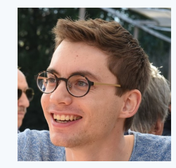
Graph Abstraction for Efficient Scheduling of Asynchronous Workloads on GPU

Tomasetti R., Arnst M.

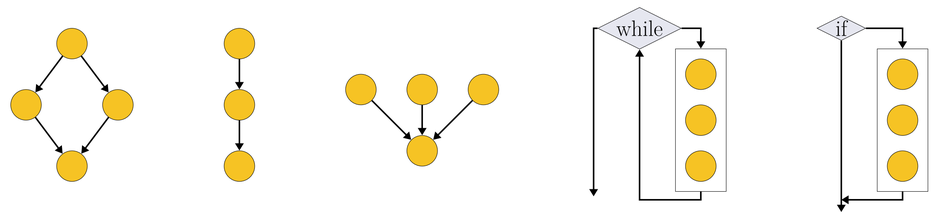
Computational and stochastic modeling, University of Liège



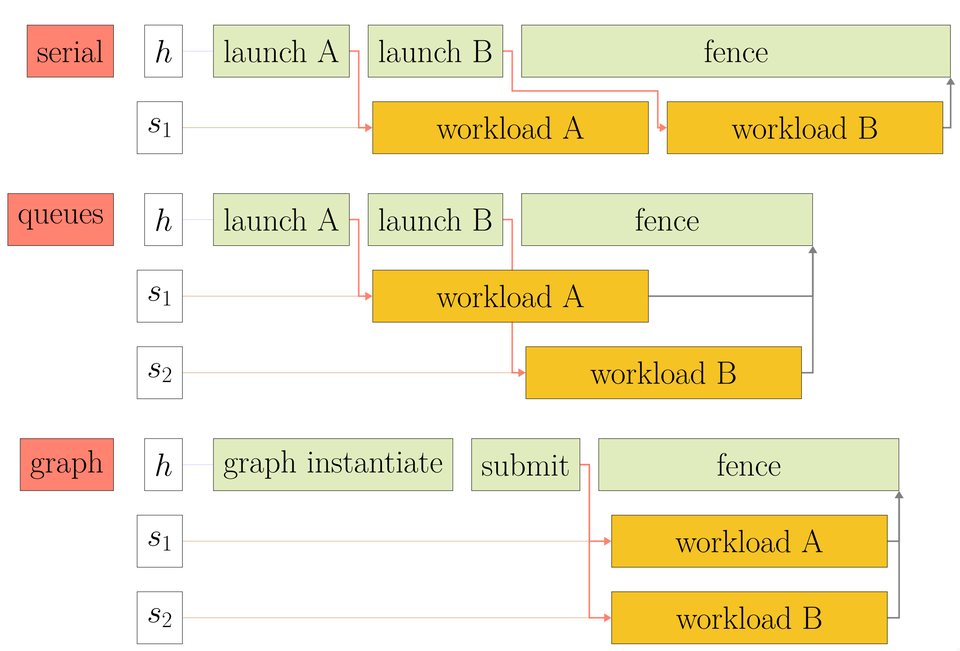
romin.tomasetti
@uliege.be

Performance portable graph abstraction

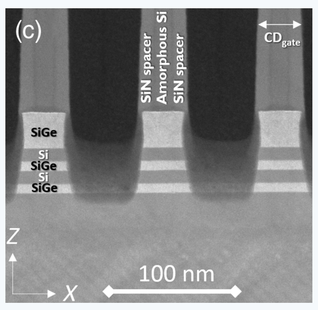
Direct Acyclic Graphs (DAGs) of asynchronous workloads arise in many applications.

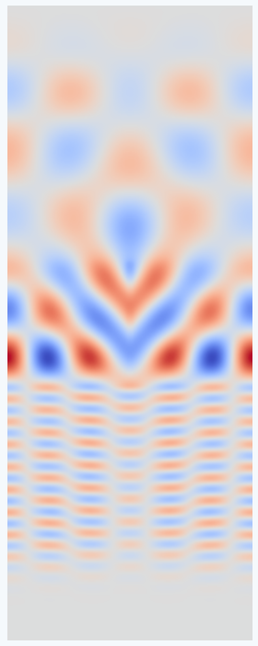


Such DAGs can be mapped to queue-based execution models in several ways.



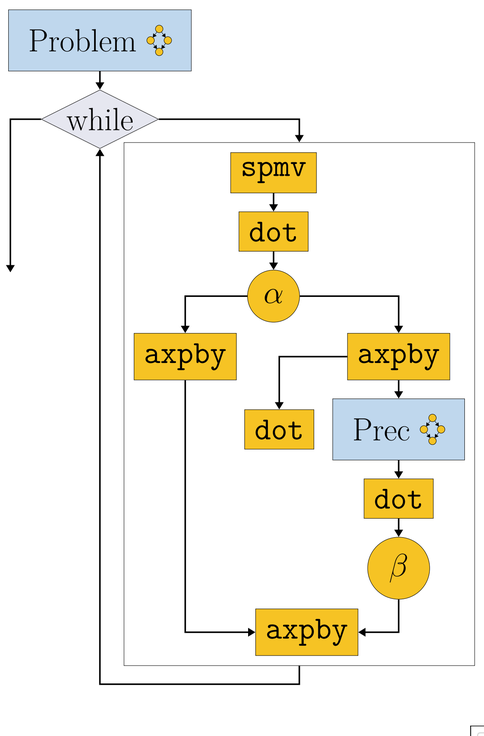
Developing a performance portable graph abstraction that can run efficiently across many hardware platforms in the evolving HPC ecosystem—like CPUs, GPUs, and specialized accelerators—without needing major code rewrites is a key challenge.





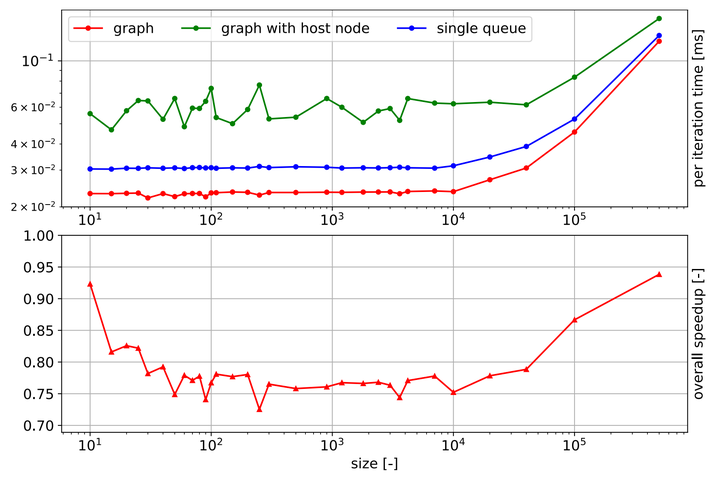
HPC applicability and benefits

Applies to many common HPC workloads: problem discretization, solver, preconditioner, ...



- ✅ Consistent benefits:
  - descriptive
  - composable
  - cleaner software design
  - easier concurrent execution
  - better use of hardware

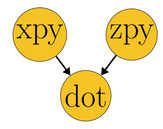
- ⚡ Potential speedup depends on:
  - graph setup amortization
  - DAG topology
  - node types and workloads
  - device and software stack



Kokkos framework

`Kokkos` is a leading performance portability framework written in `C++` targeting many backends, such as `Cuda` and `HIP`, among others.

Two AXPBY's followed by a dot product can be scheduled either using two *execution space instances* or with a *graph*.




```cpp
// 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);
```

```cpp
// 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.instantiate();
graph.submit(exec);
```

Ad hoc scheduling of many asynchronous workloads is an additional burden to your code base. It *might* kill readability and *will* give headaches for portability.
Moreover, handmade solutions might not fully exploit the available execution resources.

Using `Kokkos::Graph` to describe your computational graph:

- ✅ makes the code semantics clearer
- ✅ eases portability
- ✅ exposes the whole computational graph to the compiler/driver ahead of execution, thus enabling as many optimizations as possible

Extending `Kokkos::Graph`

- 🛠️ Support a common subset of vendor graph APIs.
  Vendor support varies — `Cuda`'s API is the most extensive (e.g., supports *if/else* and *while*).
- 📦 Increasingly allow entire control flow to reside within the graph.
- 🤹 Host-side workloads and interoperability are challenging.

🛠️ `then` Some calculations do not require a parallel region but need to happen on device.

```cpp
auto node = predecessor.then("B", KOKKOS_FUNCTION { ... });
```

🛠️ `capture` Interoperability with non-`Kokkos` code requires the `capture` mechanism. It is only supported for backends that natively implement it.

```cpp
auto node = predecessor.cuda_capture(
    exec,
    [... data to keep alive ...](const Kokkos::Cuda& exec_) {
        ... kernel dispatches to be captured ...
    }
);
```

The `capture` node extends the lifetime of needed data until graph destruction.

```cpp
auto node = predecessor.cuda_capture(
    exec,
    [handle, matrix, vector, result, alpha, beta](const Kokkos::Cuda& exec_) {
      CHECK_CUBLAS_CALL(cublasSetStream(handle.get(), exec_.cuda_stream()));
      CHECK_CUBLAS_CALL(cublasDgemv(
        handle.get(),
        CUBLAS_OP_N,
        vector.size(), result.size(), &alpha, matrix.data(),
        1, vector.data(), 1,
        &beta, result.data(), 1
      ));
    }
  );
```

🏁 `Kokkos::Graph` is actively developed to foster broader adoption.

Acknowledgements Tomasetti R. is a FRIA grantee of the Fonds de la Recherche Scientifique – FNRS.

uliegecsm.github.io

kokkos.org/kokkos-core-wiki