14th International Workshop on OpenCL and SYCL

IWOCL 2026

A GPU Backend for the SYCL-Based 2D Stencil Framework StencilStream and a Comparison with its FPGA backends

Tim Stöhr, Paderborn University

Tim Stöhr, Jan-Oliver Opdenhövel, Christian Plessl, Tobias Kenter
Paderborn Center for Parallel Computing, Paderborn University

May 6-8, 2026 | Heilbronn, Germany | iwocl.org

Agenda

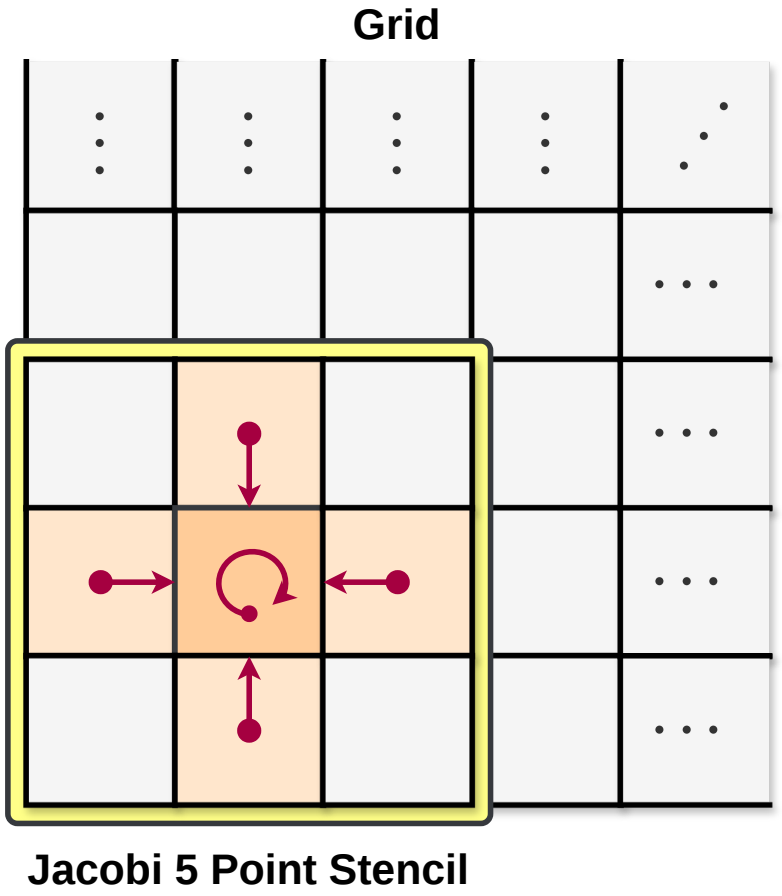
Paderborn Center for Parallel Computing

1. Stencil Computation and StencilStream
2. Memory access patterns
3. Transparent Scatter & Gather Design
4. Implementation details
5. Benchmarks

Stencil Computations (Background)

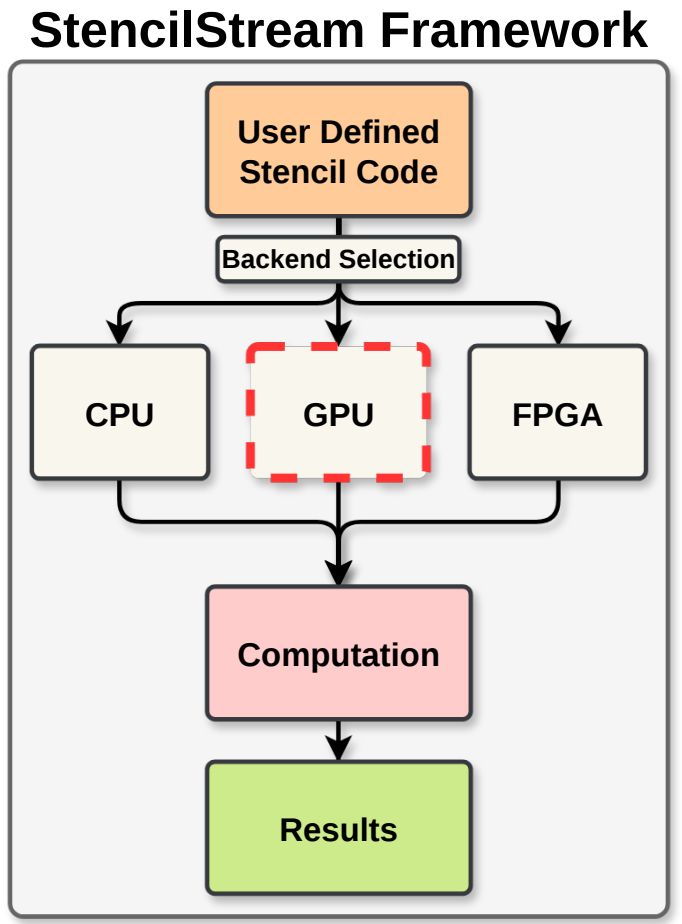
Paderborn Center for Parallel Computing

- Stencil codes update every cell in a grid based on its local neighborhood
- Used in
  - CFD
  - Thermodynamics
  - Image processing
- Simple pattern but a lot of work at scale
  - grid size * cell complexity * iterations
- Different hardware needs different optimization strategies



StencilStream Framework

- C++/SYCL-based framework:
  - User defines Cell, Transition Function, Grid, Backend
- Originally designed for FPGAs[1]
- Newly added GPU backend
- Improved FPGA backends with spatial parallelism
- Existing applications run on GPUs without modification

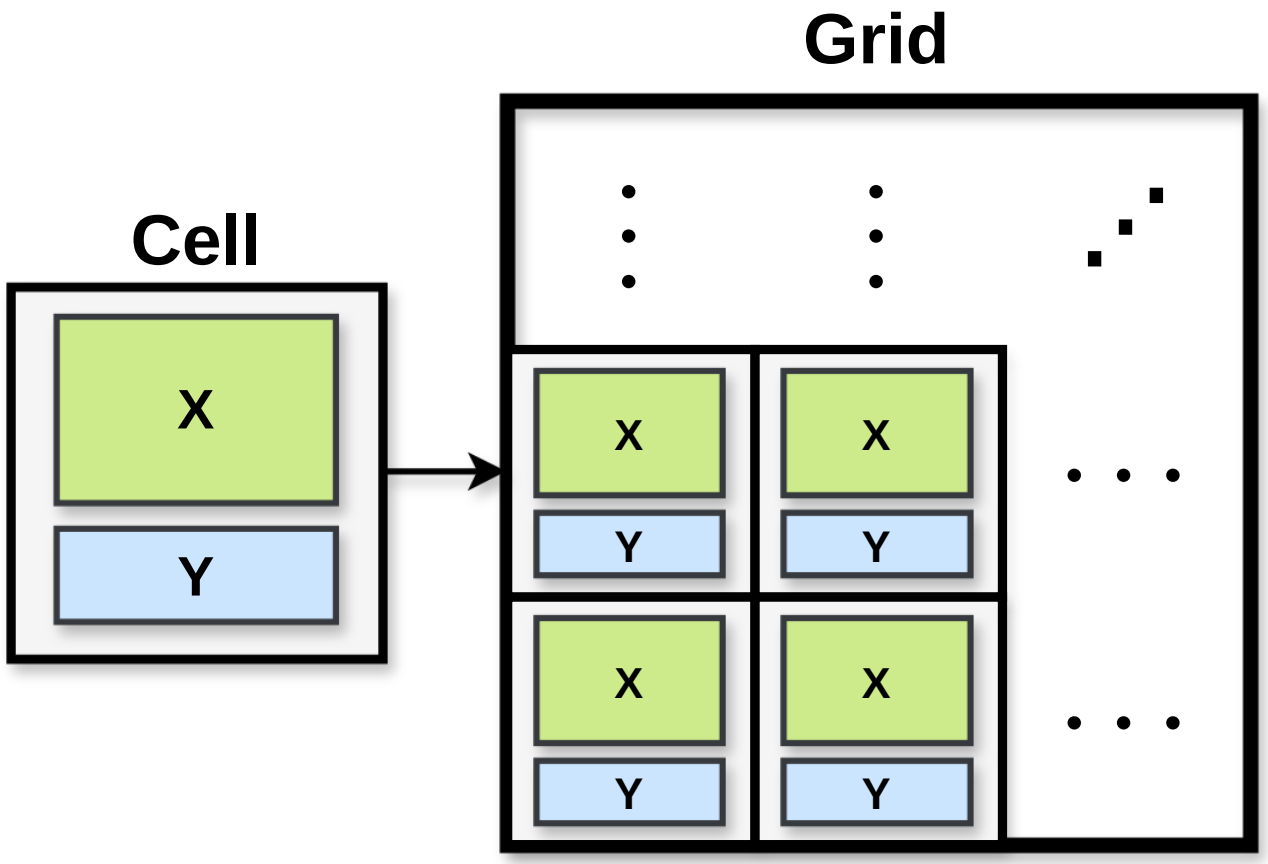


[1]A SYCL-based Stencil Simulation Framework Targeting FPGAs (DOI: 10.1109/FPL64840.2024.00023)

StencilStream User API

- User defines:
  - **Cell**

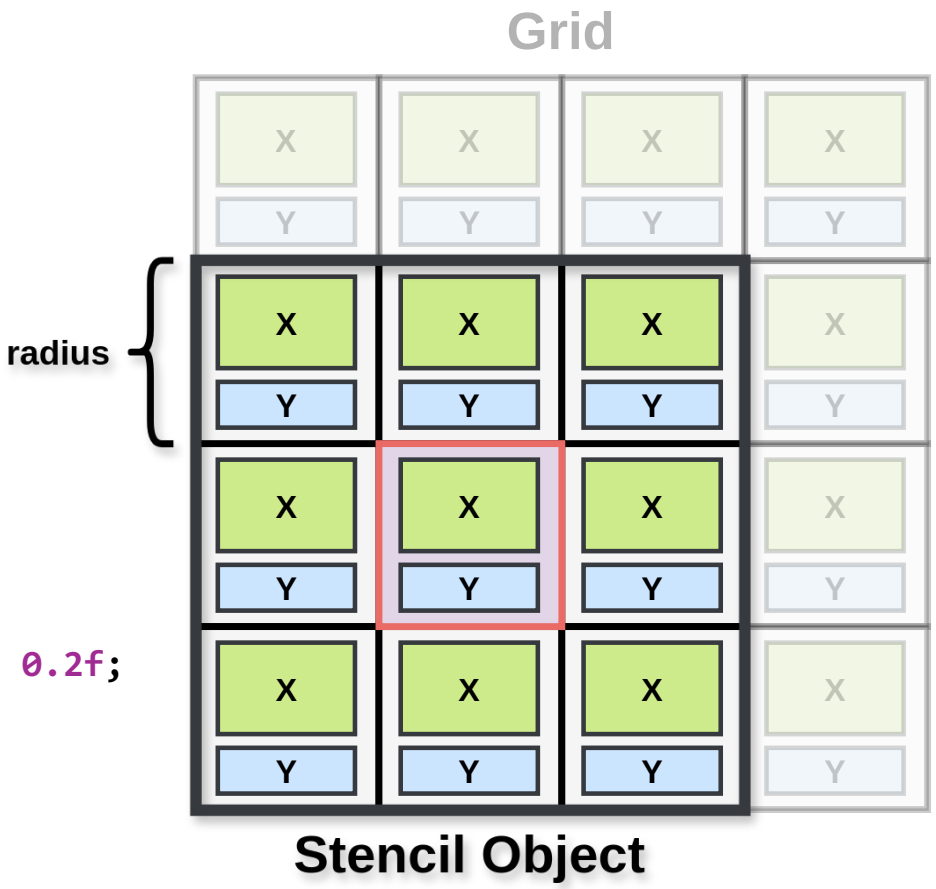
```
struct Cell {
  double X;
  float Y;
}
```



StencilStream User API

- User defines:
  - Cell
  - **Stencil (Transition Function)**

```
struct TransitionFunction {
    Cell operator()(Stencil<HotspotCell, 1> const &s) const {
        double center = s[0][0].X;
        double top    = s[-1][0].X;
        double bottom = s[1][0].X;
        double left   = s[0][-1].X;
        double right  = s[0][1].X;


        double new_X = (top + bottom + left + right + center) * 0.2f;


        return {new_X, s[0][0].Y};
    }
};
```

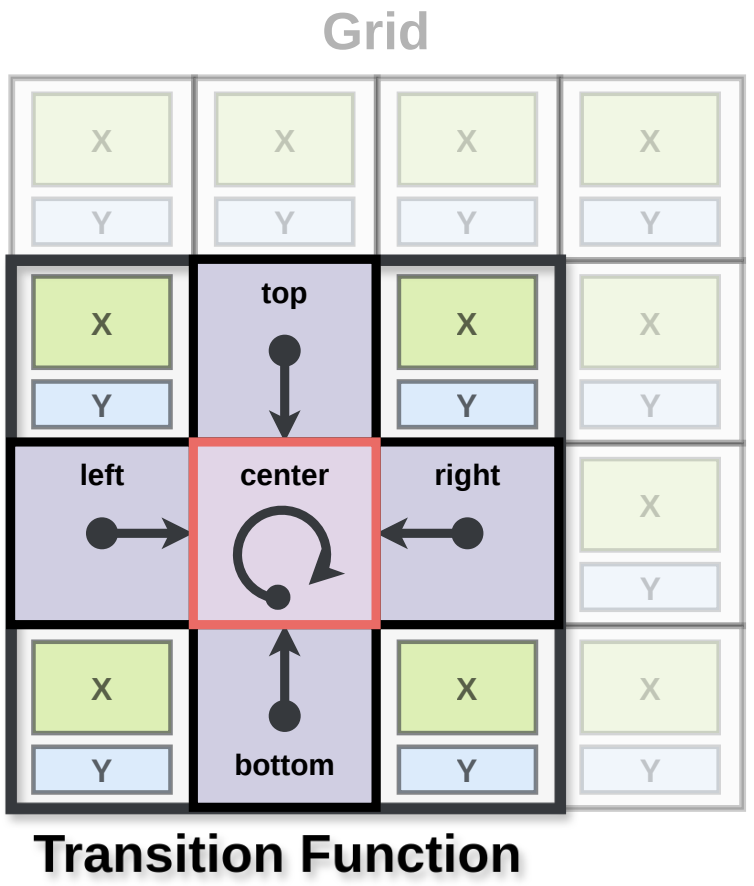


StencilStream User API

Paderborn Center for Parallel Computing

- User defines:
  - Cell
  - **Stencil (Transition Function)**

```
struct TransitionFunction {
    Cell operator()(Stencil<HotspotCell, 1> const &s) const {
        double center = s[0][0].X;
        double top    = s[-1][0].X;
        double bottom = s[1][0].X;
        double left   = s[0][-1].X;
        double right  = s[0][1].X;


        double new_X = (top + bottom + left + right + center) * 0.2f;


        return {new_X, s[0][0].Y};
    }
};
```

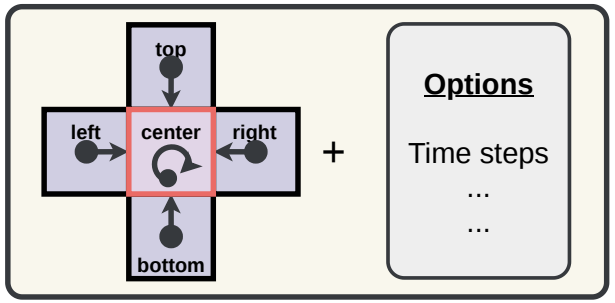


StencilStream User API

Paderborn Center for Parallel Computing

- User defines:
  - Cell
  - Stencil (Transition Function)
  - **Select backend and run**

```
using StencilUpdate = cuda::StencilUpdate<TransitionFunction>;
using Grid = StencilUpdate::GridImpl;

Grid grid = read_input(...);

StencilUpdate update({
    .transition_function = TransitionFunction{...},
    .n_iterations = sim_time
});

grid = update(grid);
```

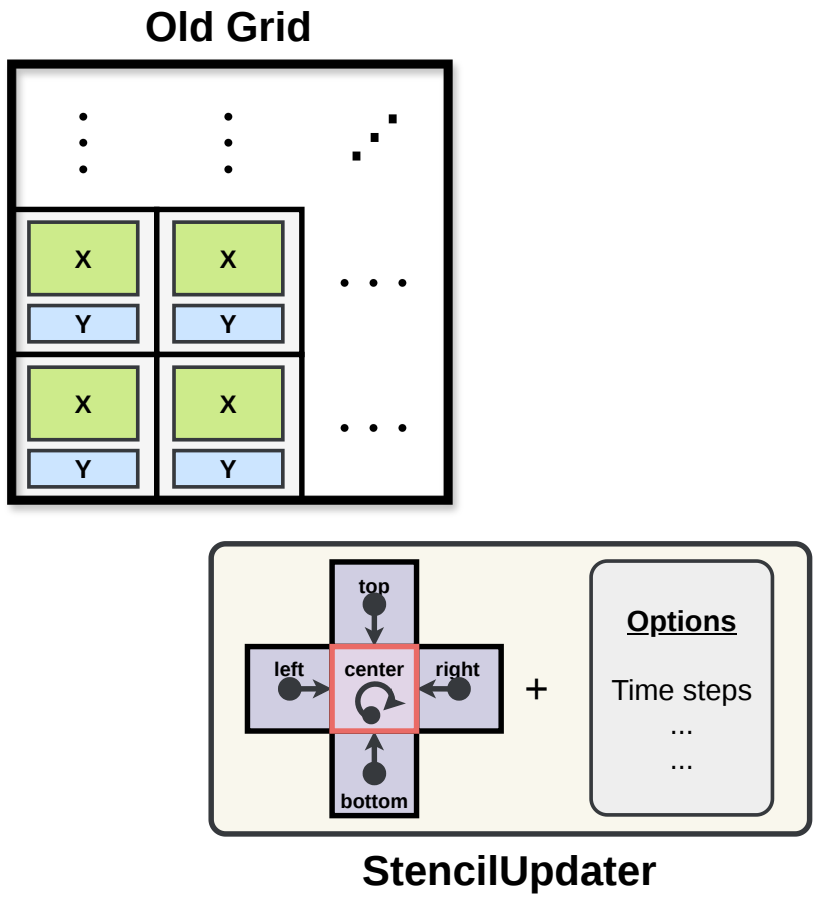


StencilUpdater

StencilStream User API

- User defines:
  - Cell
  - Stencil (Transition Function)
  - **Select backend and run**

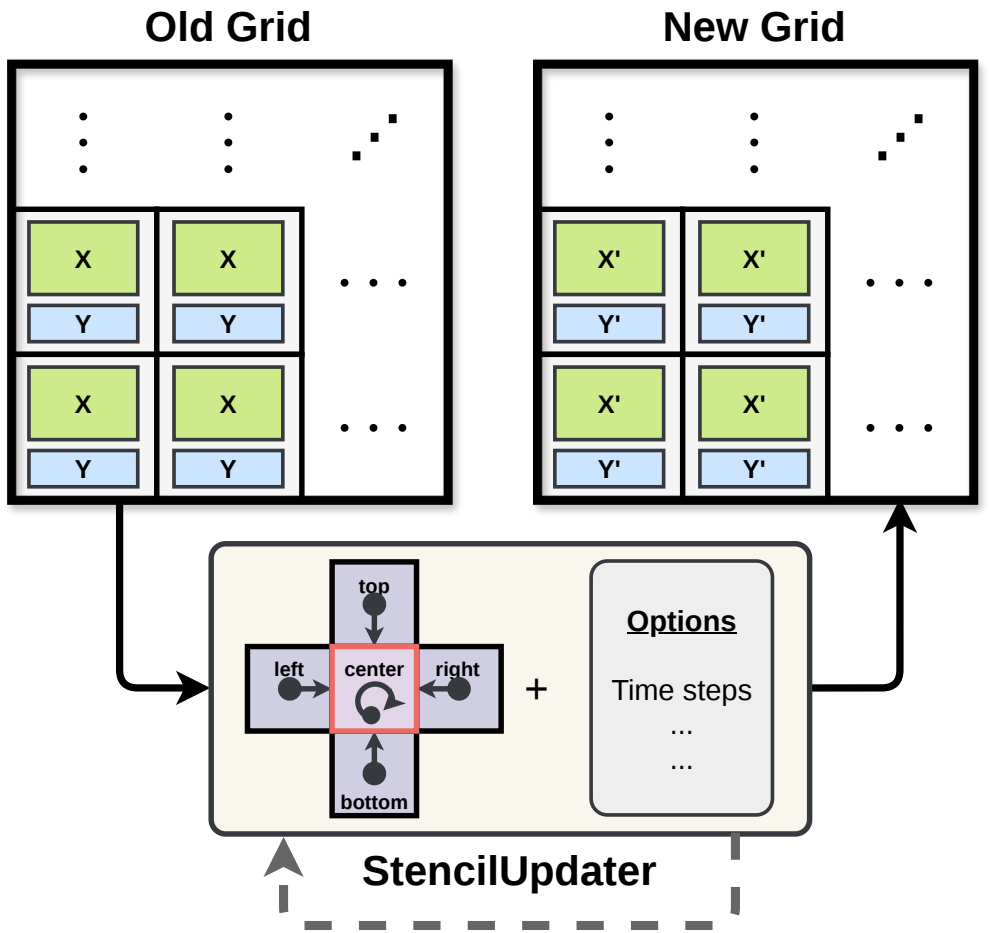
```
using StencilUpdate = cuda::StencilUpdate<TransitionFunction>;
using Grid = StencilUpdate::GridImpl;

Grid grid = read_input(...);

StencilUpdate update({
    .transition_function = TransitionFunction{...},
    .n_iterations = sim_time
});

grid = update(grid);
```



StencilStream User API

- User defines:
  - Cell
  - Stencil (Transition Function)
  - **Select backend and run**

```
using StencilUpdate = cuda::StencilUpdate<TransitionFunction>;
using Grid = StencilUpdate::GridImpl;

Grid grid = read_input(...);

StencilUpdate update({
    .transition_function = TransitionFunction{...},
    .n_iterations = sim_time
});

grid = update(grid);
```



Agenda

Paderborn Center for Parallel Computing

1. Stencil Computation and StencilStream
2. Memory access patterns
3. Transparent Scatter & Gather Design
4. Implementation details
5. Benchmarks

Strided Memory Access

Paderborn Center for Parallel Computing

- Initial GPU benchmarks showed poor memory throughput
- Profiling revealed: Strided memory access due to Array of Structs (AoS) memory layout

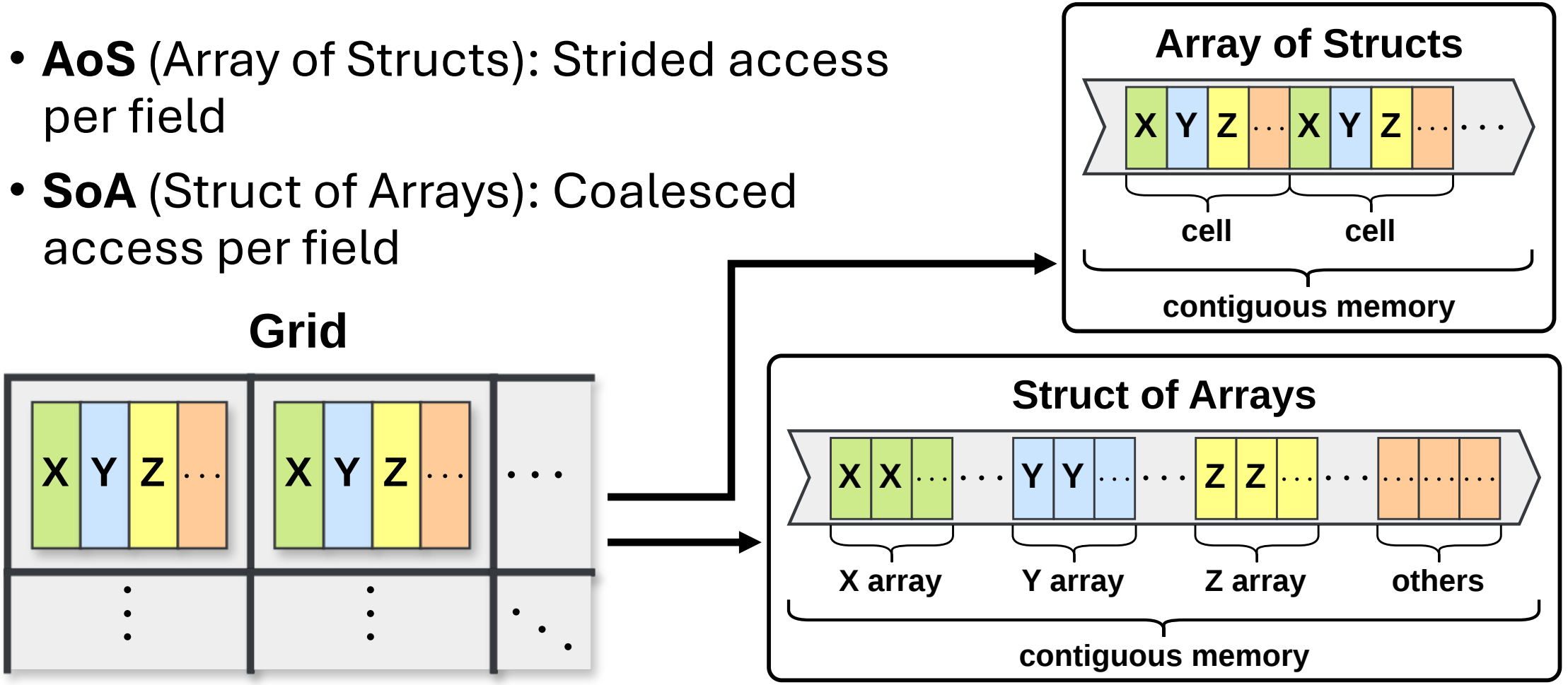
Uncoalesced Global Accesses
Est. Speedup: 73.84%

This kernel has uncoalesced global accesses resulting in a total of 37464528 excessive sectors (75% of the total 49958303 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide has additional information on reducing uncoalesced device memory accesses

Array-of-Structs vs. Struct-of-Arrays

Paderborn Center for Parallel Computing

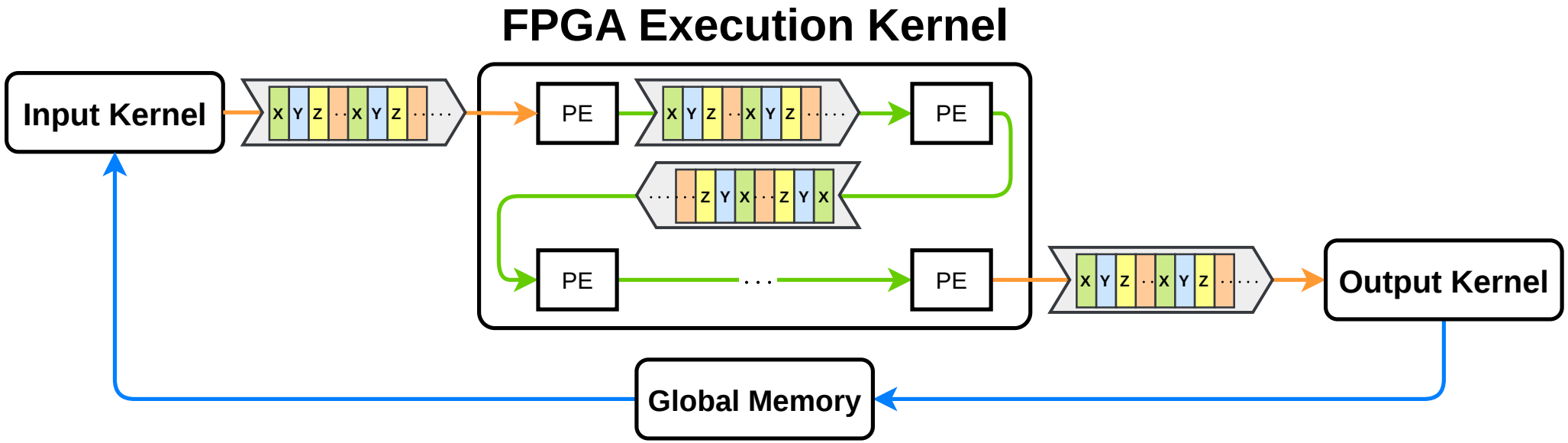
- **AoS** (Array of Structs): Strided access per field
- **SoA** (Struct of Arrays): Coalesced access per field



The Memory Layout Problem

Paderborn Center for Parallel Computing

- GPU works best with Struct-of-Arrays layout
- FPGA works best with Array-of-Structs layout



- **Changing the API would break existing FPGA apps**

Agenda

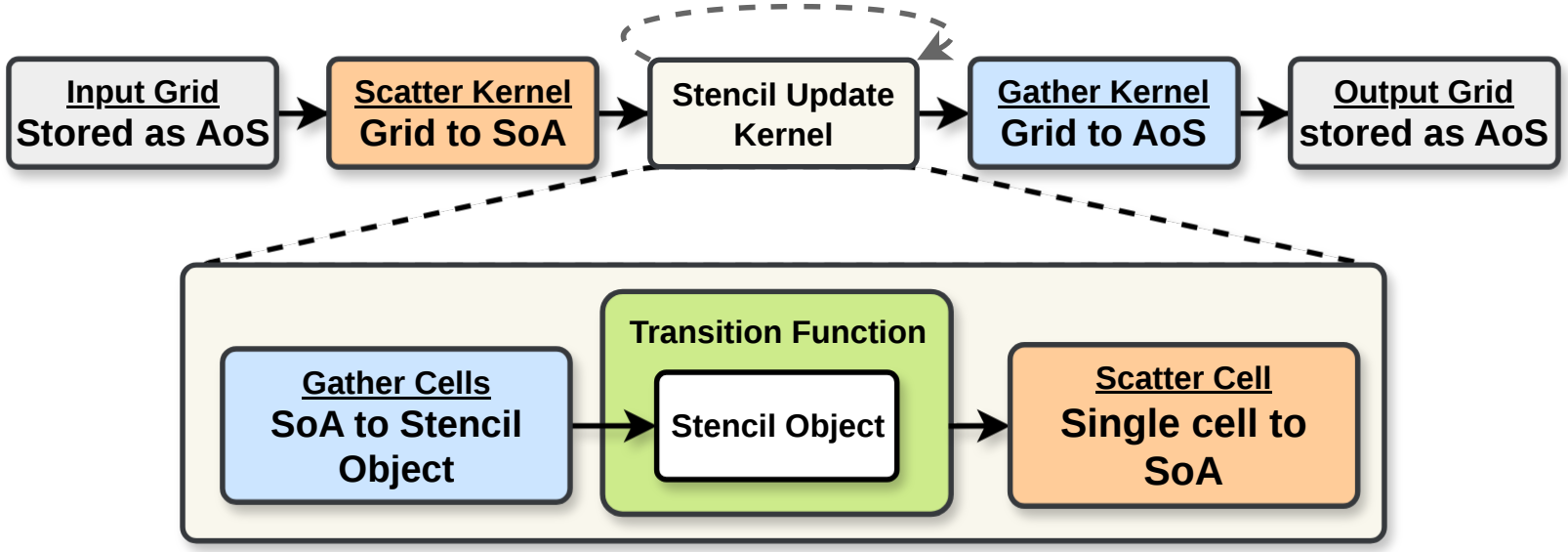
Paderborn Center for Parallel Computing

1. Stencil Computation and StencilStream
2. Memory access patterns
3. Transparent Scatter & Gather Design
4. Implementation details
5. Benchmarks

Transparent Scatter & Gather Design

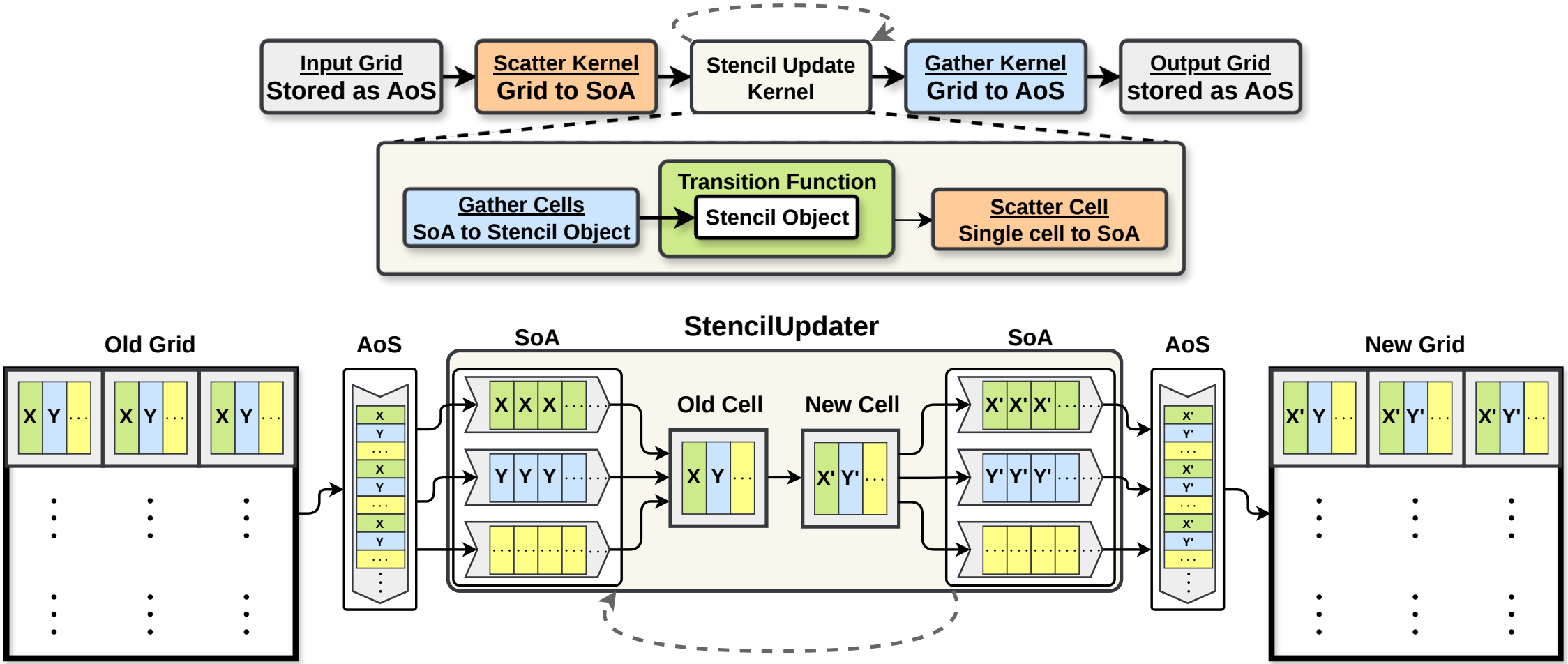
Paderborn Center for Parallel Computing

- Solution: conversion in the backend between layouts
- **Scatter**: converts input grid/cell from Array-of-Structs → Struct-of-Arrays
- Stencil Update kernel: operates entirely in SoA
- **Gather**: converts result SoA field buffers back to AoS output grid



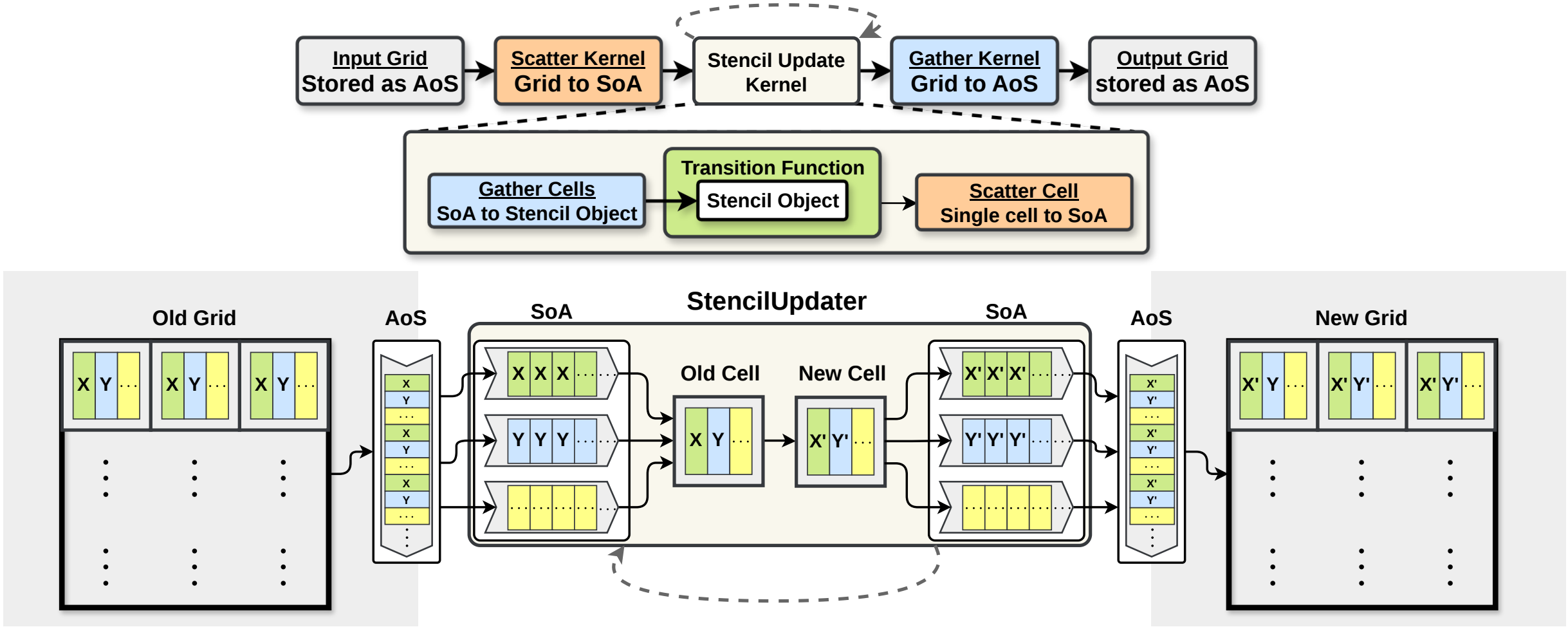
Transparent Scatter & Gather Design

Paderborn Center for Parallel Computing



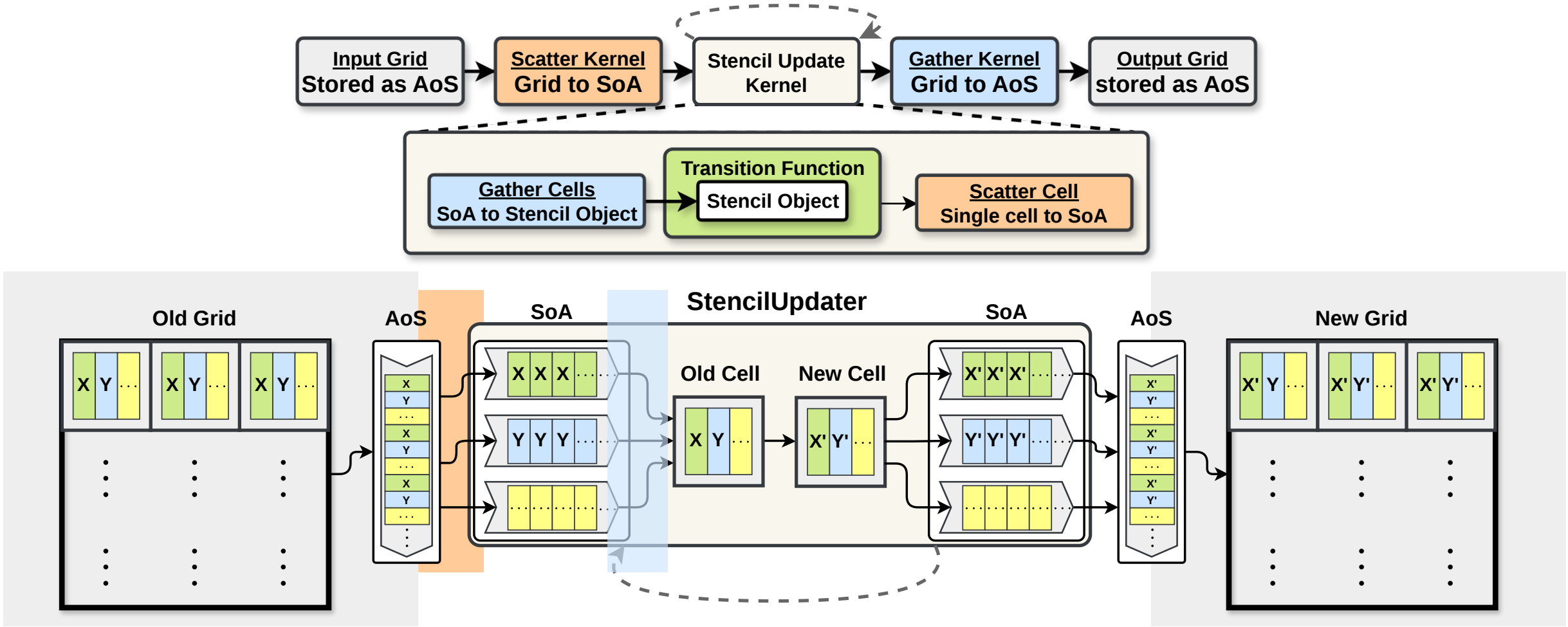
Transparent Scatter & Gather Design

Paderborn Center for Parallel Computing



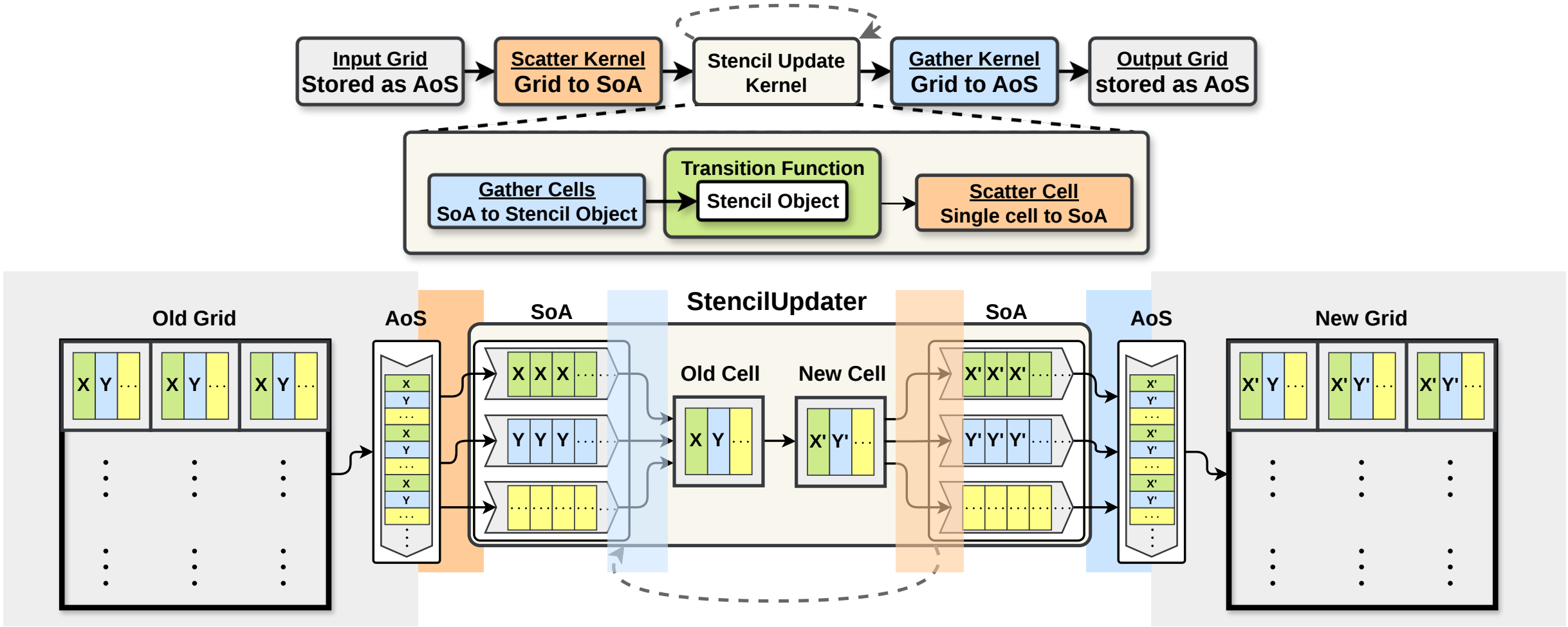
Transparent Scatter & Gather Design

Paderborn Center for Parallel Computing



Transparent Scatter & Gather Design

Paderborn Center for Parallel Computing



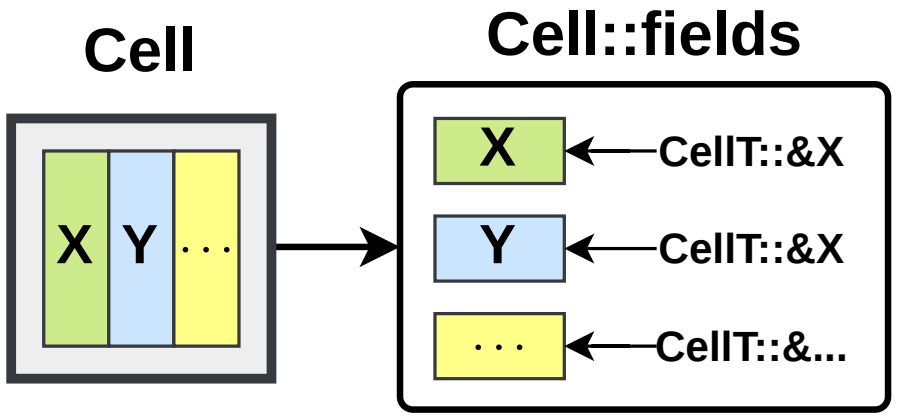
Implementation: Cell Fields Tuple

Paderborn Center for Parallel Computing

- User adds one static constexpr auto fields = std::make_tuple(...)

```
struct Cell {
  double X;
  float Y;
  /* ... */
  static constexpr auto fields = std::make_tuple(&Cell::X, &Cell::Y /* ... */);
};
```

- Contains member pointers to all fields
- Used at compile time to:
  1. determine number of buffers needed
  2. derive field datatypes
  3. drive scatter/gather operations



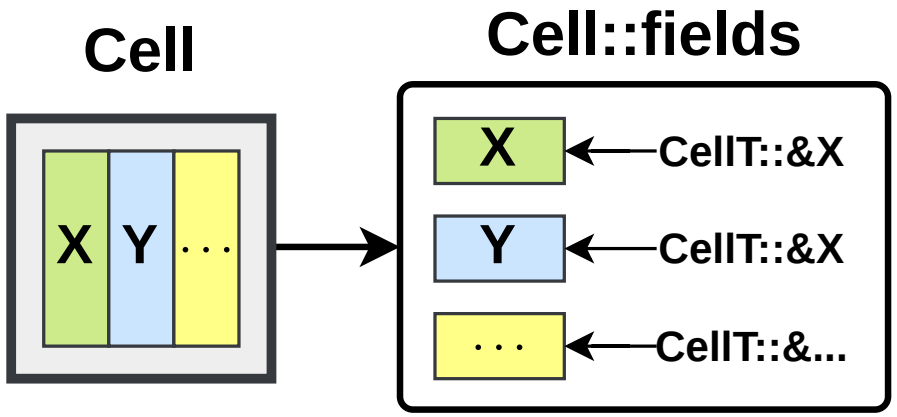
Implementation: Cell Fields Tuple

Paderborn Center for Parallel Computing

- User adds one static constexpr auto fields = std::make_tuple(...)

```
struct Cell {
  double X;
  float Y;
  /* ... */
  static constexpr auto fields = std::make_tuple(&Cell::X, &Cell::Y /* ... */);
};
```

- Contains member pointers to all fields
- Used at compile time to:
  1. determine number of buffers needed
  2. derive field datatypes
  3. drive scatter/gather operations

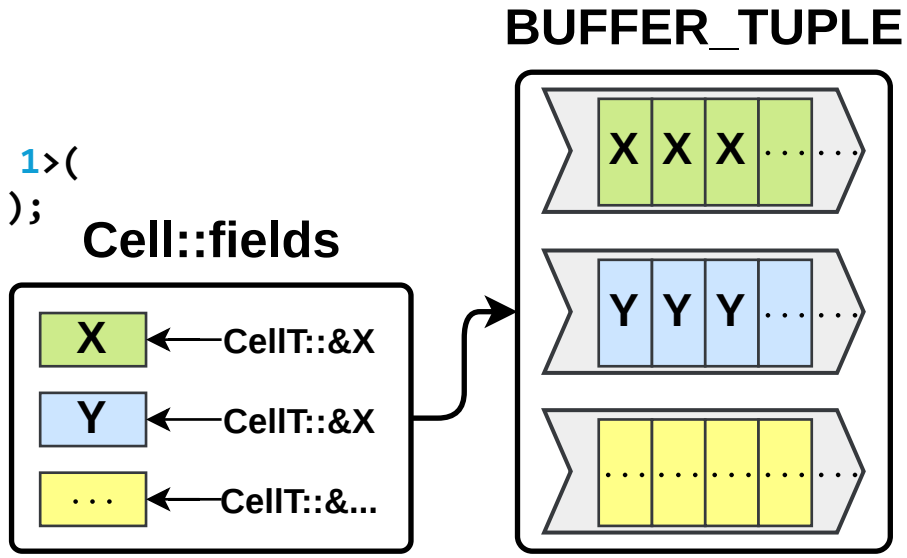


Implementation: Compile-Time Buffer

Paderborn Center for Parallel Computing

- alloc_field_buffers() uses std::apply to unpack the fields tuple
- Parameter pack expansion creates one sycl::buffer<T> per field
  - type deduced via std::declval

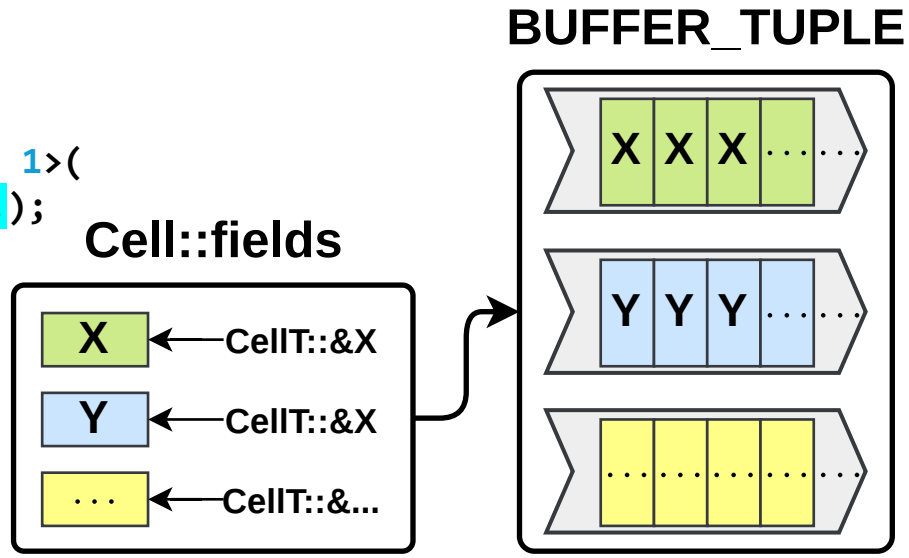
```
template <typename CellT> auto alloc_field_buffers(std::size_t N) {
    return std::apply(
        [N](auto... ptrs) {
            return std::make_tuple(
                sycl::buffer<std::remove_reference_t<decltype(
                                    std::declval<CellT>().*ptrs)>, 1>(
                                            sycl::range<1>(N))...);
        },
        CellT::fields);
}
```



Implementation: Compile-Time Buffer

- alloc_field_buffers() uses std::apply to unpack the fields tuple
- Parameter pack expansion creates one sycl::buffer<T> per field
  - type deduced via std::declval

```
template <typename CellT> auto alloc_field_buffers(std::size_t N) {
    return std::apply(
        [N](auto... ptrs) {
            return std::make_tuple(
                sycl::buffer<std::remove_reference_t<decltype(
                                    std::declval<CellT>().*ptrs)>, 1>(
                                            sycl::range<1>(N))...);
        },
        CellT::fields);
}
```

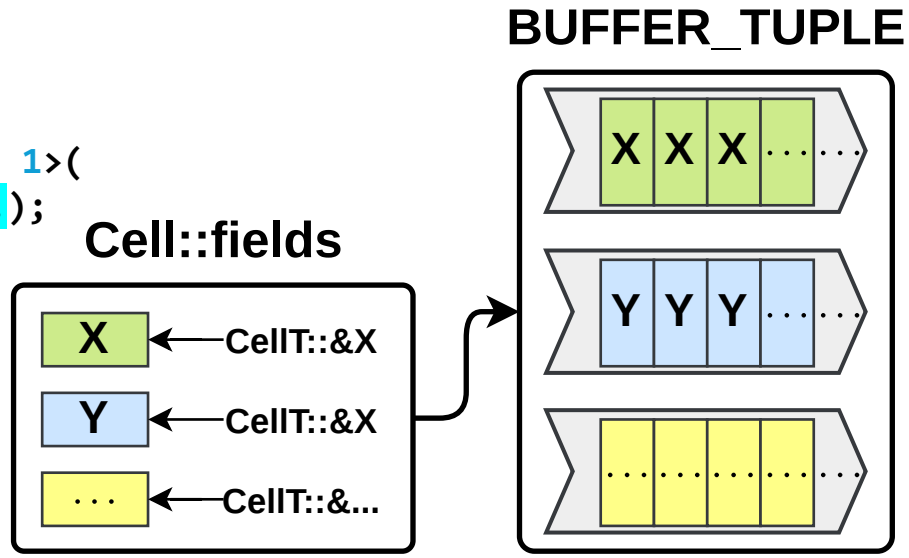


Implementation: Compile-Time Buffer

- alloc_field_buffers() uses std::apply to unpack the fields tuple
- Parameter pack expansion creates one sycl::buffer<T> per field
  - type deduced via std::declval

```
template <typename CellT> auto alloc_field_buffers(std::size_t N) {
    return std::apply(
        [N](auto... ptrs) {
            return std::make_tuple(
                sycl::buffer<std::remove_reference_t<decltype(
                                    std::declval<CellT>().*ptrs)>, 1>(
                                            sycl::range<1>(N))...);
        },
        CellT::fields);
}
```

- Result: A heterogeneous tuple of correctly-typed SYCL buffers

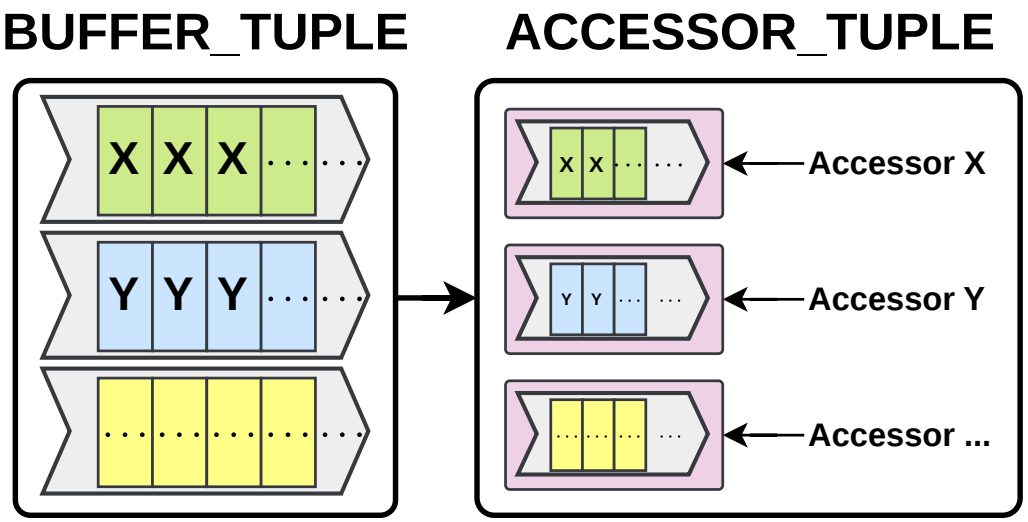


Implementation: Accessor Creation

Paderborn Center for Parallel Computing

- We need accessors to access the data in the buffers
- Accessors created the same way as Buffers

```
auto ACCESSOR_TUPLE = std::apply([&](auto& ...buf) {
    return std::make_tuple(sycl::accessor(buf, cgh, sycl::write_only)...);
    },
    BUFFER_TUPLE);
```

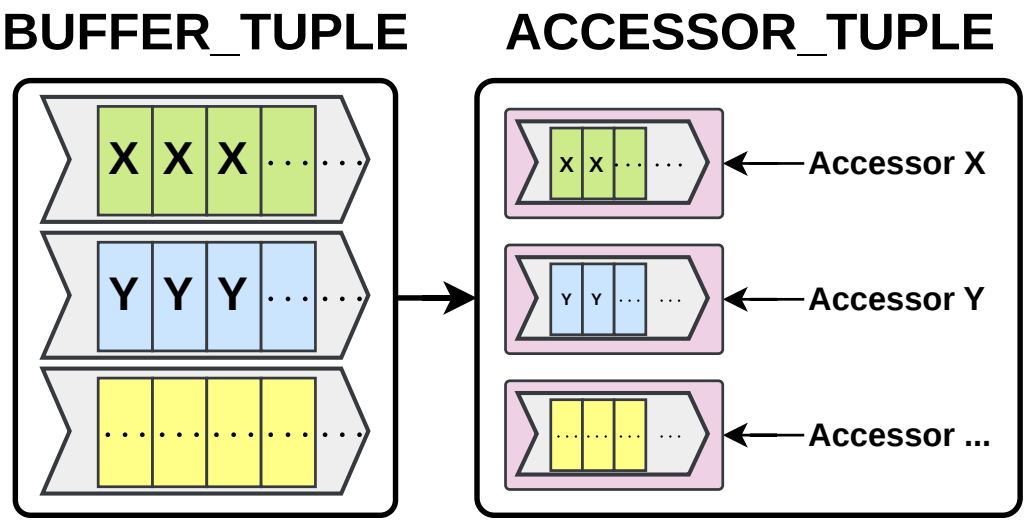


Implementation: Accessor Creation

Paderborn Center for Parallel Computing

- We need accessors to access the data in the buffers
- Accessors created the same way as Buffers

```
auto ACCESSOR_TUPLE = std::apply([&](auto& ...buf) {
    return std::make_tuple(sycl::accessor(buf, cgh, sycl::write_only)...);
    },
    BUFFER_TUPLE);
```

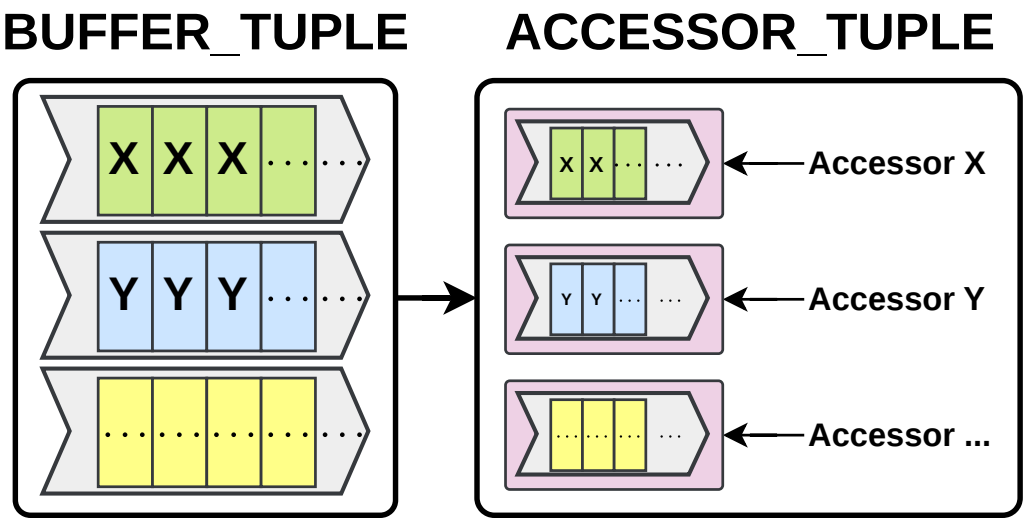


Implementation: Accessor Creation

Paderborn Center for Parallel Computing

- We need accessors to access the data in the buffers
- Accessors created the same way as Buffers

```
auto ACCESSOR_TUPLE = std::apply([&](auto& ...buf) {
    return std::make_tuple(sycl::accessor(buf, cgh, sycl::write_only)...);
    },
    BUFFER_TUPLE);
```

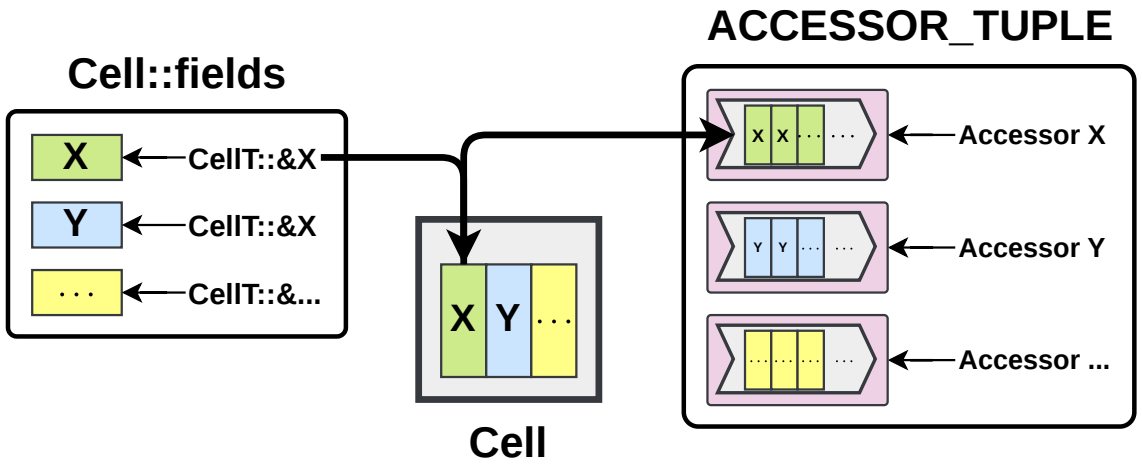


Implementation: Gather & Scatter

Paderborn Center for Parallel Computing

- Scatter lambda: acc[cell_id] = cell.*member_ptr
  - **writes each field to its buffer**

```
Cell cell = source_grid[id[0]][id[1]];
size_t cell_id = id[0] * source_grid.get_range()[1] + id[1];

for_each_in_two_tuples(ACCESSOR_TUPLE, Cell::fields, [&](auto &acc, auto member_ptr) {
    acc[cell_id] = static_cast<std::remove_reference_t<decltype(acc[cell_id])>>(cell.*member_ptr);
});
```

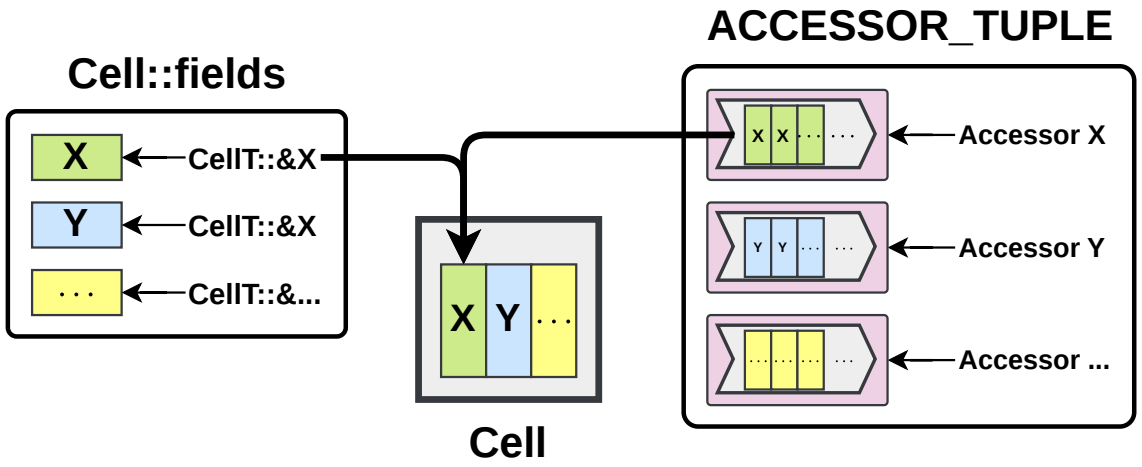


Implementation: Gather & Scatter

Paderborn Center for Parallel Computing

- Gather lambda: cell.*member_ptr = acc[cell_id]
  - **reads each field back into a cell struct**

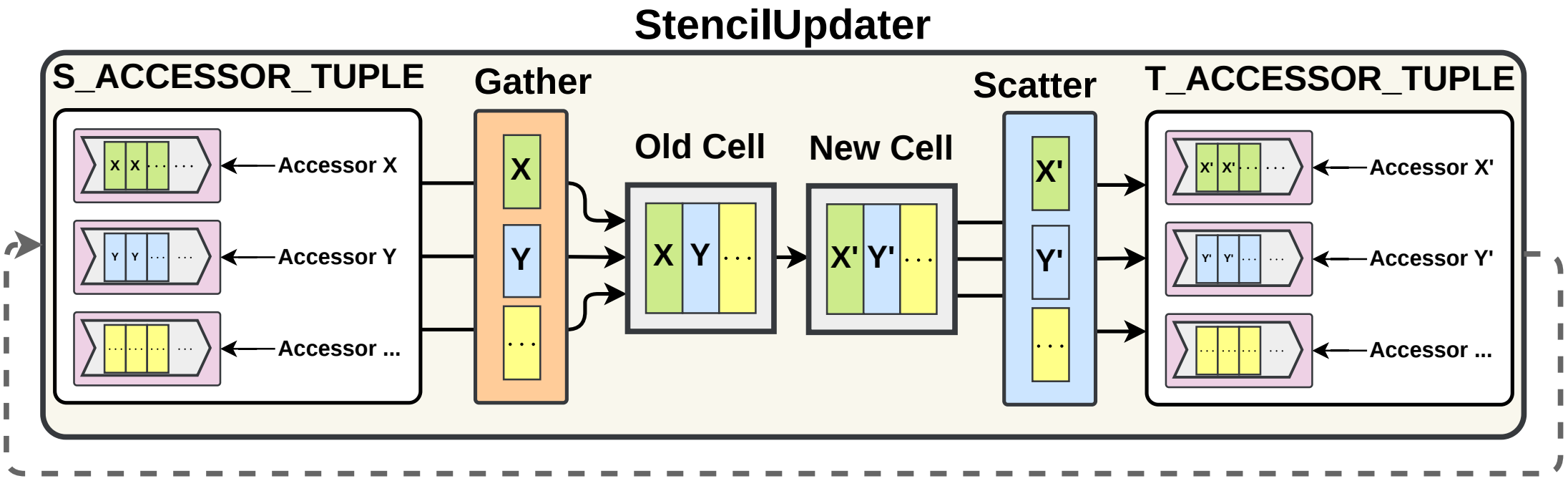
```
Cell cell;
size_t cell_id = id[0] * target_ac.get_range()[1] + id[1];

for_each_in_two_tuples(ACCESSOR_TUPLE, Cell::fields, [&](auto &acc, auto member_ptr) {
    cell.*member_ptr = static_cast<std::remove_reference_t<decltype(acc[cell_id])>> (acc[cell_id]);
});
```



Implementation: Gather & Scatter

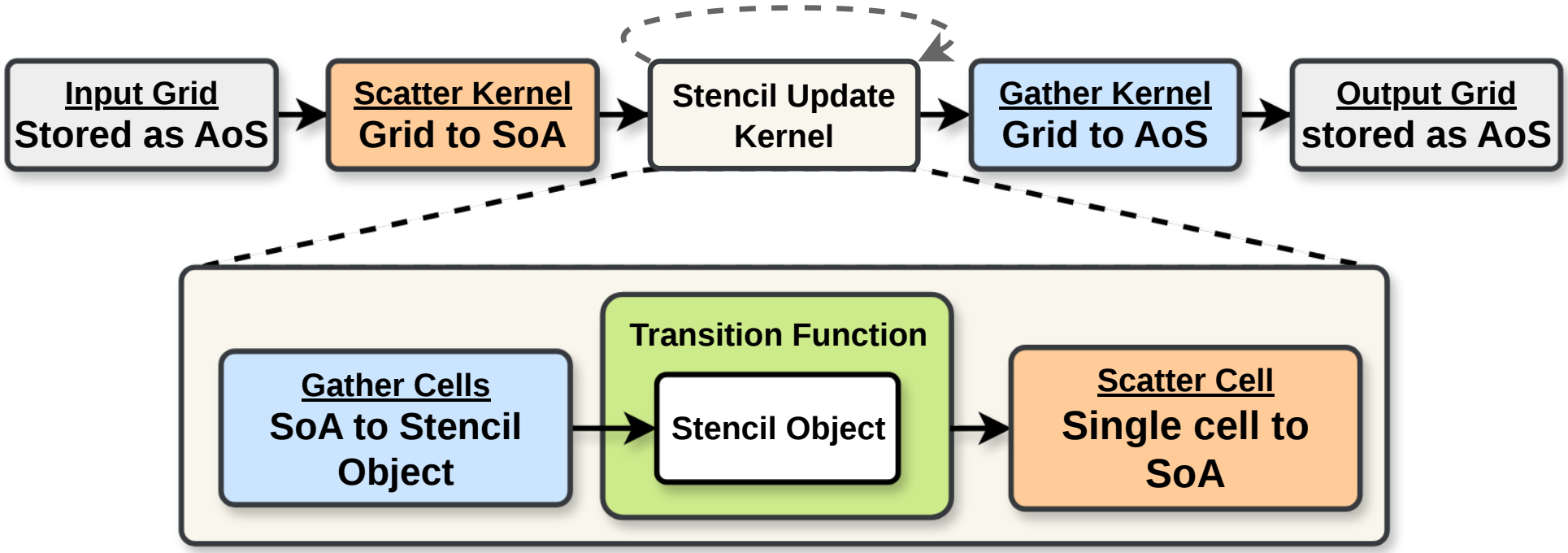
Paderborn Center for Parallel Computing

- Gather lambda: **reads each field back into a cell struct**
- Scatter lambda: **writes each field to its buffer**



Compiler optimizes the memory loads

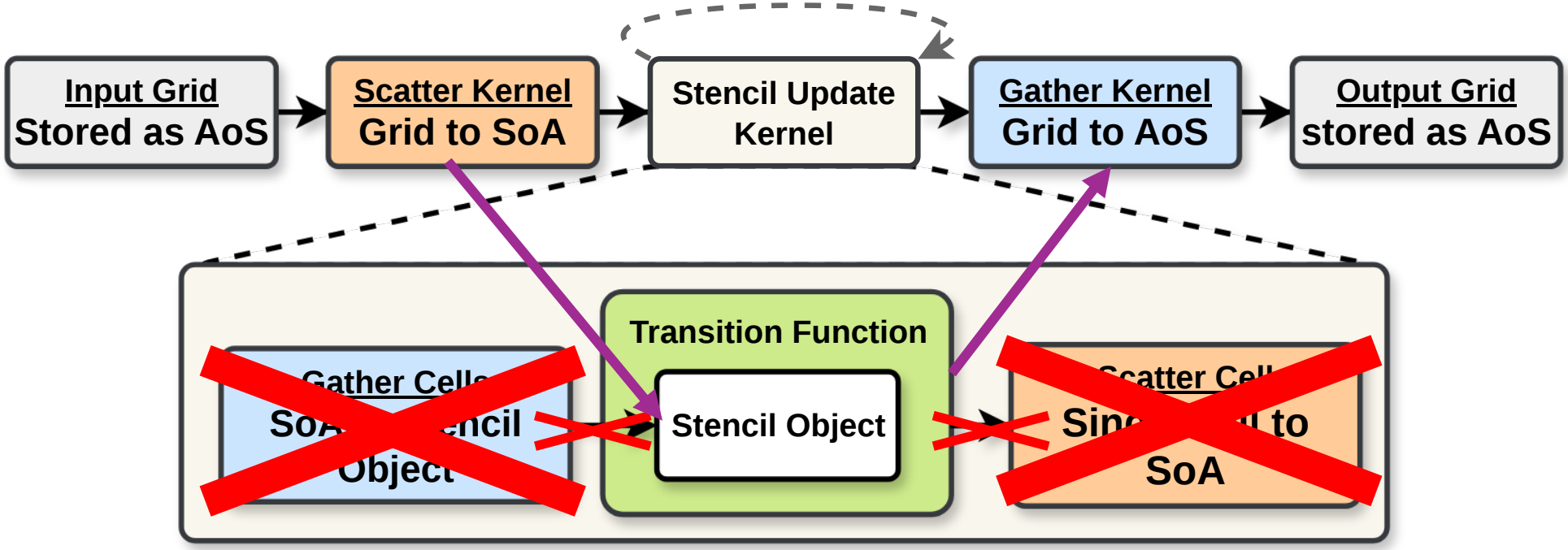
Paderborn Center for Parallel Computing



Compiler optimizes the memory loads

Paderborn Center for Parallel Computing

- The compiler omits the Array of Struct construction of the Stencil Object and loads directly from the Struct of Arrays



Agenda

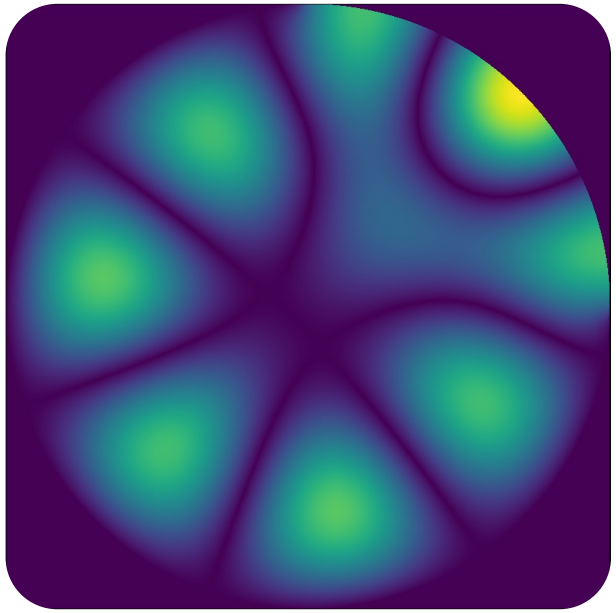
Paderborn Center for Parallel Computing

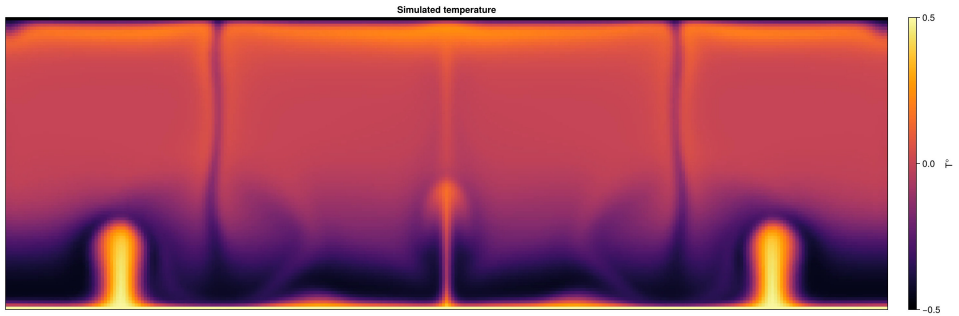
1. Stencil Computation and StencilStream
2. Memory access patterns
3. Transparent Scatter & Gather Design
4. Implementation details
5. Benchmarks

Benchmark setup

Paderborn Center for Parallel Computing

- Noctua 2 supercomputer
  - Paderborn Center for Parallel Computing
- GPU: Nvidia A100 40GB SXM
- FPGA: BittWare 520N | Altera/Intel Stratix 10 2800 GX
- **Jacobi 5-point (1 float per cell):**
  - Classic stencil benchmark
- **HotSpot (2 float per cell):**
  - IC heat simulation, from Rodinia suite
- **FDTD (8 float per cell):**
  - Photonic wave simulation
  - Uses sub-iterations & time-dependent values
- **Convection (PT kernel) (11 double per cell):**
  - Most complex, pressure & velocity solver

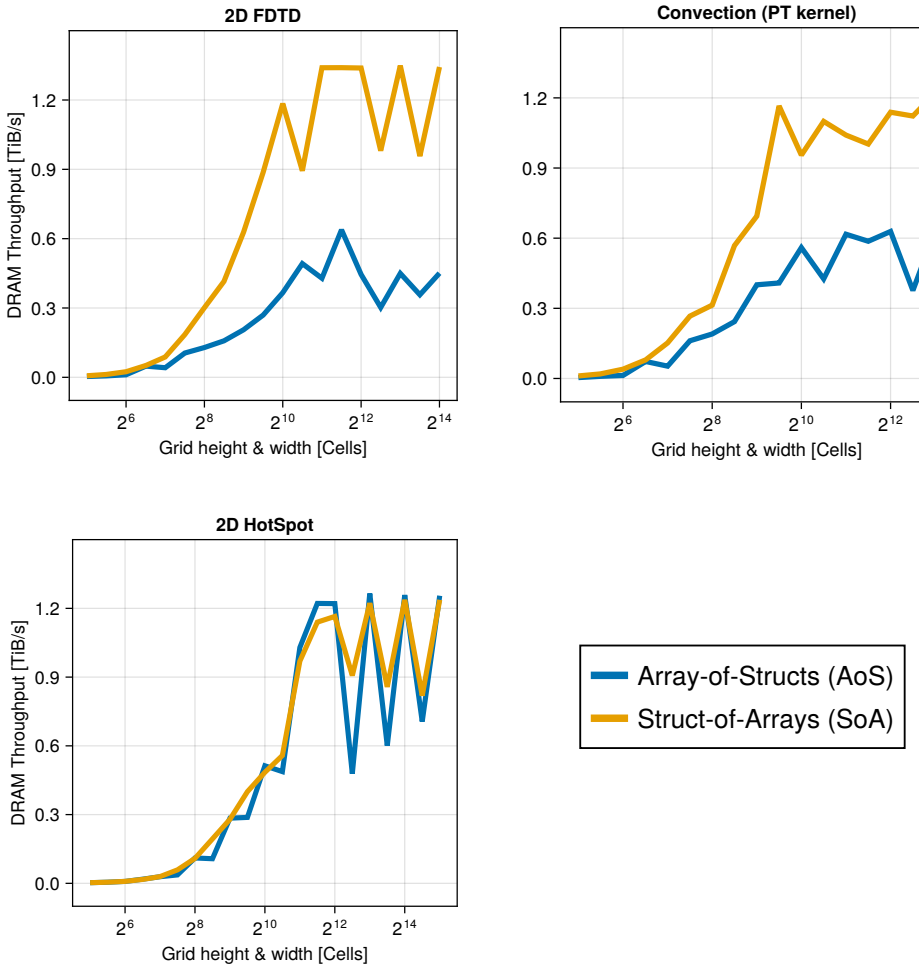




Effectiveness of AoS → SoA

Paderborn Center for Parallel Computing

- **FDTD & Convection** (large cell):
  - SoA achieves significantly higher memory throughput
  - **AoS:** 309 – 654 GB/s
    - ~20% – 42% of peak bandwidth
  - **SoA:** 978 GB/s – 1.35 TB/s
    - ~63% – 88.8% of peak bandwidth
  - Speedup: 1.7× – 3.2×
- **HotSpot** (small cell):
  - Little to no impact from layout choice



Memory Sectors

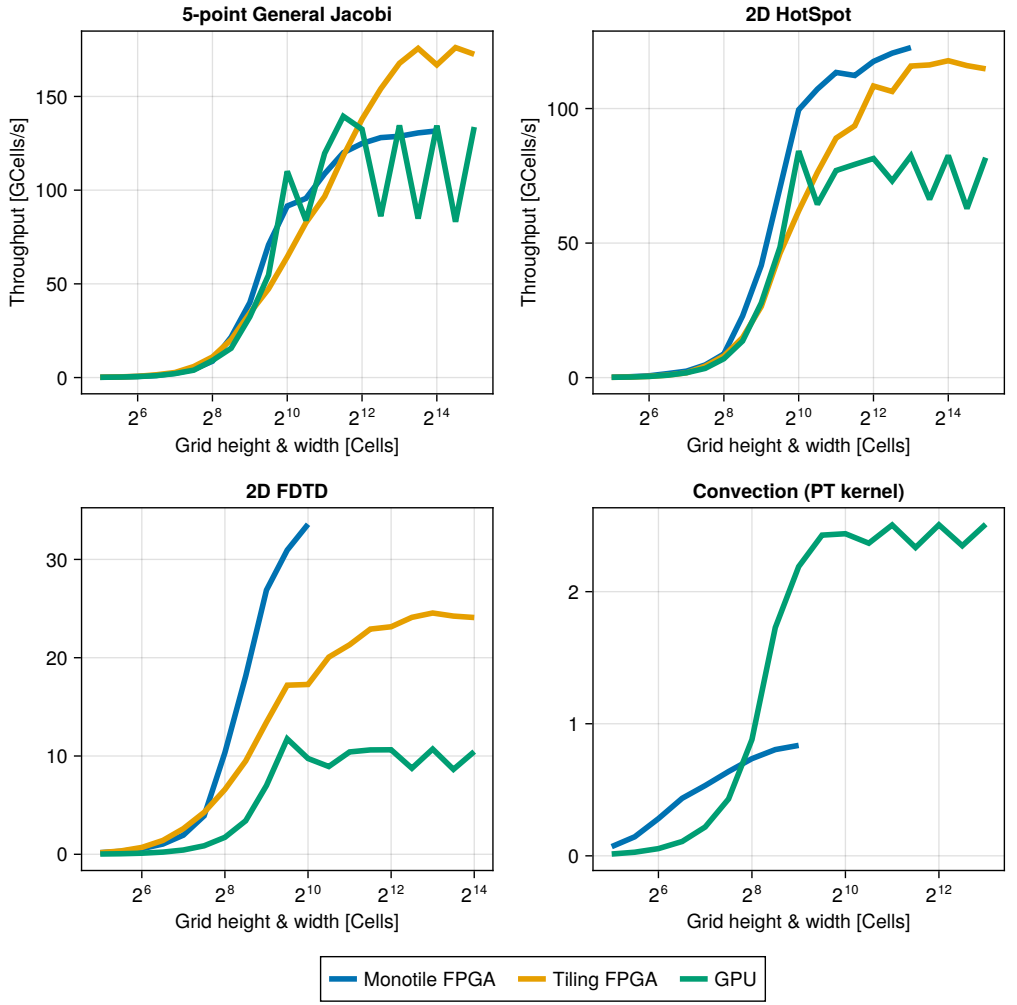
- GPU loads memory in fixed-size sectors (32 bytes)
- Median sectors per request (S/R) (fewer is better)
  - Median over avg. for each grid size
- **Array-of-Structs**
  - Up to **~32 sectors per request**
  - Low bandwidth utilization
- **Struct-of-Arrays**
  - Only **~4–8 sectors per request**
  - High bandwidth utilization
- Strided access leads to higher number of Read/Write operations

Application	Layout	Median S/R	
		Read	Write
Jacobi	N/A	4.37	4.0
HotSpot	SoA	4.31	4.0
HotSpot	AoS	8.3	8.0
FDTD	SoA	4.15	4.0
FDTD	AoS	31.98	32.0
Convection	SoA	8.30	8.0
Convection	AoS	31.89	32.0

Throughput comparison

Paderborn Center for Parallel Computing

- FPGA Tiling: up to 176 GCells/s
- GPU SoA: competitive up to 139 GCells/s
- GPU Oscillations due to in-bounds check overhead
- FPGA designs and further optimizations in the paper



Limitations & Related Work

Paderborn Center for Parallel Computing

- Current GPU backend is bound by global memory
  - small workload per thread limits compute utilization (32–76%)
- Temporal blocking could unlock higher throughput
- Implement work-group-level tiling in StencilStream's GPU backend
- Current GPU backend is already sufficient for fast application development with short compile times vs. 12h FPGA builds

Questions ?

Paderborn Center for Parallel Computing

- GPU:
  - We presented a GPU backend for StencilStream
  - Achieves 62–89% of peak bandwidth
  - Template metaprogramming for automatic layout transformations
- FPGA:
  - Strong improvements via spatial parallelism
  - Near hardware limits (DSP utilization ~75–83%)