Graph-based dispatching of FE compute workloads

Efficiently implementing FE boundary conditions using stream-orchestrated execution on GPU

Tomasetti Romin Arnst Maarten

University of Liège, Belgium

March 7th, 2024

siam 2024 Conference on Parallel Processing for Scientific Computing

LIÈGE université

fnrs FREEDOM TO RESEARCH

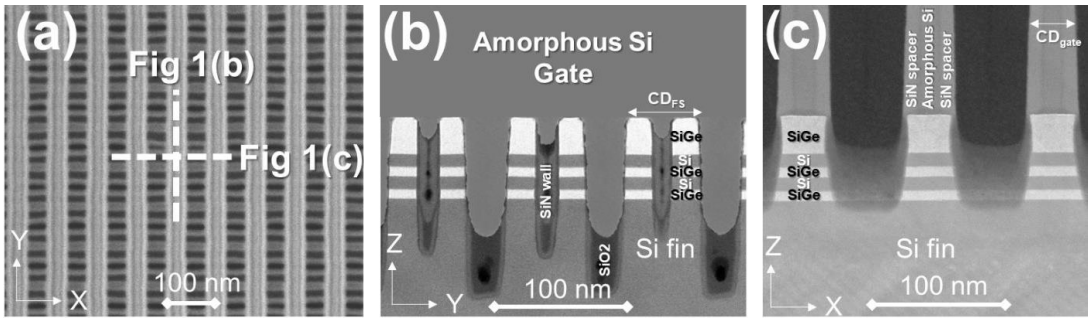
Computational metrology in semi-conductor assembly lines

Optical metrology

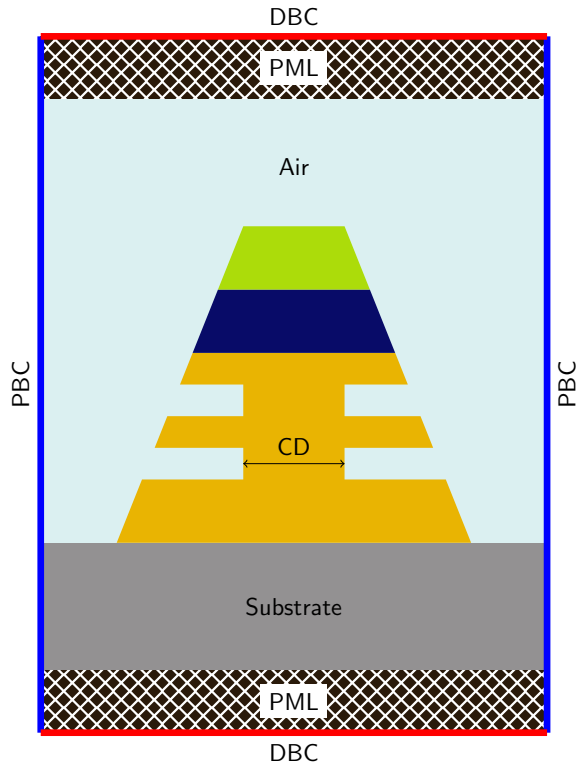
Use light to gather data about the physical properties of objects.

Focus

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



GAAFET (forksheet) [BNG+24]



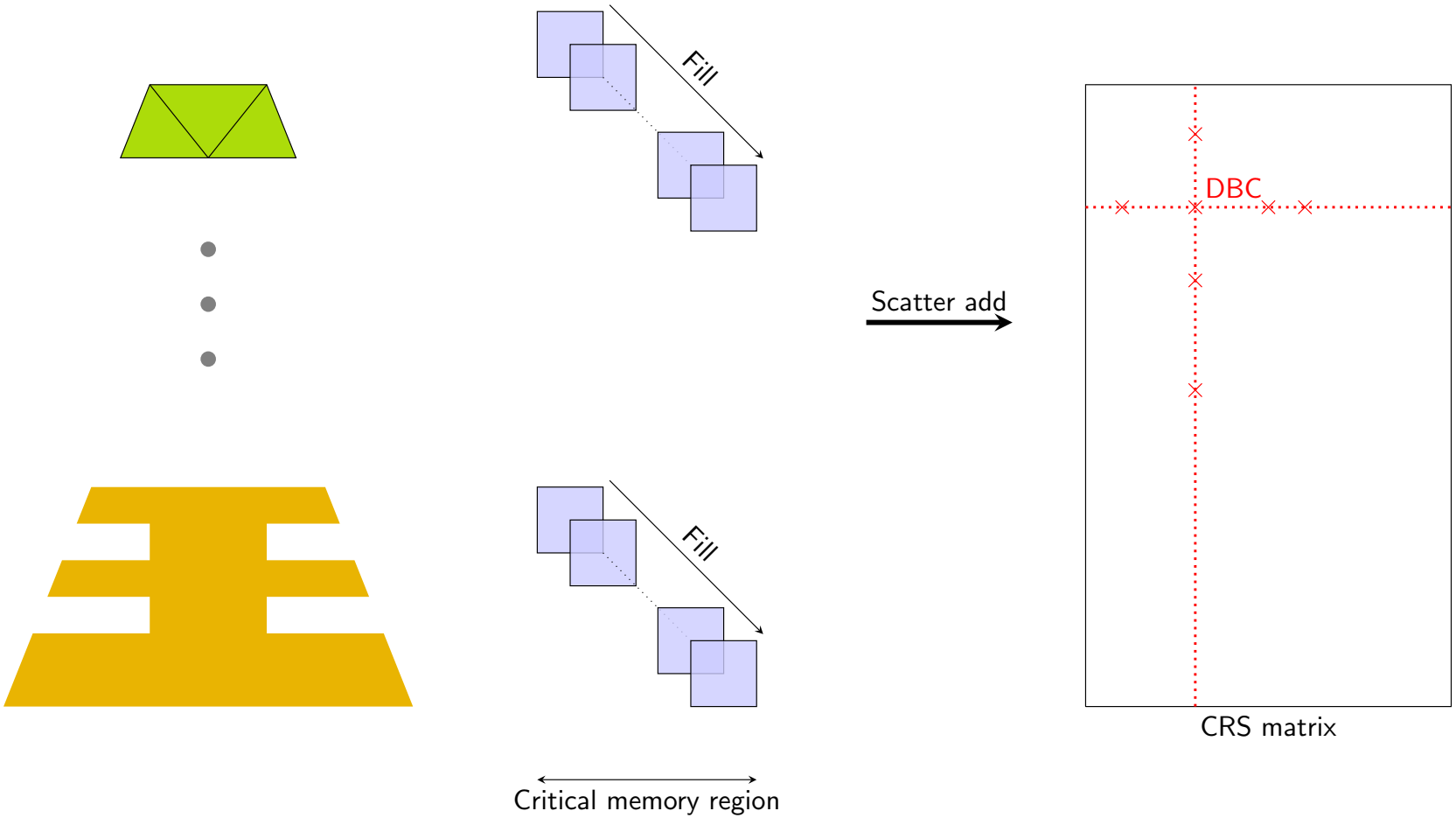
Outline

1. Motivation
2. FE assembly conceived as a graph
   - Decomposition into sub-domains
   - Organize FE assembly as a DAG
3. Performance-portable dispatch of workloads DAG
   - Performance portability with `Kokkos`
   - Asynchronicity and streams
   - Benchmarking `Kokkos::Graph`
4. FE boundary conditions DAG implemented as `Kokkos::Graph`
   - Back-of-the-envelope calculation
   - Electromagnetic scattering in 2D
5. Conclusion and outlook

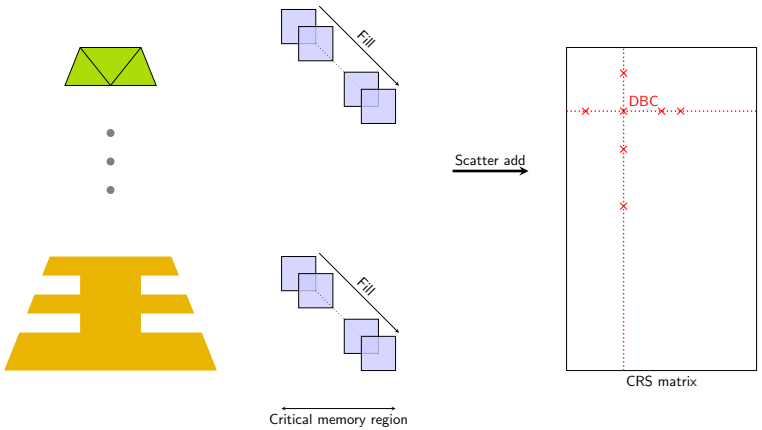
Outline

FE assembly: build up a global system from elements

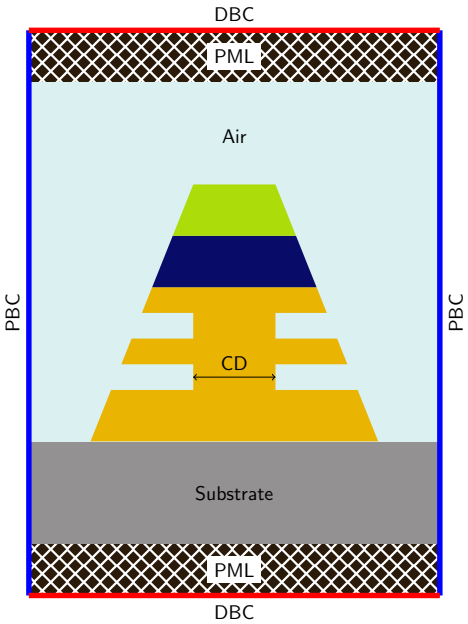
1. Decompose domain (different material properties, weak forms, ...)
2. Fill elemental matrices for every sub-domain
3. Scatter add elemental contributions into global DOFs (CRS) matrix
4. Apply boundary conditions (Dirichlet, ...)

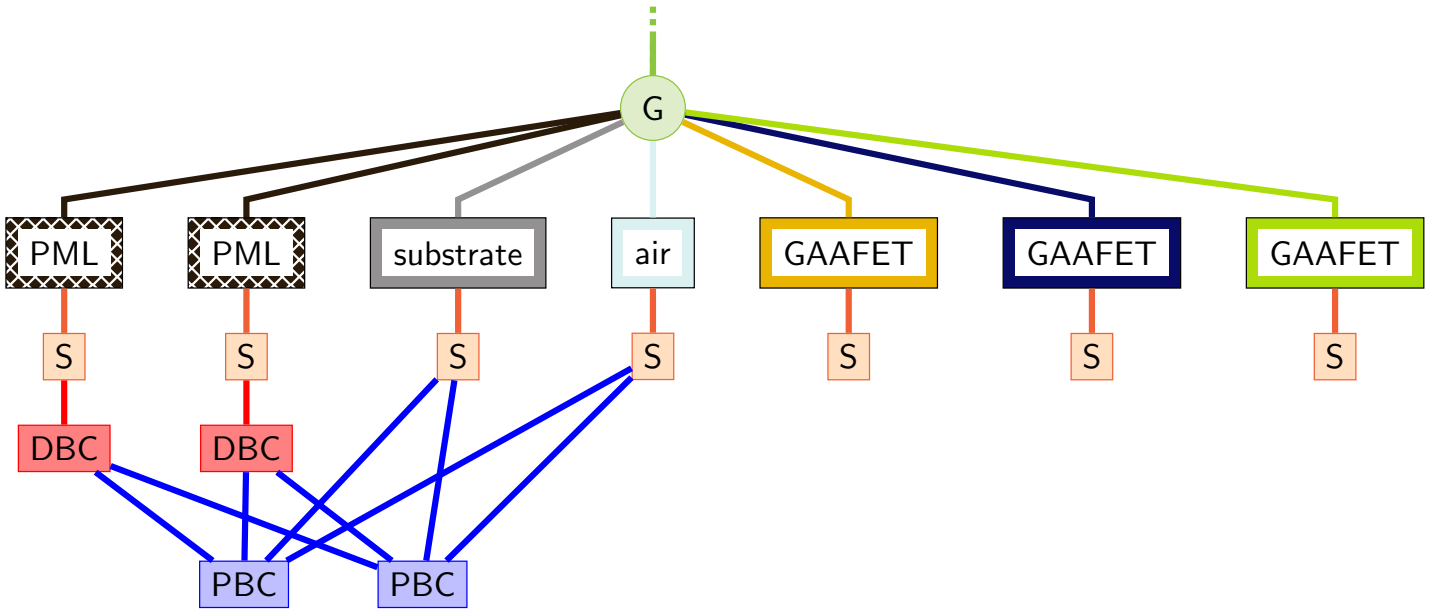


Organizing (in)dependent computations as a DAG graph



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

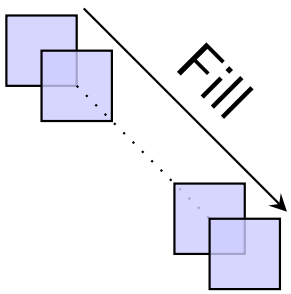


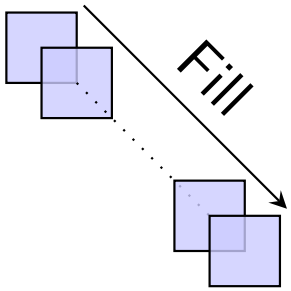


Outline

Performance portable GPU workloads

- A functor encapsulates both data and methods applied to it.
- A parallel region executes the body of a computational pattern following a given execution policy.
- A workload is thus defined by {pattern, execution policy, functor}.

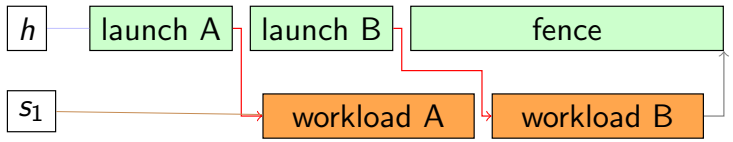





```
template <...>
struct FillFunctor
{
    ... data ...

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

template <typename execution_space, ...>
void execute_work(const execution_space& space, ... data ...)
{
    Kokkos::RangePolicy<execution_space> policy(space, 0, num_elems);

    FillFunctor<...> body(... data ...);

    Kokkos::parallel_for(policy, body);
}
```

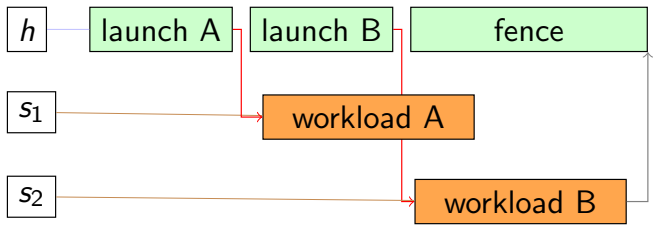
Managing asynchronicity in a DAG using streams

Sequential

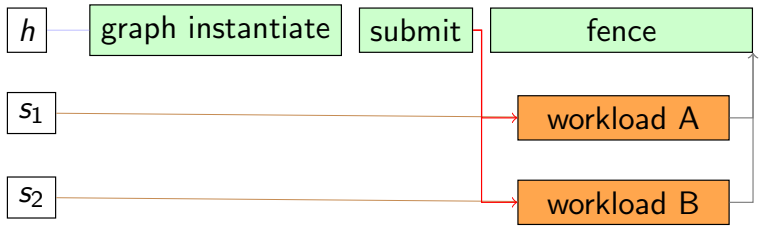


Stream-based

manual



graph



Stream (*a.k.a.* space instance)

- *queue* of workloads defined by {pattern, execution policy, functor}
- Several streams *may* run concurrently.
- Streams can be used to *expose more parallelism* to saturate GPUs.

Observation

Manual stream management incurs additional code clutter *w.r.t.* a graph.

Question 1

Efficiency of streams *vs.* sequential?

Question 2

Overhead of graph?

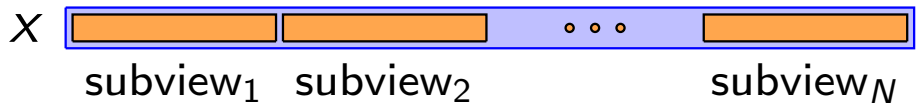
Benchmarking `Kokkos::Graph` (1/3)

Foretaste of `Kokkos::Graph`

- Portable wrapper around `cudaGraph_t` or `hipGraph_t`.
- Default *sequential* implementation for "unsupported" backends.

Iterated AXPBY distributed to nodes:

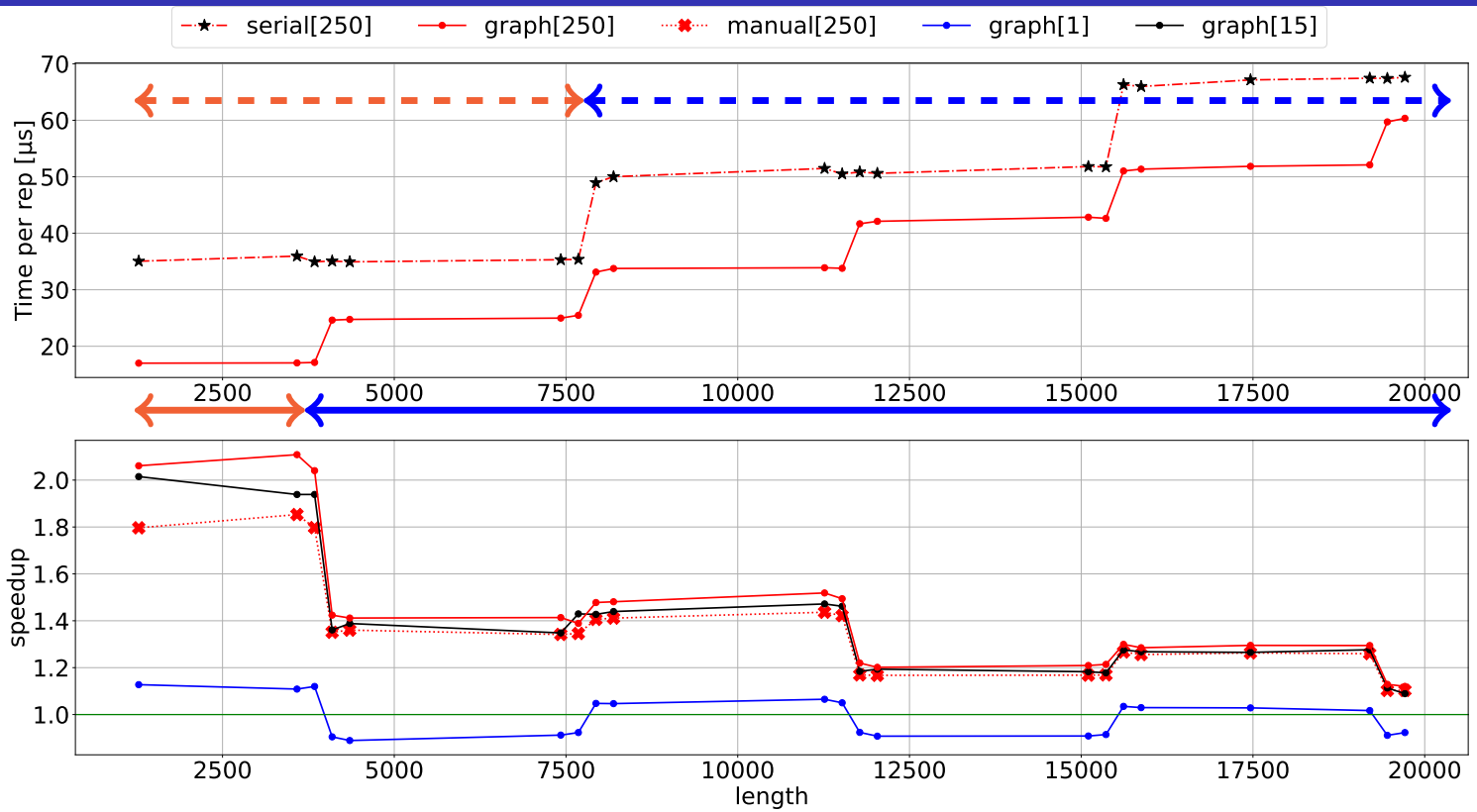
$$\big((\mathbf{x} \leftarrow\!\shortmid\, \alpha\mathbf{x} + \beta\mathbf{y}), \dots\big) \times 100$$



Questions

- Efficiency of streams *vs.* sequential?
  - Interest of concurrency for maximizing occupancy.
- Overhead of graph?
  - Cost of graph creation, instantiation, submission and destruction under complex topologies.

Benchmarking Kokkos::Graph (2/3)



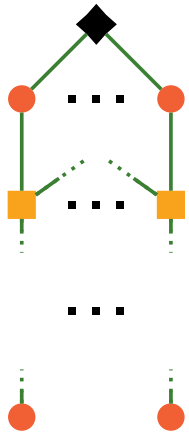
Generated on AMPERE 86, using CUDA 12.2.2 (2 graph nodes; 30 SMs with 128 block size).

- Under-utilization: Partitioning of AXPBY yields fewer blocks than the number of available compute units.
- Towards saturation: More than one block per compute unit improves performance by hiding memory latency.

Assigning asynchronous workloads to streams is always beneficial!

Benchmarking Kokkos::Graph (3/3)

	1x2	2x3	3x4	4x5	5x6
cudaGraphCreate (?)	1.9	2.6	3.7	6	11.1
cudaGraphAddDependencies	1.8	1.7	1.6	1.6	1.6
cudaGraphAddKernelNode	3.9	3.7	3.7	3.8	3.8
cudaGraphAddEmptyNode	3.8	3.6	3.6	3.6	3.6
cudaGraphInstantiate	26.8	70.6	155.1	320.9	589
cudaGraphLaunch	8.4	15.6	24.5	36.3	52.8
cudaGraphExecDestroy	7.4	12.6	20.9	34.9	58.1
cudaGraphDestroy	2.3	3	4.5	6.4	9.3



Generated with Nsight Systems on Ampere 86, using Cuda 12.2.2. Time in microseconds.

- Graph creation: Adding a node or an edge has a constant cost.
- Graph instantiation: Cost increases with graph complexity. It can be amortized through re-issue.
- Graph launch: Cost grows with number of nodes. Potential launch overhead reduction.

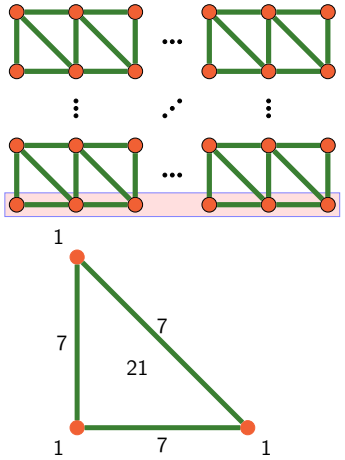
Outline

1. Motivation
2. FE assembly conceived as a graph
   - Decomposition into sub-domains
   - Organize FE assembly as a DAG
3. Performance-portable dispatch of workloads DAG
   - Performance portability with `Kokkos`
   - Asynchronicity and streams
   - Benchmarking `Kokkos::Graph`
4. FE boundary conditions DAG implemented as `Kokkos::Graph`
   - Back-of-the-envelope calculation
   - Electromagnetic scattering in 2D
5. Conclusion and outlook

Is it worth bothering with asynchronicity for BCs?

Scenario

- LUMI-G, *AMD MI250X* with 64*GB* (1 die)
- 2D Laplacian on a rectangle (N by $N/2$ elements) (*fp64*)
- Mesh with *TRI3* elements and *HGRAD* basis of degree 8 (45 dofs, 55 cubature points)



`Intrepid2`

QDR	ϕ	$\nabla\phi$	mat.	$\mathcal{M}^{\text{stacked}}_{\text{elem}}$ [bytes]
$55 \cdot 2$	$55 \cdot 45$	$55 \cdot 45 \cdot 2$	45^2	76 480

`Tpetra`

row offsets	column indices	values	$\mathcal{M}^{\text{CRS}}_{\text{dof}}$ [bytes]
1	$\mathcal{O}(6 \cdot 45)$	$\mathcal{O}(6 \cdot 45)$	541

nodes	edges	faces	dofs
$(N+1)\left(\frac{N}{2}+1\right)$	$\frac{3}{2}N(N+1)$	N^2	$32N^2 + 12N + 1$

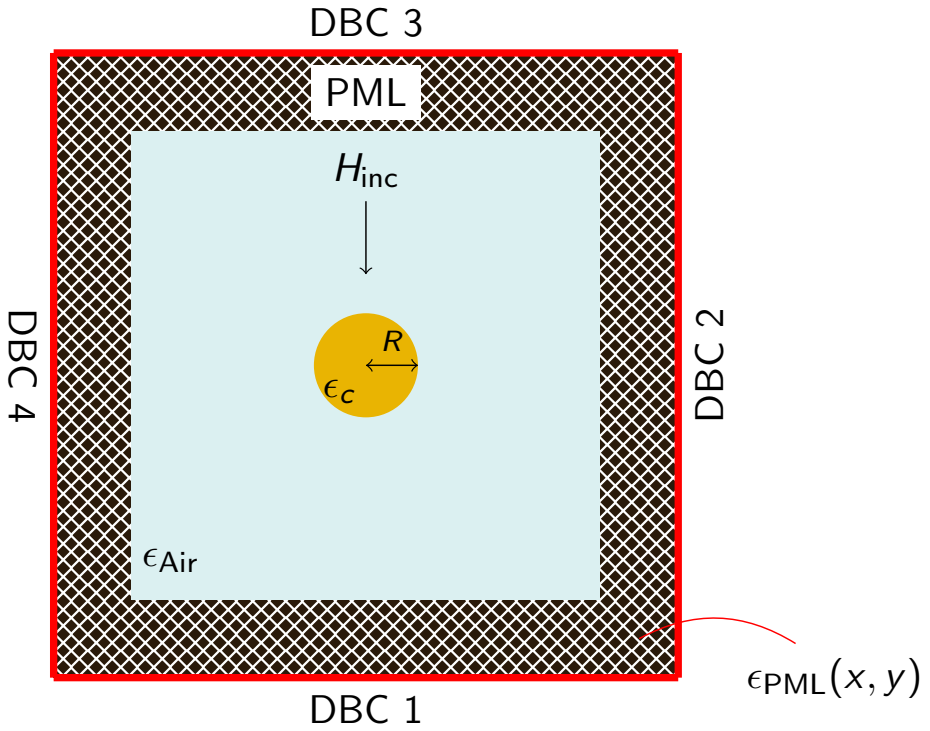
- Allowable N due to global memory constraint: 856.
- This would allow for $\frac{8 \times 856 + 1}{1024} = 6.69$ CUs to be occupied by a single functor working on one side, from the 110 CUs available.

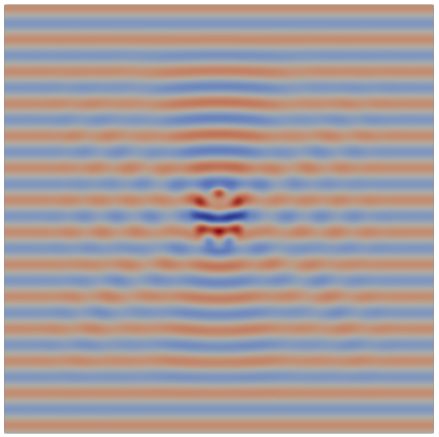
Expose BCs as asynchronous workloads to increase GPU utilization.

Electromagnetic scattering in 2D (1/2)

Mie scattering

Scatter of H_z-polarized plane wave H_{inc} over a dielectric cylinder of permittivity ϵ_c.





$H^{\text{total}}_{\text{real}}$

$H^{\text{scattered}}_{\text{real}}$

Focus

Currently DAG encompasses only boundary conditions.

Electromagnetic wave scattering in 2D (2/2)

Mesh of 43 706 *TRI3* elements with *HGRAD* basis of degree 5 (21 dofs, 25 cubature points), `std::complex<double>`.

API call	Cost/call [µs]	# [-]
`cudaGraphCreate`	4.6	1
`cudaGraphAddDependencies`	2.6	4
`cudaGraphAddKernelNode`	7.6	4
`cudaGraphAddEmptyNode`	12.3	1
`cudaGraphInstantiate`	116.3	1
`cudaGraphLaunch`	28.1	1
`cudaGraphExecDestroy`	31.2	1
`cudaGraphDestroy`	9.1	1

Kernel	Launch grid	Time [µs]
Fill	$(342, 1, 1) \times (1, 128, 1)$	$5 \cdot 10^5$
DBC	$(4, 1, 1) \times (1, 128, 1)$	$5 \cdot 10^2$

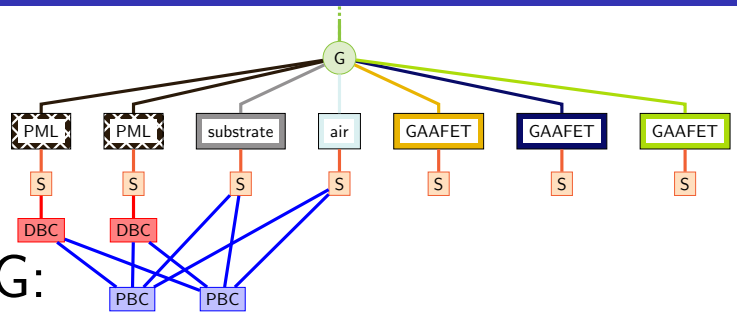
Generated using Nsight Systems and Nsight Compute on AMPERE 86, using CUDA 12.2.2.

- Graph overhead (total of ~ 240 µs):
  - is negligible *w.r.t. Fill*.
  - is acceptable *w.r.t.* DBC, given that it allows launching all 4 Dirichlet workloads concurrently, and one workload runs in $\sim 5 \cdot 10^2$ µs.
- The compute/memory intensity per dof for Dirichlet is quite low. The gain might become more significant for more complex BCs.

Outline

1. Motivation
2. FE assembly conceived as a graph
   - Decomposition into sub-domains
   - Organize FE assembly as a DAG
3. Performance-portable dispatch of workloads DAG
   - Performance portability with `Kokkos`
   - Asynchronicity and streams
   - Benchmarking `Kokkos::Graph`
4. FE boundary conditions DAG implemented as `Kokkos::Graph`
   - Back-of-the-envelope calculation
   - Electromagnetic scattering in 2D
5. Conclusion and outlook

Conclusion



- Organize FE assembly workloads as a DAG:
  - Dependencies are semantically expressed.
  - Enable concurrent execution when resources are available.
- Efficient implementation of workloads DAG using `Kokkos::Graph`:
  - Assigning asynchronous workloads to streams improves performance.
  - Graph overhead is negligible for heavy-weight workloads and can be amortized with re-issue for light-weight workloads.
- Applied to a 2D wave scattering problem
  - For *Fill*, graph overhead is negligible.
  - *BCs* workloads generally under-utilize the GPU and hence benefit from asynchronous dispatch.

Take-home message

Asynchronous FE assembly workloads can be efficiently dispatched with a device DAG.

Future work

Future directions for research:

- Explore more complex boundary conditions.
- Include bulk computations in the graph (*Fill*, *Scatter*, ...).
- Dynamic update of kernel data within `Kokkos`.
- Explore performance on other GPU architectures.
- Handle multi-GPU with `Kokkos::Graph`.
- Set node priority (similar to *stream* priority).
- Add memory allocation/deallocation nodes.
- Add host nodes.

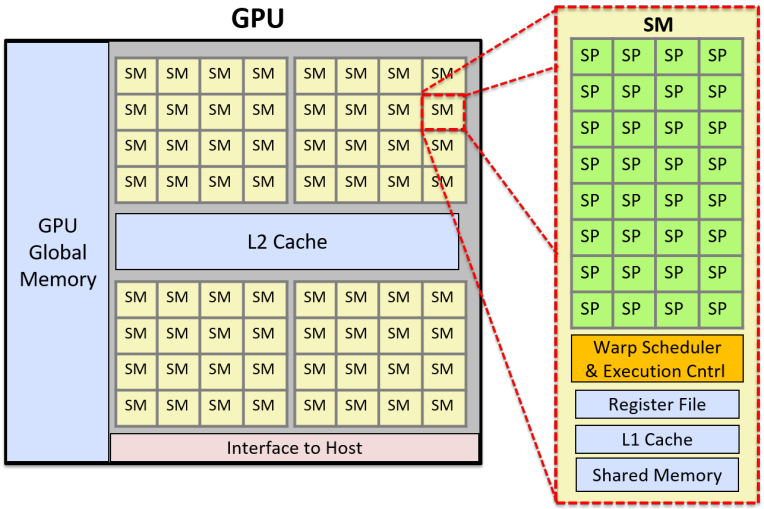
References

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.

Acknowledgment

The first author, Tomasetti Romin, would like to acknowledge the Belgian National Fund for Scientific Research (FNRS) for its financial support.

Architecture of a CUDA GPU

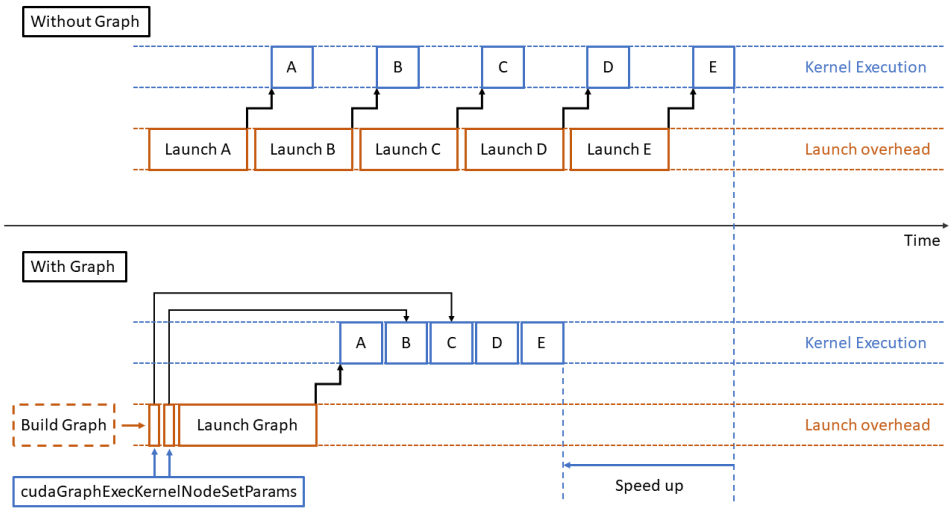


(a) Global view

(b) Zoom on *Hopper* Streaming Multiprocessor (SM)

CUDA graph in details

- `cudaGraph_t` was introduced in CUDA 10.
- A graph groups a set of kernels and other CUDA operations (memory and so on) together.
- Managing a DAG using `cudaGraph_t` can speed up a workflow by combining the driver activities associates with kernel launches and CUDA API calls.
- It enforces dependencies with hardware accelerations, instead of relying solely on CUDA streams and events, when possible.



`https://developer.nvidia.com/blog/constructing-cuda-graphs-with-dynamic-parameters/`