Graph abstraction for efficient scheduling of asynchronous workloads on GPU

CExA Coffee Time

Tomasetti Romin[1] Arnst Maarten[1] Lebrun-Grandié Damien[2]

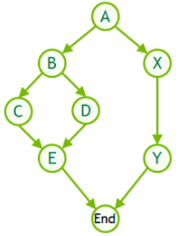
[1]University of Liège, Belgium [2]Oak Ridge National Laboratory, TN

November 4th, 2024

Context

Many computational physics simulations need to efficiently schedule **asynchronous** workloads:

- FEM assembly
- linear algebra
- your routines 😉



Reproduced from [Gra19].

Asynchronous execution models

1. Execution space instances
2. `Kokkos::Graph`

Abstraction of the day 📅

`Kokkos::Graph`

Outline

Why bother with `Kokkos::Graph`?

Gloomy points 😒

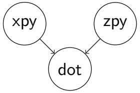
1. *Ad hoc* scheduling of many asynchronous workloads is an additional burden to your code base as
   - it will kill readability,
   - and cause headaches for portability.
2. Handmade solutions might not fully exploit the available execution resources.

A graph abstraction comes to your rescue! 🛟

1. Describe your computational graph to `Kokkos::Graph`:
   - Semantics are clear.
   - You get portability.
2. Exposing the **whole computational graph** to the compiler/driver ahead of execution enables **as many optimisations as possible**.

Example: vanilla Kokkos *versus* Kokkos::Graph

Two AXPBY's followed by a dot product


$$\begin{tikzcd} \text{xpy} \arrow[dr] & & \text{zpy} \arrow[dl] \\ & \text{dot} & \end{tikzcd}$$

```
// Async. with execution space instances.
const Kokkos::Cuda exec_1 {}, exec_2 {};

using policy_t = Kokkos::RangePolicy<Kokkos::Cuda>;

Kokkos::parallel_for(   policy_t(exec_1, 0, N), Axpby{x, y, alpha, beta});
Kokkos::parallel_for(   policy_t(exec_2, 0, N), Axpby{z, y, alpha, gamma});
exec_2.fence();
Kokkos::parallel_reduce(policy_t(exec_1, 0, N), Dotp{x, z}, dotp);
```

```
// Async. with Kokkos::Graph.
const Kokkos::Cuda exec {};

auto graph = Kokkos::Experimental::create_graph(exec, [&](const auto& root) {
   auto xpy = root.then_parallel_for(N, Axpby{x, y, alpha, beta});
   auto zpy = root.then_parallel_for(N, Axpby{z, y, alpha, gamma});

   Kokkos::Experimental::when_all(xpy, zpy).then_parallel_reduce(
       N, Dotp{x, z}, dotp
   );
});

graph.submit(exec);
```

Inspired by [Lif].

What is `Kokkos::Graph` exactly ?

How to think about it?

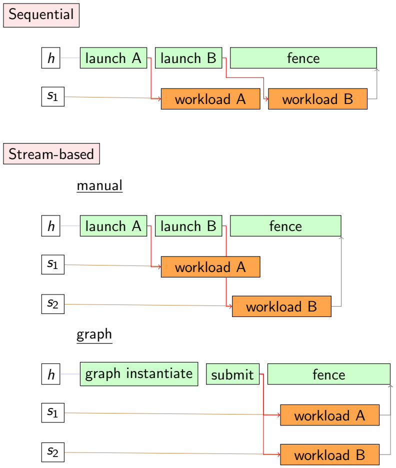
From a semantic standpoint, `Kokkos::Graph` must be used in a three-phase way (simplified):

🏗	definition	describe your DAG of workloads (topology)
♨	instantiation	check DAG for flaws and prepare the executable graph
🔥	submission	launch the executable graph

Implementation details

- Portable wrapper around:
  - `cudaGraph_t`
  - `hipGraph_t`
  - `sycl::ext::oneapi::experimental::command_graph`
- Default implementation for "unsupported" backends.

Under the hood: graph overhead and amortization



Reproduced from [TA24].

`Cuda` API calls aren't for free:

- launching a kernel on a stream
- building and submitting a graph

When to use a graph?

- workloads are organised as a DAG
- the performance bottleneck is CPU scheduling overhead rather than GPU execution

Amortize graph definition and instantiation - possibly across multiple submissions - until you beat the *manual stream-based* implementation.

Past and current research focus

Some recent work (emphasis mine):

- core(graph): **promote instantiate** to public API (Aug. 22 - 24)
- core(graph): allow submission onto an **arbitrary exec** space instance (Aug. 28 - 24)
- graph(fix): **defaulted graph submit** control flow (Sep. 6 - 24)
- core(graph): allow *create_graph* **without closure** (Sep. 10 - 24)
- graph: allow access to **native graph** object (Oct. 7 - 24)
- graph(diagnostic): enable compile-time diagnostic of **illegal reduction target** (Oct. 18 - 24)

The following content is new.

Future extensions / opportunities worth a try

Missing features for broader adoption 🌶

Integrate more backend features in `Kokkos::Graph`[3]:

- ✅ add a node depending on a runtime condition (`MPI` partitioning)
- 🔥 graph capture (`cuSPARSE`, external libraries not using `Kokkos`)
- 🕐 enable/disable a node between 2 successive launches
- 🕐 conditional if/while (routine branching, solvers like *CG*)
- 🕐 memory node (parallel reduction target)
- 🕐 update kernel parameters (range policy bounds)
- 🕐 node priority (longer kernels scheduled first)
- 🕐 host node (MPI exchange?)

[3] Note that not all backends support all the above (except `Cuda`).

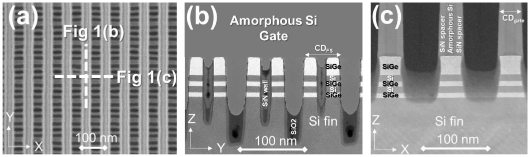
Computational metrology in semi-conductor assembly lines

Optical metrology

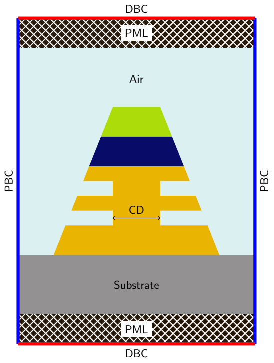
Use light to gather data about the physical properties of objects.

Focus

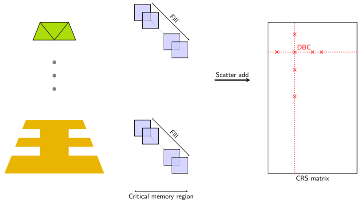
Swift FEM computed samples are needed to train a probabilistic inverse problem method.



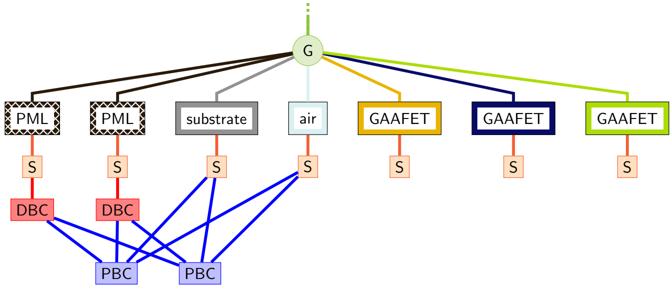
GAAFET (forksheet) [BNG+24]

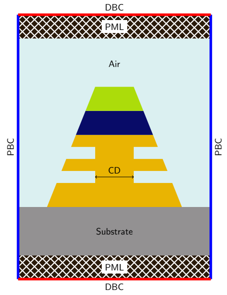


Organizing (in)dependent computations as a DAG graph



- Dependencies between workloads are clearly expressed.
- Once predecessor workloads are done, child nodes can run concurrently, once resources are available.





References I

Janusz Bogdanowicz, Thomas Nuytten, Andrzej Gawlik, Stefanie Sergeant, Yusuke Oniki, Pallavi Puttarame Gowda, Hans Mertens, and Anne-Laure Charley, *Taming the Distribution of Light in Gate-All-Around Semiconductor Devices*, Nano Letters **24** (2024), no. 4, 1191–1196.

A. Gray and S. Páll, *A Guide to CUDA Graphs in GROMACS 2023*, https://developer.nvidia.com/blog/a-guide-to-cuda-graphs-in-gromacs-2023/, 2023.

A. Gray, *Getting Started with CUDA Graphs*, https://developer.nvidia.com/blog/cuda-graphs/, 2019.

J. Lifflander, *Benchmarking Kokkos Graphs*, https://hihat.opencommons.org/images/1/1a/Kokkos-graphs-presentation.pdf.

References II

Romin Tomasetti and Maarten Arnst, *Efficiently implementing fe boundary conditions using stream-orchestrated execution on gpu*, F.R.S.-FNRS - Fonds de la Recherche Scientifique [BE], 07 March 2024 (English).