

Subsetix: Sparse 2D Geometry on GPU

From Set Algebra to AMR Simulation

Sébastien DUBOIS

HPC@Maths Team

December 2025



Outline

I. Context

1. GPU Computing & Kokkos

II. Sparse Representation

2. Intervals and CSR
3. 2D Sparse Mesh Example

III. Data Structures

4. **Device-Side Overview**
5. IntervalSet2D, Field2D, SubSet
6. Workspace & AMR

IV. Algorithms

7. Geometry Constructors
8. Set Algebra
9. Field Operations
10. Morphology & AMR

V. Demo

11. Mach2 Cylinder (Multi-level AMR)

VI. Appendices

- Project Evolution
- Why Kokkos?
- Development Methodology

I. Context: GPU & Kokkos

Project Context — Towards Exascale

Background

- **Numpex Project:** French initiative pushing scientific computing to exascale
- **Samurai:** AMR library with unique sparse data structure (interval-based)
- **Challenge:** No prior GPU implementation of Samurai's core concepts

Objective

How can Samurai's strategy evolve for exascale?

- GPU acceleration (today's focus)
- Multi-node distribution (future work)

Approach

Proof of Concept Strategy

1. **Simplify** — Isolate core problems
2. **Prototype** — Build independent bricks
3. **Validate** — Test on real simulations
4. **Integrate** — Path back to Samurai

This work: GPU-native sparse 2D geometry as a standalone proof of concept

GPU Architecture — Massively Parallel

Execution Hierarchy

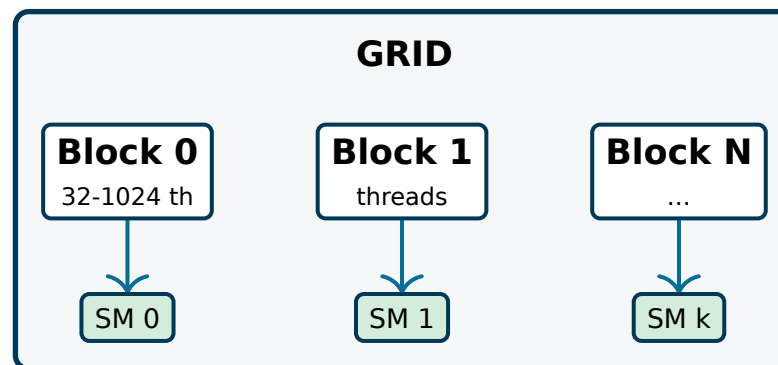


- **Warp** = 32 threads in **lockstep** (SIMT)
- **SM** = autonomous compute unit
- Multiple warps/SM → latency hiding

For Our Project

- **1 thread** = processes 1 Y row (or 1 cell)
- Thousands of rows → **saturate the GPU**

Execution Model



B200 vs EPYC 9965

	GPU B200	CPU EPYC 9965
Bandwidth	8 TB/s	576 GB/s
FP32	90 TFlops	14 TFlops

GPU: **14× more bandwidth** than CPU → ideal for large sparse meshes

Kokkos — Performance Portability

The Problem

- CUDA = NVIDIA only
- OpenMP = CPU only (limited GPU)
- HIP = AMD only
- Rewrite for each platform?

The Solution: Kokkos

```
// 1. COUNT – unknown result size
parallel_for(num_rows, KOKKOS_LAMBDA(int r) {
    counts[r] = count_intervals(r);
});
// 2. SCAN – compute offsets
exclusive_scan(counts, row_ptr);
// 3. FILL – parallel write
parallel_for(num_rows, KOKKOS_LAMBDA(int r) {
    fill_intervals(r, &out[row_ptr[r]]);
});
```

CUDA vs Kokkos

Native CUDA

```
double* d_data;
cudaMalloc(&d_data, n*8);

cudaMemcpy(d_data, h_data,
            n*8, HostToDevice);

kernel<<<B,T>>>(d_data, n);

cudaMemcpy(h_data, d_data,
            n*8, DeviceToHost);

cudaFree(d_data);
```

Kokkos

```
View<double*> data("d", n);
auto h = create_mirror_view(data);

deep_copy(data, h);

parallel_for(n, KOKKOS_LAMBDA(int
i){
    data(i) = compute(i);
});

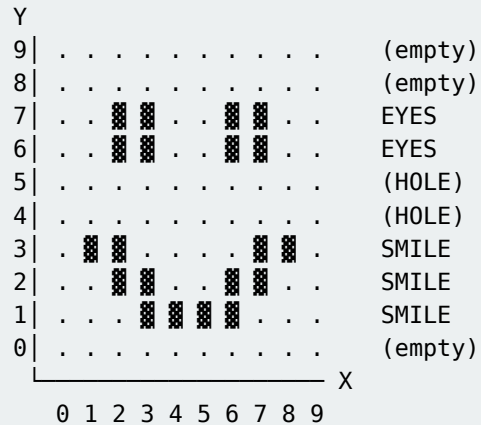
deep_copy(h, data);
// Automatic cleanup (RAII)
```

Single source code → compiles for OpenMP, CUDA, HIP, SYCL, Serial

II. Sparse Representation

Example: 2D Sparse Mesh with Intervals

“Smiley” Geometry :-)



Sparse-CSR-like Representation

```
// 5 rows, HOLE Y=4,5  
row_keys = [1, 2, 3, 6, 7] // skips 4,5!  
num_rows = 5
```

```
// Rows with 1 or 2 intervals  
row_ptr = [0, 1, 3, 5, 7, 9]
```

```
intervals = [  
    {3, 7}, // Y=1: smile bottom  
    {2, 4}, {6, 8}, // Y=2: smile thick  
    {1, 3}, {7, 9}, // Y=3: smile corners  
    {2, 4}, {6, 8}, // Y=6: EYES bottom  
    {2, 4}, {6, 8}, // Y=7: EYES top  
]
```

```
num_intervals = 9
```

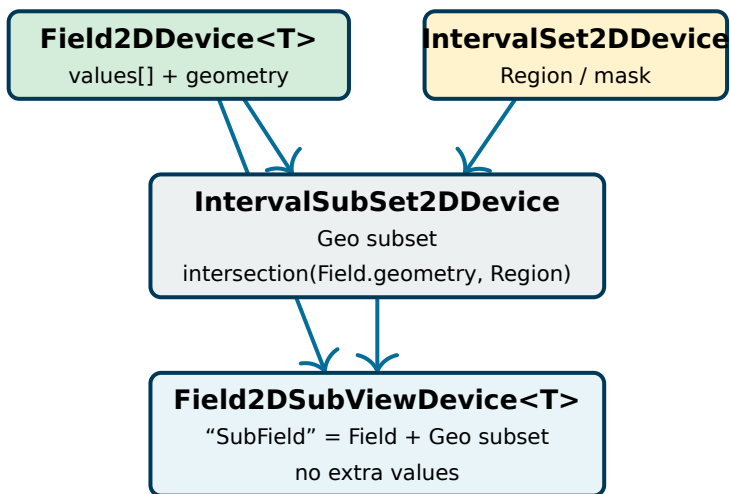
```
cell_offsets = [0,4,6,8,10,12,14,16,18,20]  
total_cells = 20
```

Hole Y=4,5: row_keys jumps from 3 to 6

III. Data Structures

Overview — Device Structures

Core Types



Device-Side Grammar

- `Field2DDevice<T>` = field values + `IntervalSet2DDevice` geometry
- `IntervalSet2DDevice (region)` = mask / target cells
- `IntervalSubSet2DDevice` = geo subset = geometry \cap region
- `Field2DSubViewDevice<T>` = “SubField” = Field + `IntervalSubSet2DDevice`

`Field2DSubViewDevice<T>` internally uses `IntervalSubSet2DDevice` to iterate only on active cells.

SubField: Usage Example

```
Field2DDevice<Real> rho(fluid_geo);    // Field

// Region = any IntervalSet2DDevice (BC, AMR, overlap)
IntervalSet2DDevice left_bc = make_box_device({0,2,0,ny});

// SubField = rho restricted to left_bc
Field2DSubViewDevice<Real> sub = make_subview(rho, left_bc);

// Apply operations only on this region
fill_subview_device(sub, rho_inlet);
apply_stencil_on_subview_device(sub, bc_stencil);
```

SubView Operations

- `fill_subview_device(sub, val)`
- `scale_subview_device(sub, alpha)`
- `copy_subview_device(dst, src)`
- `apply_stencil_on_subview_device(...)`

In `mach2_cylinder`, `overlap/guard` regions and AMR masks use exactly this SubField (SubView) + region mask design.

IntervalSet2D — Complete CSR Structure

Sparse 2D geometry stored as **rows of X-intervals**, indexed by Y coordinate — similar to CSR matrix format.

C++ Definition

```
template<class MemorySpace>
struct IntervalSet2D {
    // Y coordinates of non-empty rows
    View<RowKey2D*> row_keys; // [num_rows]

    // Index into intervals[] for each row
    View<size_t*> row_ptr; // [num_rows + 1]

    // All intervals (contiguous)
    View<Interval*> intervals; // [num_intervals]

    // Linear cell offsets
    View<size_t*> cell_offsets; // [num_intervals]

    size_t total_cells;
    int num_rows;
    int num_intervals;
};
```

Basic Types

```
using Coord = std::int32_t;

struct Interval {
    Coord begin = 0; // Inclusive
    Coord end = 0; // Exclusive
};

struct RowKey2D {
    Coord y = 0;
};
```

Invariants

- row_keys sorted by increasing Y
- Intervals sorted by X within each row
- No overlap between intervals
- row_ptr[r+1] - row_ptr[r] = nb intervals row r


Field2D — Field on Sparse Geometry

Associates a **contiguous array of values** with each cell of an IntervalSet2D geometry.

Definition

```
template<class T, class MemorySpace>
struct Field2D {
    IntervalSet2D geometry; // Geometry ref
    View<T*> values;        // [total_cells]
};
```

Memory Layout

Geometry:	
values[]:	[v0 v1 v2 v3 v4 v5 v6]
	↑ ↑ ↑
offsets:	0 2 4

Contiguous values → cache-friendly

Cell Access

```
// 0(1) - interval index + x coordinate
T val = field.at(interval_idx, x);
```

Usage

```
Field2DDevice<double> rho(fluid_geo);
fill_field_device(rho, 1.0);
auto rho_host = to_host(rho); // I/O
```

SubSet — Targeted Region Operations

Represents a **subset of the parent geometry** (intersection with a mask) — used by SubFields to restrict operations to specific cells.

Structure

```
struct IntervalSubSet2D {  
    IntervalSet2D parent; // ref to Field geo  
    interval_indices[]; // which intervals  
    x_begin[], x_end[]; // restricted range  
    row_indices[]; // Y row in parent  
    num_entries;  
};
```

Usage

```
// Build subset (intersection)  
build_interval_subset(  
    field.geometry, mask, subset, &ctx);  
  
// Operations on subset only  
fill_on_subset(field, subset, 0.0);
```

1D Example: Intersection

Parent:	[==A==]	[==B==]	[==C==]
idx:	0	1	2
	0 8 12	18 22	30
Mask:	[=====M=====]		
	5		25
SubSet:	[=]	[==B==]	[=]
	5 8 12	18 22 25	
	↑	↑	↑
entry:	0	1	2

SubSet = references to Parent

entry	interval_idx	x_begin	x_end
0	0 (A)	5	8
1	1 (B)	12	18
2	2 (C)	22	25

No data copy — just indices + bounds

⚠ Structure too complex — needs simplification

Field2DSubView — View on Field + Region

Combines a Field with a target region for **localized operations**.

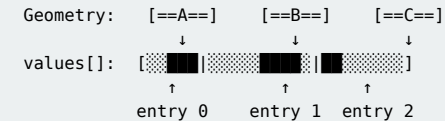
Structure

```
struct Field2DSubView<T> {  
    Field2D<T> parent;      // ref to field  
    IntervalSet2D region;    // where to operate  
    IntervalSubSet2D subset; // intersection  
};
```

Lazy Pattern

```
// 1. Create (no computation)  
Field2DSubViewDevice<T> sub = make_subview(field, region);  
// sub.subset is empty  
  
// 2. First op with ctx → triggers build  
fill_subview_device(sub, 0.0, &ctx);  
// sub.subset = field.geo n region  
  
// 3. Next ops reuse subset  
scale_subview_device(sub, 2.0); // fast!  
fill_subview_device(sub, 1.0);  // fast!
```

Memory Mapping



⋮ = skipped ■ = accessed by SubSet

Workspace & AMR Support

Reusable buffer pool to avoid repeated GPU allocations, and multi-resolution grid structure for AMR.

UnifiedCsrWorkspace

Pool of reusable buffers

```
struct UnifiedCsrWorkspace {
    View<int*> int_bufs_[5];           // indices, maps, flags
    View<size_t*> size_t_bufs_[2];    // row_ptr, counts
    View<RowKey2D*> row_key_bufs_[2]; // Y coords
    View<Interval*> interval_buf_0;   // X intervals

    auto get_int_buf(int id, size_t n) {
        if (n > int_bufs_[id].extent(0))
            Kokkos::resize(int_bufs_[id], n*1.5);
        return subview(int_bufs_[id], {0,n});
    }
};
```

Avoids repeated GPU allocations
Crucial for chained operations

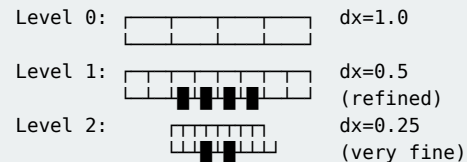
Note: Outputs must be pre-allocated
Use `allocate_interval_set_device()`

MultilevelGeo (AMR)

Multi-resolution grids

```
template<class MemorySpace>
struct MultilevelGeo {
    double origin_x, origin_y;
    double root_dx, root_dy;
    int num_active_levels;
    Array<GeoView, 16> levels;

    double dx_at(int level) {
        return root_dx / (1 << level);
    }
};
```



IV. Algorithms

Binary Search — $O(\log n)$ Lookups Everywhere

CSR structure requires binary search for row and interval lookups — efficient but suboptimal on GPU.

CSR Requires Sorted Data

All lookups rely on binary search:

1. Find row by Y coordinate

```
int find_row_by_y(row_keys, num_rows, y) {  
    // Binary search in row_keys[]  
    return lower_bound(row_keys, y);  
}
```

$O(\log R)$ — R = number of rows

2. Find interval by X coordinate

```
int find_interval_by_x(intervals, begin, end, x) {  
    // Binary search in intervals[begin..end]  
    return lower_bound(intervals, x);  
}
```

$O(\log I_{\text{row}})$ — I = intervals in row

Combined: Cell Lookup

```
T& get(Coord x, Coord y) {  
    // Step 1: find row  
    int row = find_row_by_y(row_keys, y);  
    // Step 2: find interval in row  
    int iv = find_interval_by_x(  
        intervals, row_ptr[row], row_ptr[row+1], x);  
    // Step 3: compute offset  
    return values[offsets[iv] + (x - intervals[iv].begin)];  
}
```

Total: $O(\log R + \log I)$

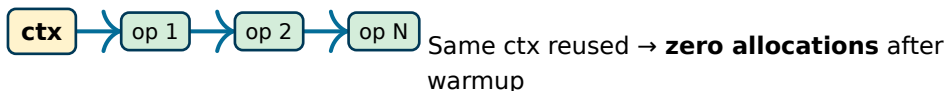
GPU: Binary search = suboptimal
(future work)

Set Algebra — Binary Operations

Binary set operations (\cup , \cap , \setminus) combine geometries using a shared workspace to avoid repeated GPU allocations.

CsrSetAlgebraContext

```
struct CsrSetAlgebraContext {
    UnifiedCsrWorkspace workspace;
    // Pool of reusable GPU buffers:
    // - int_bufs_[5], size_t_bufs_[2]
    // - row_key_bufs_[2], interval_buf_
    // Auto-grows on demand, never shrinks
};
```



Complete Example

```
CsrSetAlgebraContext ctx; // create once

auto domain = make_box_device({0,400,0,160});
auto obstacle = make_disk_device({80,80,20});

auto fluid = allocate_interval_set_device(
    domain.num_rows,
    domain.num_intervals + obstacle.num_intervals);

set_difference_device(domain, obstacle, fluid, ctx);
```

Chaining with Buffer Reuse

```
CsrSetAlgebraContext ctx;

// Pre-allocate output buffers ONCE
auto set1 = allocate_interval_set_device(512, 2048);
auto set2 = allocate_interval_set_device(512, 2048);

// Compute: set1 = A  $\cup$  B
set_union_device(A, B, set1, ctx);

// Compute: set2 = set1  $\setminus$  C
set_difference_device(set1, C, set2, ctx);

// ... use set2 (e.g., create Field2D on it) ...

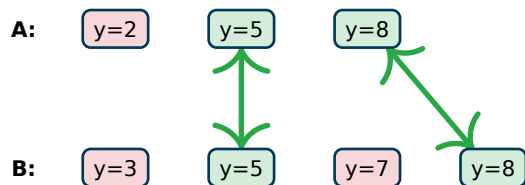
// Later: reuse same buffers!
set_intersection_device(D, E, set1, ctx); // set1 reused
set_union_device(set1, F, set2, ctx);    // set2 reused
```

Allocate once → reuse for entire simulation
ctx + set1 + set2: zero GPU malloc in hot loop

Intersection — How It Works

Intersection uses the **Count-Scan-Fill** pattern: count output intervals, compute offsets via prefix sum, then fill in parallel.

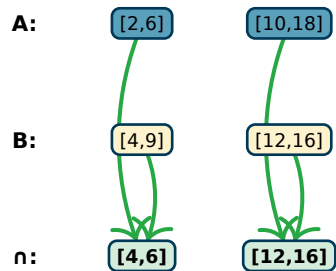
Phase 1: Row Mapping



Binary search: $O(\log n)$ per row

Phase 2: Interval Merge (per row)

Row $y=5$ — 2 intervals each



$O(n+m)$ sweep — $\max(\text{begin}), \min(\text{end})$

GPU Pattern: Count-Scan-Fill

Why? GPU threads can't dynamically allocate — output size must be known before parallel write.



1. **COUNT** — how many intervals per row?

```
row_counts[i] = count_intersect(A[i], B[i])
```

Parallel per row — don't write yet

2. **SCAN** — where does each row start?

```
row_ptr = exclusive_scan(row_counts)
```

Prefix sum \rightarrow $\text{row_ptr}[i]$ = write offset

3. **FILL** — write results at known offsets

```
fill_intersect(A[i], B[i], out, row_ptr[i])
```

Parallel per row — no conflicts!

Same pattern for \cup , \cap , \oplus

Row Mapping — Why and How

GPU parallelization requires knowing output rows before processing — row mapping creates a correspondence table between output and input rows.

GPU Constraint

1 thread = 1 output row

We need to know output rows **before** parallel processing.

The Mapping Structure

```
struct RowMergeResult {
    row_keys[]; // Y coords of output rows
    row_index_a[]; // index in A (-1 if absent)
    row_index_b[]; // index in B (-1 if absent)
};
```

Usage in Parallel

```
parallel_for(num_rows_out, [&](int i) {
    int ia = row_index_a[i]; // -1 or valid
    int ib = row_index_b[i]; // -1 or valid

    intervals_a = (ia >= 0) ? A.row(ia) : ∅;
    intervals_b = (ib >= 0) ? B.row(ib) : ∅;

    merge(intervals_a, intervals_b, out[i]);
});
```

Each thread knows exactly what to read → **no conflicts**

Concrete Example: $A \cup B$

A: y=2 y=5 y=8

B: y=3 y=5 y=8 y=9

i	y	idx_a	idx_b	Signification
0	2	0	-1	A[0] seul
1	3	-1	0	B[0] seul
2	5	1	1	A[1] ∪ B[1]
3	8	2	2	A[2] ∪ B[2]
4	9	-1	3	B[3] seul

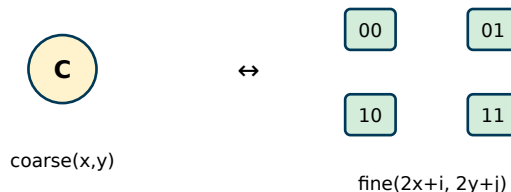
-1 = ligne absente dans ce set
Le mapping est construit par recherche binaire

Op	Mapping
$A \cap B$	Garde les y communs
$A \cup B$	Fusionne tous les y
$A \setminus B$	Garde les y de A

AMR — Restrict & Prolong

Inter-level transfers use 2:1 coordinate mapping — restrict averages 4 fine cells, prolong interpolates with gradient correction.

Coordinate Mapping (2:1)



Restrict (Fine → Coarse)

```
coarse[x,y] = 0.25 * (
    fine[2x,  2y ] + fine[2x+1, 2y ] +
    fine[2x,  2y+1] + fine[2x+1, 2y+1]
);
```

Average of 4 fine cells → 1 coarse cell

Floor/Ceil for Negative Coords

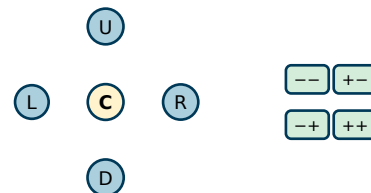
```
// Handles negative coordinates correctly
floor_div2(x) = (x>=0) ? x/2 : (x-1)/2;
ceil_div2(x)  = (x>=0) ? (x+1)/2 : x/2;
// Example: floor_div2(-3) = -2, not -1
```

Prolong (Coarse → Fine)

1. Injection (simple copy)

```
fine[x,y] = coarse[x/2, y/2];
```

2. Linear Prediction (with gradients)



```
grad_x = 0.125 * (R - L);
grad_y = 0.125 * (U - D);
sign_x = (x%2==0) ? -1 : +1;
sign_y = (y%2==0) ? -1 : +1;
fine = C + sign_x*grad_x + sign_y*grad_y;
```

O(1) per cell with pre-computed mapping

Field Operations

Field operations include algebra, stencil application, and geometry-aware transforms (threshold, remap) for AMR workflows.

Basic Operations

```
// Algebra & reductions
field_add_device(a, b, result);
T dot = field_dot_device(a, b);

// 5-point stencil (W, C, E, S, N)
apply_csr_stencil_on_set_device(
    dst, src, region,
    KOKKOS_LAMBDA(CsrStencilPoint p) {
        return 0.25 * (p.west + p.east
                      + p.south + p.north);
    });
```

AMR: Restrict & Prolong

```
// Fine → Coarse (average 4 cells)
restrict_field_device(fine, coarse);

// Coarse → Fine (interpolation)
prolong_field_device(coarse, fine);
```

Threshold: Field → Geometry

```
// Select cells where |value| > epsilon
IntervalSet2DDevice active =
    threshold_field(field, epsilon);
// Use case: detect shock, refine there
```

Remap: Change Geometry

```
// Project src onto dst geometry
// (overlap → copy, else → default)
remap_field_device(src, dst, default_val);
```

src geo: 
dst geo: 
result: 
copy ↑ ↑ default

V. Demo

Mach2 Cylinder — Problem & Setup

Description

2D compressible flow simulation:

- **Mach 2** supersonic around a cylinder
- 1st order Godunov scheme + Rusanov flux
- **Dynamic AMR**: 4 levels

Sparse: computation only on fluid cells!

Subsetix Setup

Geometry Construction

```
auto fluid = set_difference_device(  
    make_box_device(domain),  
    make_disk_device(cylinder), ctx);
```

Conserved Fields (ρ , ρu , ρv , E)

```
ConservedFields U_coarse(coarse_geo);  
ConservedFields U_fine(fine_geo);
```

Flux on Coarse Level (excluding fine)

```
auto coarse_active = set_difference_device(  
    coarse_geo, fine_projection, ctx);  
apply_stencil_on_set_device(  
    flux, U_coarse, coarse_active, Flux{});
```


Mach2 Cylinder — AMR Operations

1. Gradient Indicator (detect shock)

```
apply_csr_stencil_on_set_device(  
    indicator, U_fine.rho, interior, GradStencil{});
```

2. Threshold → Refinement Mask

```
auto mask = threshold_field(indicator, thresh);
```

3. Expand → Guard Zones

```
expand_device(mask, guard_size, guard_size, fine_geo, ctx);
```

4. Inter-level Transfers

```
restrict_fine_to_coarse(U_coarse, U_fine, overlap);  
prolong_guard_from_coarse(U_fine, guard_region, U_coarse);
```

5. Intersect → Overlap Region

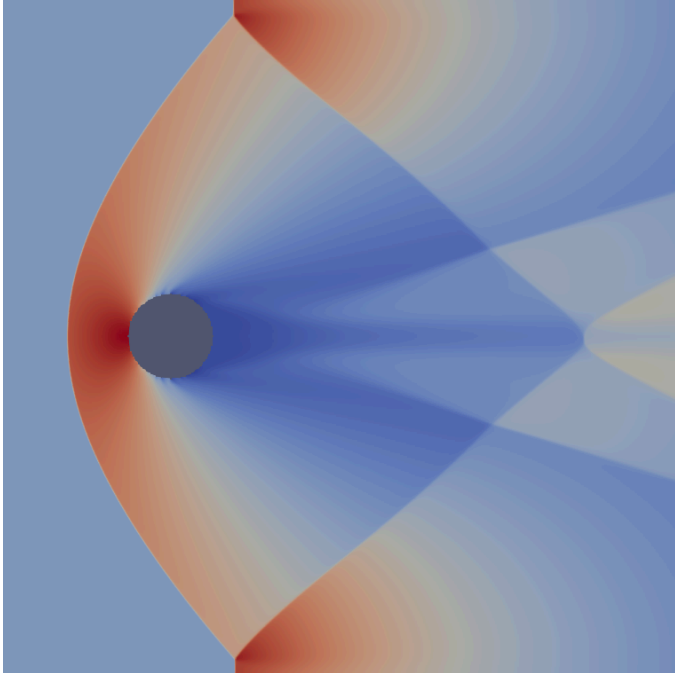
```
auto overlap = intersect_device(  
    coarse_geo, fine_geo, ctx);
```

6. Union → Merge Geometries

```
auto all_refined = union_device(  
    mask_level1, mask_level2, ctx);
```

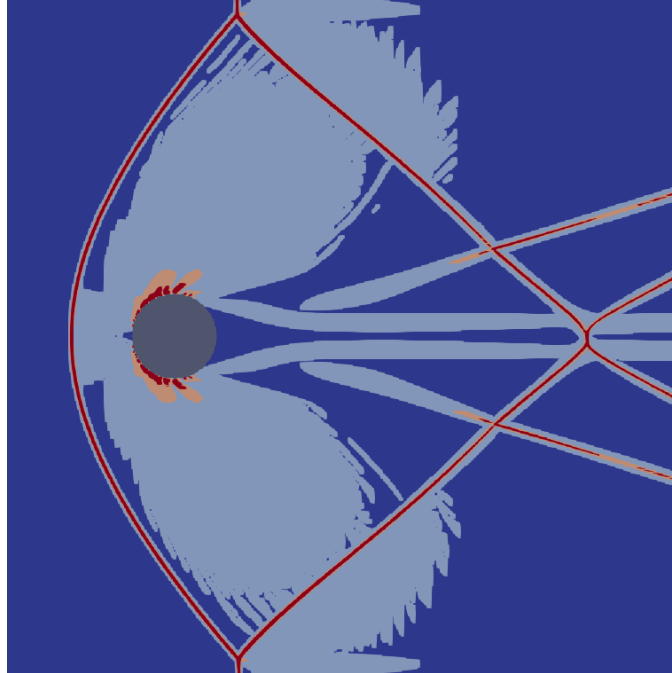
Mach2 Cylinder — Visual Results

Density Field



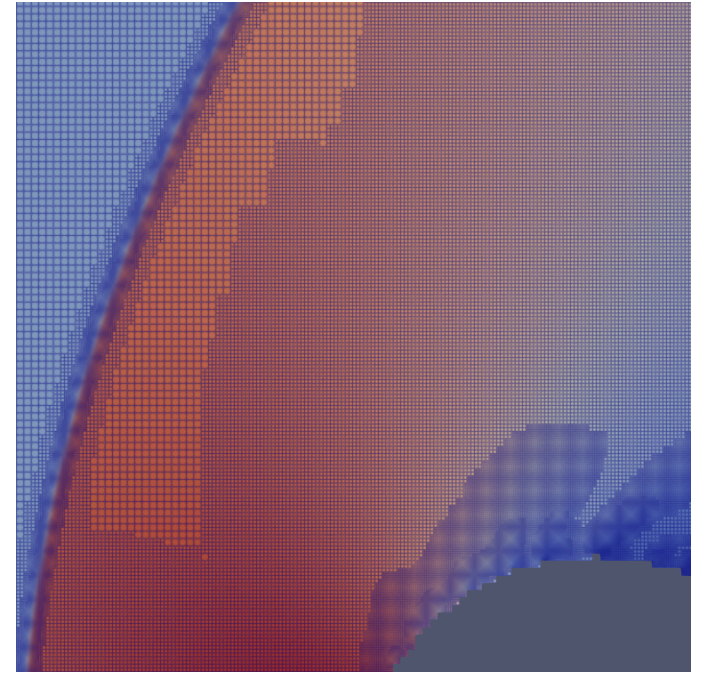
Bow shock in front of cylinder
Colormap: blue (low) → red (high)

AMR Levels



Automatic refinement zones
near the shock front

Mesh Zoom



Multi-level AMR resolution
near the bow shock

4 AMR levels (9-12) — Automatic refinement based on density gradient

Thank You!

Questions?

Key Points

- CSR interval representation
- Count-Scan-Fill pattern
- Kokkos parallelism (CPU/GPU)
- Workspace for memory reuse
- Multi-level AMR (Mach2)

Contact

Sébastien DUBOIS
HPC@Maths Team

Code: `include/subsetix/`
Demo: `examples/mach2_cylinder/`

Appendices

Appendix A: Project Evolution

Implementation History

Version	Description	Performance	Status
v1	CPU only, Sparse CSR + Workspaces First sequential implementation	Faster than baseline	✓ Stable
v2	Multithreaded Tiled Sparse CSR OpenMP and TBB backends Tiling for locality	Excellent on large mesh	⚠ Complex Likely bugs
v3	CUDA only GPU set algebra Proof of concept	Fastest	✓ PoC validated
v4	Kokkos (current version) Non-tiled Sparse CSR OpenMP + CUDA portability	Slower than v2/v3	✓✓ Reliable Verified

Lessons Learned

- **Tiling** improves locality but greatly increases complexity
- Native CUDA faster but less portable
- Kokkos = best **reliability/portability** tradeoff

Final Choice: Kokkos

- **Single** code for CPU and GPU
- Simplified maintenance
- Easy testing and verification
- Active ecosystem (Sandia, Trilinos)

Appendix B: Why Kokkos?

Comparison with Native CUDA

Aspect	CUDA	Kokkos
Portability	NVIDIA only	Multi-vendor
Syntax	<<<>>>	C++ standard
Memory	cudaMalloc	View<T*>
CPU Debug	Difficult	Easy (Serial)
Maintenance	Duplicated code	Single code
Performance	Optimal	90-95%

Supported Backends

- **OpenMP**: CPU multi-thread
- **CUDA**: NVIDIA GPU
- **HIP**: AMD GPU
- **SYCL**: Intel GPU
- **Serial**: debug and tests

Benefits for This Project

1. Faster Development

Debug on CPU (Serial/OpenMP), deploy on GPU

2. Reliable Tests

Same code tested on CPU and GPU
No hidden “GPU-only” bugs

3. Std Algorithms

transform, reduce, scan, copy...
Familiar API, platform-optimized

4. Ecosystem

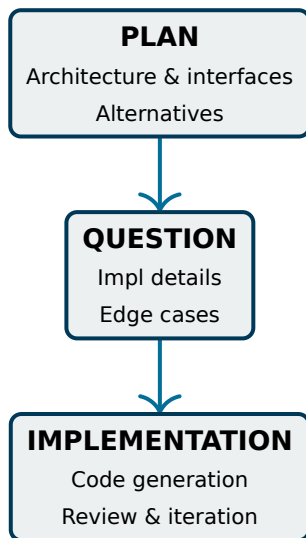
Trilinos, ArborX, Cabana...
Sandia National Labs support

Appendix C: Development Methodology

Models Used

- **Claude 4.5 Opus** (Anthropic)
- **Claude 4.5 Sonnet** (Anthropic)
- **gpt-5.1-codex-max** (OpenAI)

Work Pattern



Observed Benefits

- **Rapid exploration** of designs
- Generated inline documentation
- Automatically suggested tests
- Assisted refactoring

Points of Attention

- Systematic code verification
- LLMs can hallucinate APIs
- Always compile and test
- Maintain **architectural control**

LLM = **accelerator**, not replacement
Human expertise remains essential