial. In turn, whether the graph-based scheduling outperforms the execution-space-based alternative is also case-dependent: as the case studies of Section 5 illustrate, it depends on many factors, such as the opportunity for amortising the upfront definition and instantiation costs over repeated submissions. Here, the problem being linear, the assembly comprises only a single pass with no resubmission, so that the amortisation is forgone (for a non-linear problem, the assembly is resubmitted at each iteration of the solver, restoring the amortisation; see Section 5.4.3). The upfront definition and instantiation costs are, however, small as compared with the cost of the assembly as a whole. Altogether, the case for `Kokkos::Graph` in HELM rests primarily on the code-design benefits and on the exposure of the concurrency between the operations, with the speedup itself case-dependent.

[Figure 12 about here.]

7. Discussion

7.1. Use of *Kokkos* features and optimisations

The graph abstraction leverages much of the execution space abstraction internals. In particular, the implementation details that map the dispatch of an operation to a backend are largely shared. The main difference is that code paths which, under the execution space abstraction, enqueue work into a vendor queue instead create a node in a vendor graph.

As such, an operation enqueued into an execution space instance and one added as a graph node can be expected to behave similarly. For instance, CUDA kernel launch configuration heuristics are reused. This allows applications to

benefit from features and optimisations originally developed for the execution space abstraction.

Beyond execution abstractions, most `Kokkos::Graph` performance portability features, such as the data structures, mathematical functions and atomic operations, can be expected to work identically and with similar performance in graphs.

7.2. *Memory effects visibility*

Edges of a `Kokkos::Graph` imply not only precedence constraints, *i.e.*, execution order constraints of the nodes, but also a happens-before memory-order relationship. In other words, any node operation is guaranteed to observe all memory effects produced by its predecessor operations. The execution space abstraction has the same guarantee for operations enqueued into the same execution space instance, as well as for those enqueued into different instances when ordered through fences.

The SYCL command graph specification explicitly states the happens-before relationship [31, Section 8.2]. CUDA guarantees “a full dependency with memory synchronizing behavior” [20, Section 4.2.1.2].

7.3. *Compile-time opportunities*

By default, the `Kokkos::Graph` API returns fully typed node handles that carry the types of their operation and predecessors. As such, inspecting the type of `node_C` in Listing 3 would reveal the type of its parallel pattern, policy and functor, as well as those of its predecessors (in this case, `node_A`), recursively. `Kokkos::Graph` also supports type-erased node handles, as needed for progressive graph definition driven by runtime conditions.

Fully typed nodes may enable optimisations, such as compile-time kernel fusion [55, 56, 57]. However, `Kokkos::Graph` does not currently seek to exploit such optimisation opportunities.

7.4. *Dynamic control flow in static DAGs*

The topology of a `Kokkos::Graph` does not need to be known at compile time: it can be defined progressively, driven by runtime conditions. However,

instantiation locks the graph, preventing topology modification through the portable `Kokkos::Graph` API between submissions or during execution.

CUDA, HIP and SYCL executable graphs are static DAGs as well: their topology cannot be modified. CUDA and HIP partially mitigate this limitation by allowing individual nodes to be selectively enabled or disabled between submissions. Additionally, CUDA exposes conditional `if` and `while` nodes that embed dynamic control flow directly into the topology [58]. `Kokkos::Graph` provides access to such vendor-specific features through interoperability (Section 4.4), though at the expense of portability. Topology changes between submissions can also be accommodated by maintaining multiple `Kokkos::Graph` objects and switching among them.

7.5. Adoption

As `Kokkos::Graph` matures, integrating it into the broader `Kokkos` ecosystem is a natural next step. `Kokkos Kernels`, the performance-portable linear algebra library built on `Kokkos Core`, currently supports only the execution space abstraction. As such, it lends itself well to discussing developments that would be needed to extend the API and internals of a library to support both abstractions.

By design, `Kokkos` already entails that the work of computational operations is described in functors, which are inherently reusable across both abstractions. Many `Kokkos Kernels` functors could hence be used across the execution space implementation and a potential graph implementation.

Still, extending a library to support the graph abstraction presents significant challenges. At the user-facing API level, `Kokkos Kernels` functions typically take an execution space instance as their first argument to enqueue operations. Functions returning a scalar reduction result are blocking and thus incompatible with the graph abstraction. Functions returning `void` are potentially asynchronous. Depending on whether their computational work lends itself to being expressed in terms of graph node operations, such functions may be amenable to `Kokkos::Graph`. Enabling such compatible `Kokkos Kernels` functions to be

used with `Kokkos::Graph` would require the API to be extended with overloads that take a predecessor node instead of an execution space instance, add their operations to the parent graph after the passed node, and return a single node handle, thus allowing a single-entry single-exit region to be formed in the parent graph (Listing 10).

A representative challenge arises when `Kokkos Kernels` lowers to vendor libraries such as cuBLAS. Mapping such a lowering to `Kokkos::Graph` would require graph support from the vendor library itself, which cuBLAS provides through stream capture (Section 4.3).

Dynamic control flow presents a more fundamental challenge. For instance, `Kokkos Kernels` often optimises performance by selecting kernels based on library configuration and runtime coefficient values, such as by bypassing `axpy` scaling if the coefficient equals one. The selection logic happens on host at each function invocation. By contrast, the graph abstraction would require the selection to be encoded once at definition time. Therefore, whenever the selection depends on runtime values that vary across submissions, conditional nodes would be required, and the current lack of portable support for these represents a significant barrier.

While supporting both execution abstractions may lead to code divergence, the two abstractions would maintain interoperability: graph submission can be enqueued into an execution space instance, thus enabling graph-based operations to integrate with and be ordered relative to execution-space-based code (Section 3.2).

[Listing 10 about here.]

8. Conclusion

This paper described the graph abstraction of the `Kokkos` performance portability library, contrasted it with the well-established execution space abstraction, resolved issues of underspecified semantics, and provided new features. The main semantic resolutions are: the three-stage lifecycle is clearly exposed in the

API; the graph submission may be enqueued into an execution space instance, bridging the two abstractions; the operation data lifetime is tied to at least that of the graph object; and node execution policies must not be constructed with a user-provided execution space instance, with device selection instead expressed through a dedicated device handle passed to a new node properties construct. The new features, such as `then`, capture and interoperability, broaden the range of applicability of `Kokkos::Graph` and thereby foster its adoption.

The performance evaluation introduced two metrics: the asymptotic speedup S and the break-even number of submissions n_{be} . Across backends, only CUDA showed an asymptotic speedup consistently greater than one, though the one-time costs must still be amortised over multiple submissions. On CUDA, HIP and SYCL alike, the asymptotic speedup tends to one as the work per operation grows and the scheduling overhead thus becomes negligible relative to the operation execution time. A concrete application to FE assembly highlighted the code-design benefits of delegating DAG storage and scheduling to `Kokkos::Graph`.

Several directions are natural next steps. `Kokkos::Graph` does not yet surface some features already common in the vendor graph implementations, such as memory nodes. The `parallel_scan` pattern is likewise yet to be implemented. Future work could explore additional node properties beyond the device handle, such as node priorities to guide the underlying scheduler. The graph abstraction could also allow for custom scheduler implementations, thus allowing different scheduling strategies to be explored with little change to application code.

Even though HIP and SYCL do not always deliver a performance benefit, vendors continue to improve their implementations; notably, recent HIP and SYCL releases showed significant gains.

Key challenges remain. First, certain coding patterns that can be expressed in the execution space programming model are not currently supported by `Kokkos::Graph`. Second, a library that adopts both abstractions must provide and maintain two incompatible user-facing APIs with distinct execution

semantics. A third challenge is supporting dynamic control flow within the static DAG.

Author contributions

C. Trott and D. Lebrun-Grandié were involved in the original design and development of `Kokkos::Graph`. R. Tomasetti developed the design and implemented the improvements and extensions described in Section 3 and Section 4 and carried out the performance evaluation in Section 5, in collaboration with D. Lebrun-Grandié, C. Trott and M. Arnst. R. Tomasetti and M. Arnst developed the FE code described in Section 6.

Acknowledgements

This work was supported by a FRIA fellowship of the Fonds de la Recherche Scientifique de Belgique, Belgium (F.R.S.-FNRS).

This work was supported by the U.S. Department of Energy through the Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001).

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the authors used Claude and Chat-GPT in order to improve the clarity and check the internal consistency of the manuscript. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

- [1] A. Doboli, P. Eles, Scheduling under data and control dependencies for heterogeneous architectures, in: IEEE International Conference on Computer Design: VLSI in Computers and Processors, (ICCD), 1998, pp. 602–608. doi:10.1109/ICCD.1998.727121.
- [2] J. Coleman, B. Krishnamachari, Pisa: An adversarial approach to comparing task graph scheduling algorithms, in: 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2025, pp. 54–66. doi:10.1109/IPDPS64566.2025.00014.
- [3] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274. doi:10.1109/71.993206.
- [4] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, T. R. Scogland, RAJA: Portable Performance for Large-Scale Scientific Applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81. doi:10.1109/P3HPC49587.2019.00012.
- [5] The Khronos SYCL Working Group, SYCL 2020 Specification (revision 11), <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>, accessed: 2026-06-21 (2020).

- [6] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, M. Bussmann, Alpaka – an abstraction library for parallel kernel acceleration, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 631–640. doi:10.1109/IPDPSW.2016.50.
- [7] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216. doi:10.1016/j.jpdc.2014.07.003.
- [8] 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, J. Wilke, Kokkos 3: Programming Model Extensions for the Exascale Era, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2022) 805–817. doi:10.1109/TPDS.2021.3097283.
- [9] D. Lebrun-Grandié, C. Trott, D. Arndt, M. Arnst, J. Bludeau, C. Chevalier, J. Ciesko, C. Clevenger, N. Ellingwood, P. Gannay, R. Gayatri, E. Harvey, D. Ibanez, C. Junghans, D. Lee, S. Lee, N. Liber, N. Morales, T. Morvany, T. Padioleau, C. Pearson, A. Prokopenko, F. Rizzi, M. Simberg, C. Skrzyński, A. Szczepkiewicz, A. Taberner, R. Tomasetti, B. Turcksin, P. Zhener, Kokkos 5.1.0 (Mar. 2026). doi:10.5281/zenodo.19324404.
- [10] J. Ekelund, S. Markidis, I. Peng, Boosting Performance of Iterative Applications on GPUs: Kernel Batching with CUDA Graphs, in: 2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2025, pp. 70–77. doi:10.1109/PDP66500.2025.00019.
- [11] L. Toledo, P. Valero-Lara, J. S. Vetter, A. J. Peña, Towards Enhancing Coding Productivity for GPU Programming Using Static Graphs, Electronics 11 (9) (2022). doi:10.3390/electronics11091307.

- [12] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhersch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, S. Chintala, PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2024, p. 929–947. doi:10.1145/3620665.3640366.
- [13] A. Ghosh, A. Nayak, A. Panwar, A. Basu, PyGraph: Robust Compiler Support for CUDA Graphs in PyTorch (2025). arXiv:2503.19779. URL <https://arxiv.org/abs/2503.19779>
- [14] M. Doijade, A. Alekseenko, A. Brown, A. Gray, S. Páll, Redesigning GROMACS Halo Exchange: Improving Strong Scaling with GPU-initiated NVSHMEM, in: Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2025, p. 1314–1329. doi:10.1145/3731599.3767508.
- [15] S. Jones, CUDA: New Features and Beyond, <https://www.nvidia.com/en-us/on-demand/session/gtc26-s81859/>, at NVIDIA GTC 2026 (2026).
- [16] N. Otterness, J. Anderson, Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices", in: Proceedings of the 29th International Conference on Real-Time Networks and Systems, 2021, pp. 24–34. doi:10.1145/3453417.3453432.
- [17] G. Gilman, S. S. Ogden, T. Guo, R. J. Walls, Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent

- Kernels, ACM SIGMETRICS Perform. Eval. Rev. 48 (3) (2021) 81–88.
doi:10.1145/3453953.3453972.
- [18] J. Wong, Add experimental::cudaevent for fine-grained stream dependencies, <https://github.com/kokkos/kokkos/pull/9122> (2026).
 - [19] R. Tomasetti, core(graph): then_parallel_reduce correctness for interfering nodes, <https://github.com/kokkos/kokkos/issues/9271> (2026).
 - [20] NVIDIA, CUDA Programming Guide: Release 13.2, accessed: 2026-05-05 (2026).
 - [21] R. Armstrong, R. Nertney, J. Marusarz, F. Oh, R. Jensen, CUDA toolkit 12.4 enhances support for NVIDIA Grace Hopper and confidential computing, <https://developer.nvidia.com/blog/cuda-toolkit-12-4-enhances-support-for-nvidia-grace-hopper-and-confidential-computing/>, accessed: 2026-08-14 (2024).
 - [22] R. Tomasetti, core(graph): allow submission onto an arbitrary exec space instance, <https://github.com/kokkos/kokkos/pull/7249/> (2024).
 - [23] R. Tomasetti, core(graph): submit on default exec by default to align with kokkos semantics, <https://github.com/kokkos/kokkos/pull/8365> (2025).
 - [24] R. Tomasetti, core(graph): promote instantiate to public API, <https://github.com/kokkos/kokkos/pull/7240/> (2024).
 - [25] R. Tomasetti, graph(fix): nodes must be stored by the graph, whatever the backend, <https://github.com/kokkos/kokkos/pull/7619> (2024).
 - [26] R. Tomasetti, core(graph): allow building kokkos::graph object directly, <https://github.com/kokkos/kokkos/pull/8108> (2025).
 - [27] R. Tomasetti, core(graph): node properties and device handle, <https://github.com/kokkos/kokkos/pull/8191> (2026).

- [28] R. Tomasetti, core(graph): then node, <https://github.com/kokkos/kokkos/pull/7629> (2025).
- [29] R. Tomasetti, core(graph): enforce unit launch bound, <https://github.com/kokkos/kokkos/pull/8192> (2025).
- [30] R. Tomasetti, core(graph): add host node, <https://github.com/kokkos/kokkos/pull/8075> (2025).
- [31] P. Reble, et al., sycl_ext_oneapi_graph, https://github.com/intel/llvm/blob/5fb2603b4882a7059abb64409597f51cd319c826/sycl/doc/extensions/experimental/sycl_ext_oneapi_graph.asciidoc#82-sycl-graph-terminology, accessed: 2026-05-05 (2026).
- [32] R. Tomasetti, core(graph): capture, <https://github.com/kokkos/kokkos/pull/7552> (2025).
- [33] R. Tomasetti, graph: allow access to native graph object, <https://github.com/kokkos/kokkos/pull/6904> (2024).
- [34] R. Tomasetti, core(graph): allow construction from native graph (interop), <https://github.com/kokkos/kokkos/pull/7664> (2025).
- [35] R. Tomasetti, core(graph): rename interoperability graph access with <backend>_ prefix, <https://github.com/kokkos/kokkos/pull/9159> (2026).
- [36] R. Tomasetti, core(graph): add vendor node, <https://github.com/kokkos/kokkos/pull/8480> (2025).
- [37] R. Tomasetti, core(graph): allow access to vendor node, <https://github.com/kokkos/kokkos/pull/9147> (2026).
- [38] Advanced Micro Devices, ihipgraphupload, https://github.com/ROCm/clang/blob/6dec1ecee9826c68e0a0b1c114f49ff27efccd2/hipamd/src/hip_graph.cpp#L27-L36, accessed: 2026-06-22 (2026).

- [39] Advanced Micro Devices, isgraphexecvalid, https://github.com/ROCm/clr/blob/6dec1ecee9826c68e0a0b1c114f49ff27efccd2/hipamd/src/hip_graph_internal.cpp#L933-L939, accessed: 2026-06-22 (2026).
- [40] H. Hoffman, F. Oh, Constant Time Launch for Straight-Line CUDA Graphs and Other Performance Enhancements, <https://developer.nvidia.com/blog/constant-time-launch-for-straight-line-cuda-graphs-and-other-performance-enhancements/>, accessed: 2026-06-21 (2024).
- [41] R. Tomasetti, M. Arnst, Efficiently implementing FE boundary conditions using stream-orchestrated execution on GPU, <https://orbi.uliege.be/handle/2268/314520>, at SIAM Conference on Parallel Processing for Scientific Computing (2024).
- [42] K. Rupp, J. Weinbub, A. Jüngel, T. Grasser, Pipelined Iterative Solvers with Kernel Fusion for Graphics Processing Units, *ACM Trans. Math. Softw.* 43 (2) (2016) 1–27. doi:10.1145/2907944.
- [43] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd Edition, Society for Industrial and Applied Mathematics, Philadelphia, 2003. doi:10.1137/1.9780898718003.
- [44] S. Chien, I. Peng, S. Markidis, Performance evaluation of advanced features in CUDA unified memory, in: 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), 2019, pp. 50–57. doi:10.1109/MCHPC49590.2019.00014.
- [45] H. Kim, J. Sim, P. Gera, R. Hadidi, H. Kim, Batch-aware unified memory management in GPUs for irregular workloads, in: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, p. 1357–1370. doi:10.1145/3373376.3378529.
- [46] G. Schieffer, J. Wahlgren, J. Ren, J. Faj, I. Peng, Harnessing integrated CPU-GPU system memory for HPC: a first look into Grace Hopper, in:

- Proceedings of the 53rd International Conference on Parallel Processing, 2024, p. 199–209. doi:10.1145/3673038.3673110.
- [47] P. Wriggers, *Nonlinear Finite Element Methods*, Springer Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-71001-1.
 - [48] R. P. Pawlowski, E. T. Phipps, A. G. Salinger, S. J. Owen, C. M. Siefert, M. L. Staten, Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, part ii: Application to partial differential equations, *Sci. Program.* 20 (3) (2012). doi:10.3233/SPR-2012-0351.
 - [49] M. Garcia-Gasulla, G. Houzeaux, R. Ferrer, A. Artigues, V. López, J. Labarta, M. Vázquez, MPI+X: task-based parallelisation and dynamic load balance of finite element assembly, *Int. J. Comput. Fluid Dyn.* 33 (3) (2019) 115–136. doi:10.1080/10618562.2019.1617856.
 - [50] M. Mayr, A. Heinlein, C. Glusa, S. Rajamanickam, M. Arnst, R. A. Bartlett, L. Berger-Vergiat, E. G. Bowman, K. Devine, G. Harper, M. Heroux, M. Hoemmen, J. J. Hu, B. Kelley, K. Kim, D. P. Kouri, P. Kurberry, K. Liegeois, C. C. Ober, R. P. Pawlowski, C. Pearson, M. Perego, E. Phipps, D. Ridzal, N. V. Roberts, C. M. Siefert, H. K. Thornquist, R. Tomasetti, C. R. Trott, R. S. Tuminaro, J. M. Willenbring, M. M. Wolf, I. Yamazaki, Trilinos: Enabling scientific computing across diverse hardware architectures at scale, *ACM Trans. Math. Softw.* 52 (2) (2026). doi:10.1145/3802822.
 - [51] A. Gawlik, Light interactions with periodic nanoline arrays for nanoelectronic applications, Ph.D. thesis, Katholieke Universiteit Leuven and Wrocław University of Science and Technology (2021).
 - [52] J. Bogdanowicz, T. Nuytten, A. Gawlik, S. Sergeant, Y. Oniki, P. P. Gowda, H. Mertens, A.-L. Charley, Taming the distribution of light in gate-all-around semiconductor devices, *Nano Lett.* 24 (4) (2024) 1191–1196. doi:10.1021/acs.nanolett.3c04033.

- [53] V. Brunini, J. Clausen, M. F. Hoemmen, A. Kucala, C. R. Trott, M. Howard, Runtime polymorphism in Kokkos applications, <https://www.osti.gov/biblio/1592283>, accessed: 2026-06-22 (2019).
- [54] M. Arnst, R. Tomasetti, Design patterns and performance analysis of polymorphism in multiphysics FE assembly on GPU, <https://orbi.uliege.be/handle/2268/314521>, at SIAM Conference on Parallel Processing for Scientific Computing (2024).
- [55] O. Amoros, A. Andaluz, J. Nunez, A. J. Pena, The Fused Kernel Library: A C++ API to Develop Highly-Efficient GPU Libraries (2025). [arXiv:2508.07071](https://arxiv.org/abs/2508.07071).
URL <https://arxiv.org/abs/2508.07071>
- [56] E. Crawford, P. Reble, B. Tracy, J. Miller, Towards Deferred Execution of a SYCL Command Graph, in: Proceedings of the 2023 International Workshop on OpenCL, 2023, pp. 1–2. doi:10.1145/3585341.3585375.
- [57] V. Pérez, L. Sommer, V. Lomüller, K. Narasimhan, M. Goli, User-driven Online Kernel Fusion for SYCL, ACM Trans. Archit. Code Optim. 20 (2) (2023) 1–25. doi:10.1145/3571284.
- [58] S. Jones, Dynamic Control Flow in Graphs, https://hihat.opencommons.org/images/f/f7/CUDA_Graphs_Update_2023.pdf, accessed: 2026-06-21 (2023).
- [59] R. Tomasetti, M. Arnst, D. Lebrun-Grandié, Kokkos::Graph : Asynchronous execution, use cases and adoption, <https://orbi.uliege.be/handle/2268/331046>, at Kokkos Tea Time (2024).

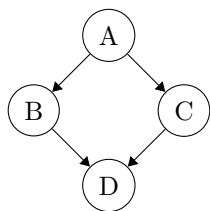


Figure 1: Asynchronous execution in Kokkos: Diamond-shaped dependency pattern.

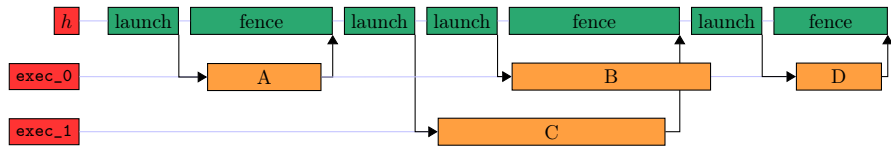


Figure 2: Asynchronous execution in `Kokkos`: Possible execution timeline for the diamond-shaped dependency pattern using the execution space abstraction as in Listing 2.

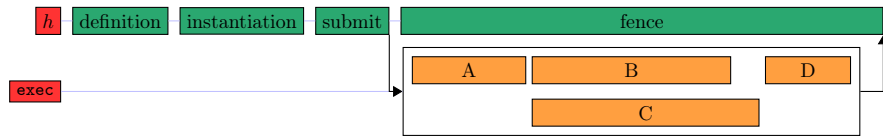


Figure 3: Asynchronous execution in `Kokkos`: Possible execution timeline for the diamond-shaped dependency pattern using the graph abstraction as in Listing 3.

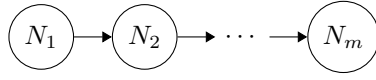
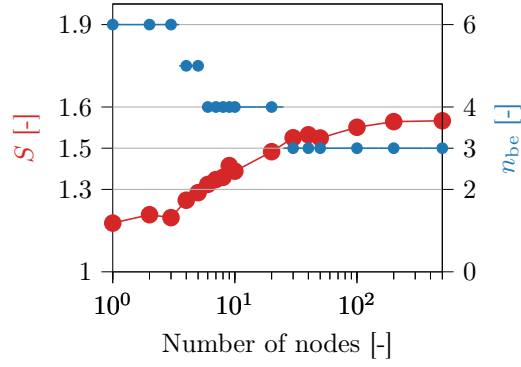
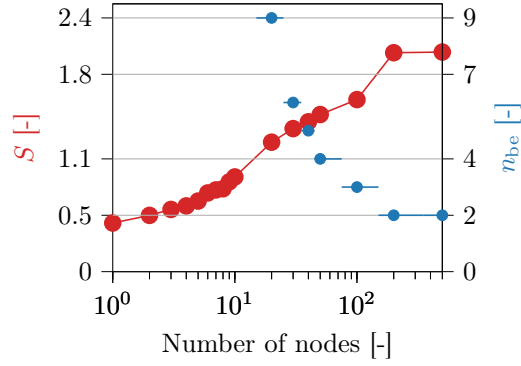


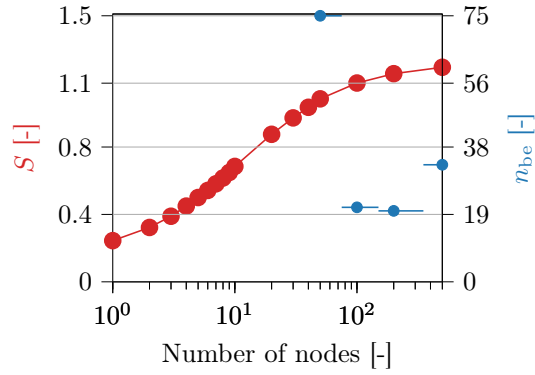
Figure 4: Performance evaluation across backends: A straight-line graph of m operations.



(a) CUDA GH200.



(b) HIP SPR-MI300X.



(c) SYCL SPR-PVC.

Figure 5: Performance evaluation across backends: Asymptotic speedup S (red, dots, connected solid line) and break-even point n_{be} (blue, dots, interrupted solid line) as a function of the number of operations m for the straight-line graph.

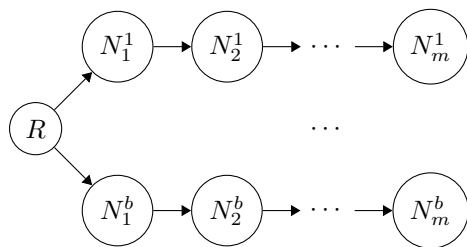


Figure 6: Performance evaluation across backends: A fork graph with b branches of m operations each. R is the root.

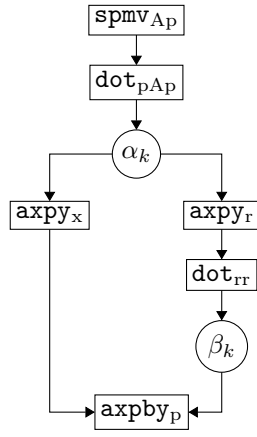
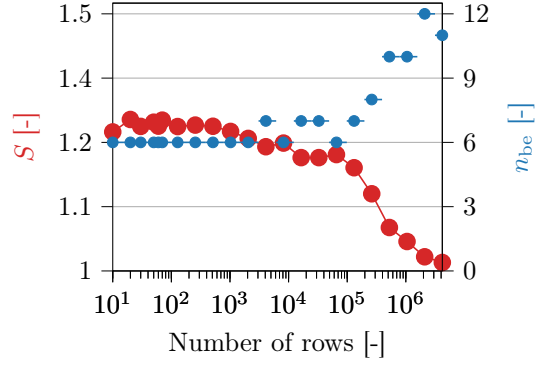
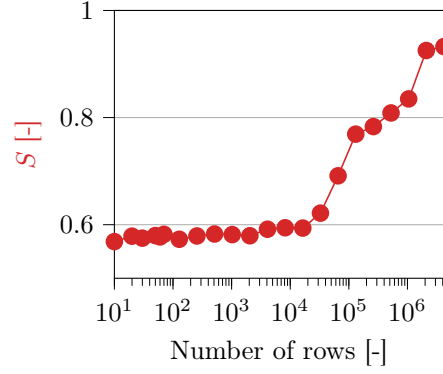


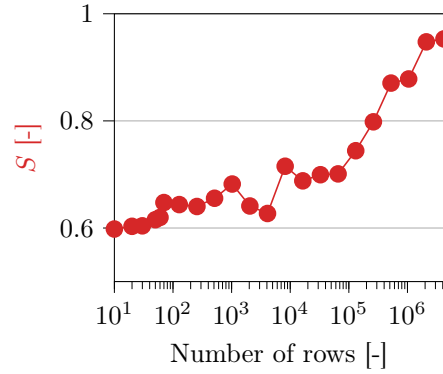
Figure 7: Performance evaluation across backends: CG algorithm graph [43, Section 6.7].



(a) CUDA GH200.



(b) HIP SPR-MI300X.



(c) SYCL SPR-PVC.

Figure 8: Performance evaluation across backends: Asymptotic speedup S (red, dots, connected solid line) and break-even point n_{be} (blue, dots, interrupted solid line) as a function of the size of the linear system for the CG solver.

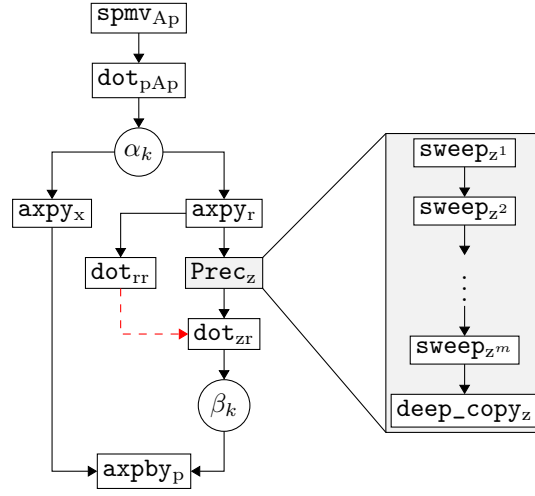
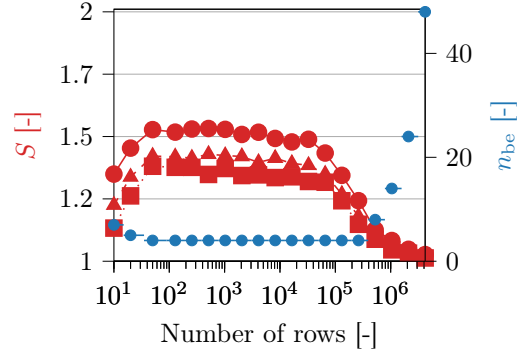
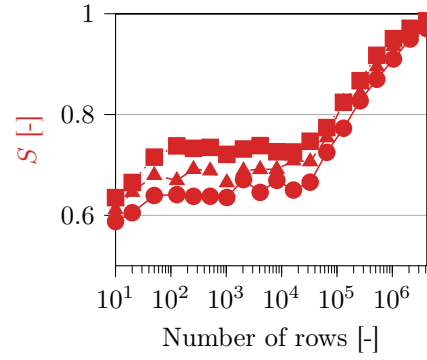


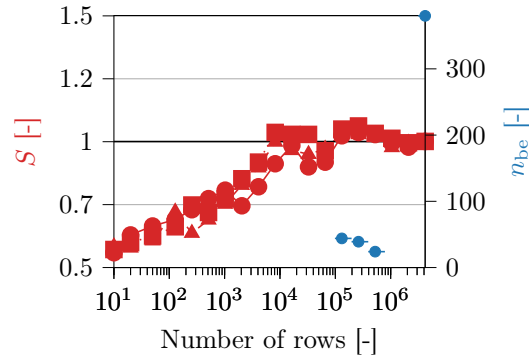
Figure 9: Performance evaluation across backends: PCG algorithm graph [43, Section 9.2], with Jacobi stationary iteration as the preconditioner. Note that `deep_copyz` is only present when m is odd.



(a) CUDA GH200.



(b) HIP SPR-MI300X.



(c) SYCL SPR-PVC.

Figure 10: Performance evaluation across backends: Asymptotic speedup S as a function of the number of rows for the PCG solver using $m = 4$ sweeps (red, circles), $m = 8$ sweeps (red, triangles), $m = 12$ sweeps (red, squares). If S is beneficial for $m = 8$ sweeps, the corresponding break-even point n_{be} is shown (blue, dots).

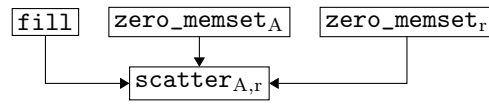


Figure 11: Performance evaluation across backends: Assembly algorithm graph.

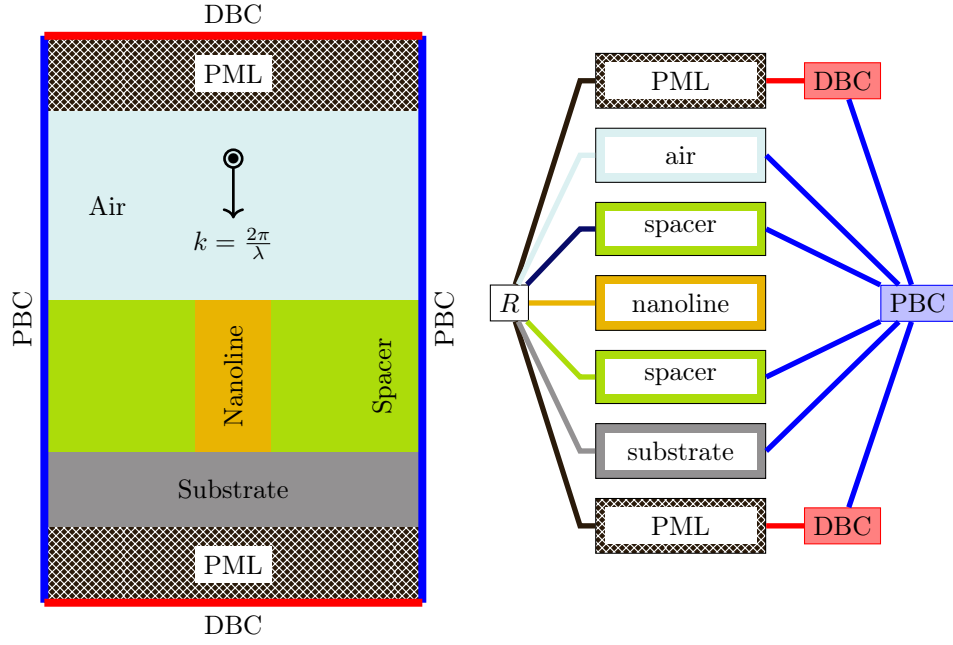


Figure 12: Application to FE assembly: Schematic views of the computational domain [51, Figure 4.1] (left) and the operations organised as a DAG (right), with the root (R), the perfectly matched layer (PML), Dirichlet boundary condition (DBC) and periodic boundary condition (PBC) subregions, for a frequency-domain electromagnetic wave scattering problem.

Vendor	Queue	Graph
CUDA	<code>cudaStream_t</code>	<code>cudaGraph_t</code>
HIP	<code>hipStream_t</code>	<code>hipGraph_t</code>
SYCL	<code>sycl::queue</code>	<code>command_graph</code>

Table 1: Asynchronous execution in Kokkos: Vendor asynchronous execution primitives. For each backend, the Kokkos execution space wraps a vendor queue. `Kokkos::Graph` is a templated type that, when specialised for a given backend, maps to the corresponding vendor graph type. The full SYCL graph type is `sycl::ext::oneapi::experimental::command_graph`.

	CUDA	HIP	SYCL
T^E	28	33	31
T_d^G	11	13	14
T_i^G	11	41	32
T_{s0}^G	24	52	49
T_{sr}^G	20	38	48
S	1.37	0.88	0.64
n_{be}	4	-	-

Table 2: Performance evaluation across backends: Elapsed times [μ s] for a straight-line graph of $m = 10$ operations. Please note that S and n_{be} were calculated using the unrounded timing measurements. The dashes mean that no finite break-even point exists because $S \leq 1$.

	CUDA	HIP	SYCL
T^E	95	128	129
T_d^G	46	67	79
T_i^G	39	3043	111
T_{s0}^G	36	108	115
T_{sr}^G	25	87	112
S	3.84	1.47	1.15
n_{be}	2	77	12

Table 3: Performance evaluation across backends: Elapsed times [μ s] for a fork graph of $b = 4$ branches of $m = 10$ operations each.

system size	model	CUDA	HIP	SYCL
1000	execution space	89	89	100
	graph	57	121	144
5000	execution space	440	458	521
	graph	287	594	663

Table 4: Performance evaluation across backends: Overall time to solution [μ s] for the Newton solver for two sizes of the system of non-linear equations.

```
Kokkos::Cuda exec{};

Kokkos::parallel_for(Kokkos::RangePolicy(exec, 0, N),
                    functor);

// Overlapping work on the calling thread.

exec.fence();
```

Listing 1: Asynchronous execution in Kokkos: An asynchronous operation is enqueued into an execution space instance `exec` for the CUDA backend. The caller may perform overlapping work before fencing `exec` to wait for completion.

```

Kokkos::Cuda exec{};
auto [exec_0, exec_1] =
    Kokkos::Experimental::partition_space(exec, 1, 1);

Kokkos::parallel_for(
    Policy_A(exec_0, ...), functor_A);

exec_0.fence();

Kokkos::parallel_for(
    Policy_C(exec_1, ...), functor_C);
Kokkos::parallel_for(
    Policy_B(exec_0, ...), functor_B);

exec_1.fence();

Kokkos::parallel_for(
    Policy_D(exec_0, ...), functor_D);

exec_0.fence();

```

Listing 2: Asynchronous execution in Kokkos: Expressing the diamond-shaped dependency pattern (Fig. 1) using the execution space abstraction with two execution space instances obtained by partitioning.

```

Kokkos::Cuda exec{};

// Definition stage.
Kokkos::Experimental::Graph graph{};
auto root = graph.root_node();
auto node_A = root.then_parallel_for(Policy_A(...), functor_A);
auto node_B = node_A.then_parallel_for(Policy_B(...), functor_B);
auto node_C = node_A.then_parallel_for(Policy_C(...), functor_C);
auto node_D = Kokkos::Experimental::when_all(
    node_B, node_C
).then_parallel_for(Policy_D(...), functor_D);

// Instantiation stage.
graph.instantiate();

// Submission stage.
while(...) {
    graph.submit(exec);

    // Overlapping work on the calling thread.

    exec.fence();
}

```

Listing 3: Asynchronous execution in Kokkos: Expressing the diamond-shaped dependency pattern (Fig. 1) using the graph abstraction with a `Kokkos::Graph` on the default device.

```

auto graph = Kokkos::Experimental::create_graph(
    exec_graph, [=](auto root) {
        auto node = root.then_parallel_for(
            label,
            Kokkos::RangePolicy(exec_node, 0, N),
            functor_t{data});
        ...
    });

graph.submit();
graph.get_execution_space().fence();

```

Listing 4: Resolving underspecified Kokkos::Graph semantics: Defining and submitting a graph with Kokkos 3 [8].

```

auto dev_handle = Kokkos::Experimental::get_device_handle(exec);
Kokkos::Experimental::Graph graph{dev_handle};

...

auto node = graph.root_node().then_parallel_for(
    Kokkos::Experimental::node_props(label, dev_handle),
    Kokkos::RangePolicy<Exec>(0, N),
    functor_t{data}
);

...

graph.instantiate();

...

graph.submit(exec_submit);

```

Listing 5: Resolving underspecified `Kokkos::Graph` semantics: Defining, instantiating and submitting a graph as of Kokkos 5.1 [9].

```

struct ThenFunctor {
    ViewType data;

    KOKKOS_FUNCTION void operator()() const {
        ...
    }
};

...

auto node = pred.then(<node props>, ThenFunctor{...});

```

Listing 6: Extending Kokkos::Graph capabilities: Defining a `then` node.

```
struct ThenHostFunctor {  
    std::vector<Type> data;  
  
    void operator()() const {  
        ...  
    }  
};  
  
...  
  
auto node = pred.then_host(<label>, ThenHostFunctor{...});
```

Listing 7: Extending Kokkos::Graph capabilities: Defining a `then_host` node.

```

struct CudaCaptureFunctor {
    ViewType v;
    ResultViewType mag;
    std::shared_ptr<cublasContext> handle;

    void operator()(const Kokkos::Cuda& exec) const {
        custom_kernel<<<..., exec.cuda_stream()>>>(v);
        cublasSetStream(handle.get(), exec.cuda_stream());
        cublasDnrm2(
            handle.get(), v.size(), v.data(), 1, mag.data());
    }
};

...

auto node =
    pred.cuda_capture(exec_capture, CudaCaptureFunctor{...});

```

Listing 8: Extending Kokkos::Graph capabilities: Example of a `cuda_capture` node. The resulting Kokkos::Graph capture node will comprise at least two CUDA nodes: one for `custom_kernel` and one or more for the cuBLAS call.

```

cudaGraph_t outer;
...

cudaGraphConditionalHandle cond_handle;
...

// Create and add the while conditional node.
cudaGraphNodeParams while_params
    = { cudaGraphNodeTypeConditional };
while_params.conditional.handle = cond_handle;
while_params.conditional.type = cudaGraphCondTypeWhile;
...
cudaGraphNode_t while_node;
cudaGraphAddNode(
    &while_node, outer, nullptr, nullptr, 0, &while_params);

// Get the body graph of the conditional node
// and wrap it into a Kokkos::Graph.
cudaGraph_t inner_cuda = while_params.conditional.phGraph_out[0];
Kokkos::Experimental::Graph<Kokkos::Cuda> inner{
    device_handle, inner_cuda};

// Populate the body graph with Kokkos nodes.
auto node = inner.root_node().then(...);
...

```

Listing 9: Extending Kokkos::Graph capabilities: Wrapping a CUDA `while` node body graph in a Kokkos::Graph to populate it with Kokkos nodes. Inspired by [20, Section 4.2.4.4].

```

// axpby(exec, ...);
template <Kokkos::ExecutionSpace Exec, ...>
void axpby(const Exec& exec, ...) {
    Kokkos::parallel_for(
        Kokkos::RangePolicy(exec, ...),
        AxpbyFunctor{...});
}

// auto node = axpby(predecessor, ...);
template <typename Predecessor, ...>
auto axpby(const Predecessor& pred, ...) {
    return pred.then_parallel_for(
        Kokkos::RangePolicy(...),
        AxpbyFunctor{...});
}

```

Listing 10: Discussion: Dual API for Kokkos Kernels with reuse of the computational operation functor `AxpbyFunctor`, adapted from [59].