Virtual dispatch on GPUs

Design Patterns and Performance Analysis of Polymorphism in Multiphysics FE Assembly on GPU

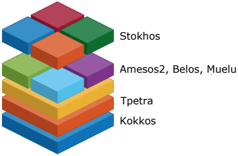
Arnst Maarten   Tomasetti Romin

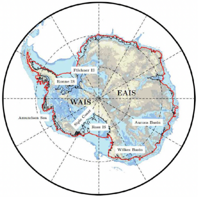
University of Liège, Belgium

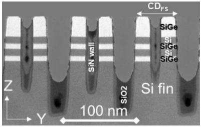
March 7th, 2024

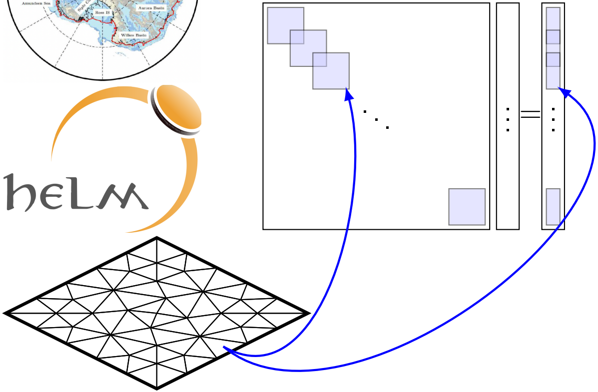
Conference on Parallel Processing for Scientific Computing

Motivation









Polymorphism in fine-grained parallel FE assembly for multidomain and multiphysics simulation.

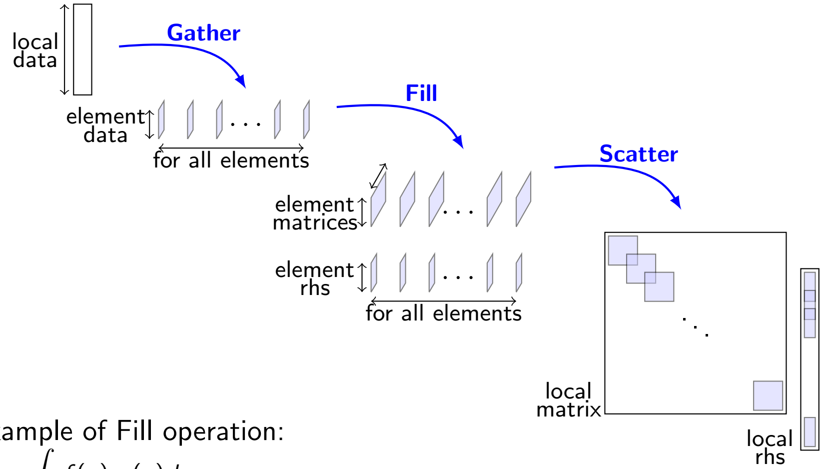
Outline

1. Fine-grained parallel FE assembly
    - Gather-fill-scatter approach
    - Polymorphism patterns
2. Virtual functions on device
    - Implementation aspects
    - Performance evaluation
3. Application
4. Conclusion and outlook

Outline

1. Fine-grained parallel FE assembly
    - Gather-fill-scatter approach
    - Polymorphism patterns
2. Virtual functions on device
    - Implementation aspects
    - Performance evaluation
3. Application
4. Conclusion and outlook

Gather-fill-scatter approach



Example of Fill operation:

$$f_i^K = \int_K f(\boldsymbol{x})\varphi_i(\boldsymbol{x})d\boldsymbol{x}$$

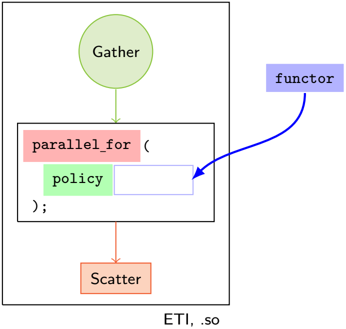
$$\approx \sum_q f\left(\boldsymbol{g}_K(\hat{\boldsymbol{x}}_q)\right)\hat{\varphi}_i(\hat{\boldsymbol{x}}_q)\mathrm{det}\boldsymbol{J}_K(\hat{\boldsymbol{x}}_q)w_q.$$

Polymorphism patterns I

- Kokkos parallel region: parallel pattern, policy, functor:

```
parallel_for (
    RangePolicy(numWorkItems) , functor
);
```

- Pattern 1: polymorphic functor (device virtual calls):



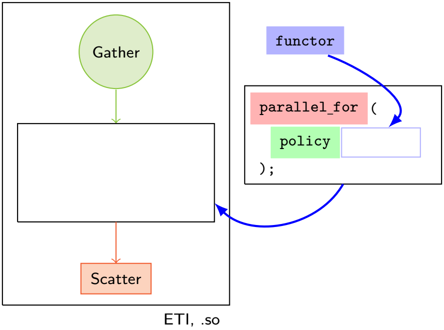

```
struct FunctorInterface
{

    KOKKOS_FUNCTION
    virtual void operator()(
        const ordinal_t elemID
    ) const = 0;
};
```

```
struct FunctorImplementation
  : public FunctorInterface
{
    KOKKOS_FUNCTION
    void operator()(
        const ordinal_t elemID
    ) const override { ... };
};
```

Polymorphism patterns II

- Pattern 2: polymorphic driver (host virtual call):




```
struct NodeBase
{
    virtual void execute() const = 0;
};

template <
    typename Policy ,
    typename FunctorImplementation
>
struct Node
  : public NodeBase
{
    void execute() const override {
    parallel_for (
        policy , functor
    ); }

    Policy policy;
    FunctorImplementation functor ;
};
```

Outline

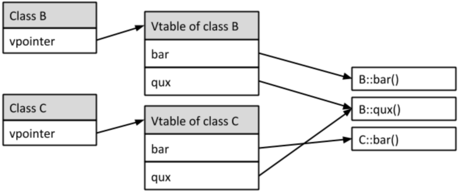
1. Fine-grained parallel FE assembly
    - Gather-fill-scatter approach
    - Polymorphism patterns

2. Virtual functions on device
    - Implementation aspects
    - Performance evaluation

3. Application

4. Conclusion and outlook

How do virtual functions work?



- Virtual function calls work by using a layer of indirection. A call through an interface pointer results in a lookup in a virtual function table (vTable) to determine the implementation function to run.
- For virtual function calls to work on device, the vTable must be set up properly, i.e., point to device code. Achieved by constructing derived class instances on device.

[Ari17] *P. Arias. Understanding Virtual Tables in C++. 2017.*
[BCH+19] *V. Brunini et al. Runtime polymorphism in Kokkos applications. Sandia, 2019.*

Virtual functions on device

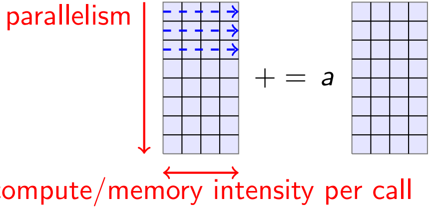
Generated using Nsight Compute
on Nvidia V100 using Cuda 12.2.2.

- Direct overhead of virtual function calls on device:
  - Loads from memory for vTable lookups.
  - Indirect function call.
- Indirect effects of virtual function calls on device:
  - Loss of opportunities for optimization by the compiler.

[ZAR21] *M. Zhang et al. Characterizing Massively Parallel Polymorphism. IEEE, 2021.*

Performance of virtual functions on device

$$[Y] += \alpha[X]$$
$$[X], [Y] \in \mathrm{M}_{m\times n}(\mathbb{R})$$
$$m = 2^{17} = 131\,072,$$
$$n = 16, 32, 64, \dots, 4\,096.$$



size of vTable

```
template <int Idx>
struct Implementation<Idx>
  : public Interface
KOKKOS_FUNCTION
void operator()(
    const int i,
    ...
) const override {
    for (int j = 0; j < n; ++j)
        y(i,j)+=a*x(i,j);
}
```

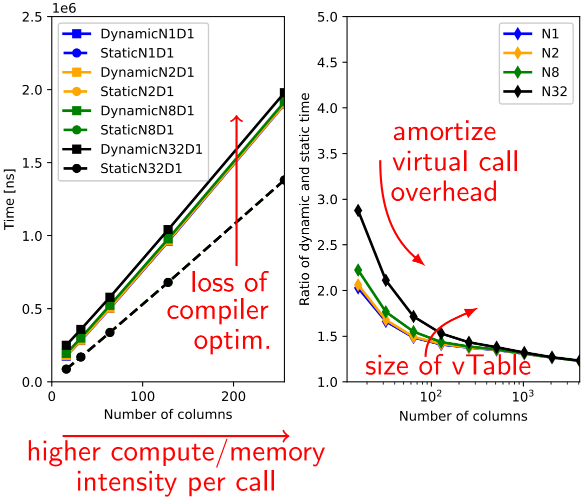
```
parallel_for(
    RangePolicy<ExeSpace>(0,m),
    KOKKOS_LAMBDA (const int i) {
        objects_dptr(i % divergence_level)
            ->operator()(i, ...);
    }
);

View<
    PointerWrapper<Interface>*, ExeSpace
> objects_dptr;
```

Implemented in Kokkos and adapted from: [ZAR21] *M. Zhang et al. Characterizing Massively Parallel Polymorphism. IEEE, 2021.*

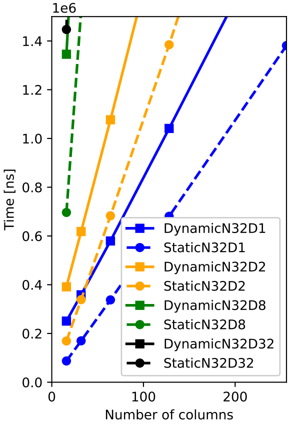
Performance of virtual functions on device

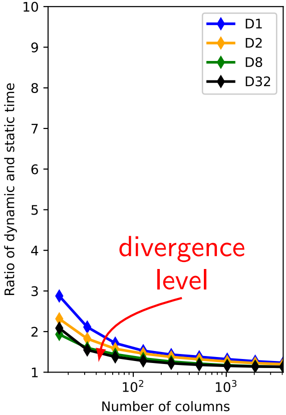
Generated on Nvidia V100 using Cuda 12.2.2 and `Kokkos` 4.3.
Number of derived classes: $N = 1, 2, 8, 32$ (increasing size of vTable).
Divergence level: $D = 1$ (no warp-level divergence).



Performance of virtual functions on device

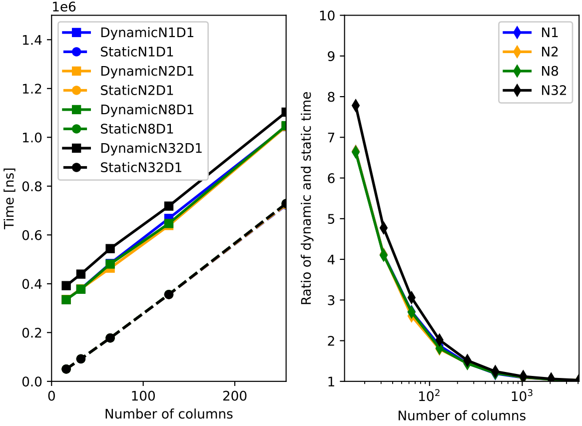
Generated on Nvidia V100 using Cuda 12.2.2 and `Kokkos` 4.3.
Number of derived classes: $N = 32$ (fixed size of vTable).
Divergence level: $D = 1, 2, 8, 32$ (increasing warp-level divergence).





Performance of virtual functions on device

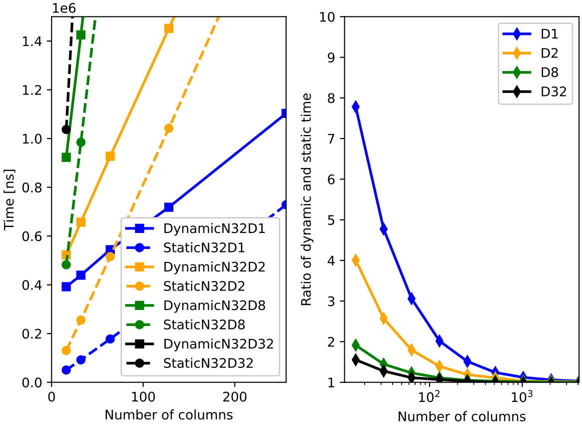
Generated on AMD MI250X using ROCm 6.0.1 and `Kokkos` 4.3. Blocksize 256.
Number of derived classes: $N = 1, 2, 8, 32$ (increasing size of vTable).
Divergence level: $D = 1$ (no warp-level divergence).



More significant overhead per virtual call on device on AMD MI250X than on Nvidia V100.
No clear impact of differences in compiler optimization.

Performance of virtual functions on device

Generated on AMD MI250X using ROCm 6.0.1 and `Kokkos` 4.3. Blocksize 256.
Number of derived classes: $N = 32$ (fixed size of vTable).
Divergence level: $D = 1, 2, 8, 32$ (increasing warp-level divergence).



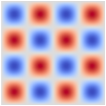
Outline

1. Fine-grained parallel FE assembly
   - Gather-fill-scatter approach
   - Polymorphism patterns
2. Virtual functions on device
   - Implementation aspects
   - Performance evaluation
3. Application
4. Conclusion and outlook

$$\begin{cases} -\triangle_{\boldsymbol{x}}\Phi = f & \text{in } \Omega, \\ \Phi = 0 & \text{on } \partial\Omega. \end{cases}$$

$f = -2k^2\pi^2 \sin(k\pi x_1)\sin(k\pi x_2)$

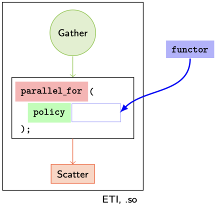
100×100 TRI3 elements

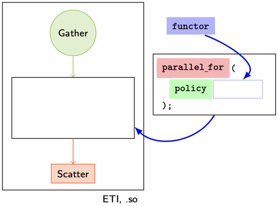


$$A_{ij}^K = \int_K \boldsymbol{\nabla}_x \varphi_i \cdot \boldsymbol{\nabla}_x \varphi_j d\boldsymbol{x} \approx \sum_q \boldsymbol{J}_K^{-\mathrm{T}}(\hat{\boldsymbol{x}}_q)\boldsymbol{\nabla}_{\hat{x}}\hat{\varphi}_i(\hat{\boldsymbol{x}}_q) \cdot \boldsymbol{J}_K^{-\mathrm{T}}(\hat{\boldsymbol{x}}_q)\boldsymbol{\nabla}_{\hat{x}}\hat{\varphi}_j(\hat{\boldsymbol{x}}_q)\det\boldsymbol{J}_K(\hat{\boldsymbol{x}}_q)w_q$$

Finite elements of degree p: $\hat{\varphi}_0, \hat{\varphi}_1, \ldots$ in $\mathbb{P}_2^p$ with $\mathrm{card}(\mathbb{P}_2^p) = \dfrac{(p+1)(p+2)}{2}$.

compute/memory intensity per call



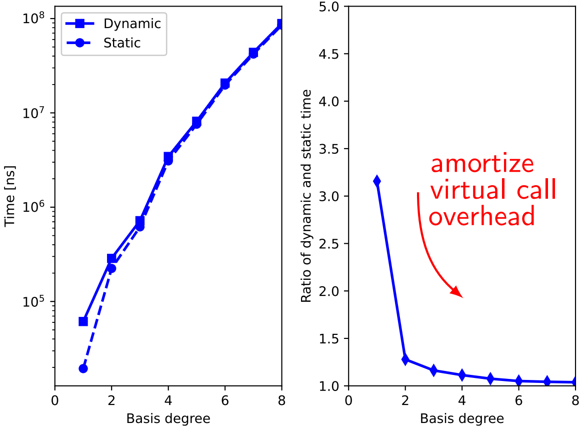


Application

Generated on Nvidia V100 using Cuda 12.2.2 and `Kokkos` 4.3.

"Dynamic": Polymorphic functor pattern with virtual calls on device.

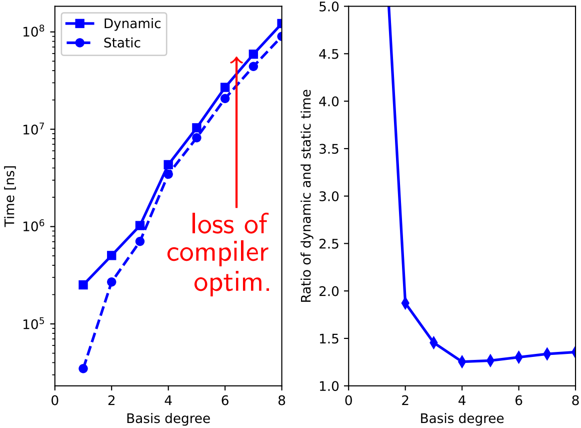
"Static": Polymorphic driver pattern with static calls on device.



Virtual calls on device entail significant overhead for low-order FE.

Application

Generated on AMD MI250X using ROCm 6.0.1 and `Kokkos` 4.3. Blocksize 256.
"Dynamic": Polymorphic functor pattern with virtual calls on device.
"Static": Polymorphic driver pattern with static calls on device.



Clearer impact of differences in compiler optimization on AMD MI250X than on Nvidia V100.

Outline

1. Fine-grained parallel FE assembly
   - Gather-fill-scatter approach
   - Polymorphism patterns
2. Virtual functions on device
   - Implementation aspects
   - Performance evaluation
3. Application
4. Conclusion and outlook

Conclusion

- We compared two polymorphism patterns for fine-grained parallel FE assembly for multi-domain multi-physics simulation.
- *Polymorphic functor:* Polymorphism is expressed by using a base type for the functor. Virtual calls inside the parallel region on device.
- *Polymorphic driver:* Polymorphism is expressed by using a base type for the parallel region. The functor is wrapped into a larger type that keeps the pattern, policy and functor together. Virtual call on host.

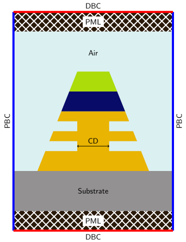
Take-home message

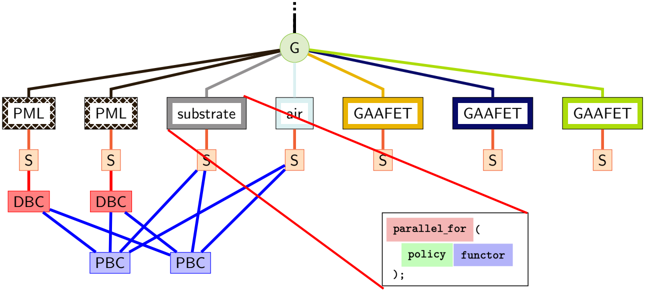
Prefer a *polymorphic driver* pattern over a *polymorphic functor* pattern:

- Avoid *vTable* lookup overhead on device. Significant for low-order FE.
- Keep compiler optimization opportunities open.

Directions for future work

- Explore mechanism of virtual dispatch on AMD GPUs.
- Explore effect of enabling relocatable device code.
- Performance evaluation in application in 3D.
- Graph-based assembly.





[TA24] *R. Tomasetti and M. Arnst. Efficiently implementing FE boundary conditions using stream-orchestrated execution on GPU. SIAM PP2024.*

References

P. Arias, *Understandig virtual tables in C++*, 2017.

V. Brunini, J. Clausen, M. Hoemmen, A. Kucala, C. Trott, and M. Howard, *Runtime polymorphism in Kokkos applications*, https://www.osti.gov/biblio/1592283, 2019, Presentation at the 2019 Exascale Computing Project Annual Meeting.

R. Tomasetti and M. Arnst, *Efficiently implementing FE boundary conditions using stream-orchestrated execution on GPU*, 2024, Presentation at the 2024 SIAM Conference on Parallel Processing.

M. Zhang, A. Alawneh, and T. Rogers, *Characterizing massively parallel polymorphism*, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2021.