Subsetix: Sparse 2D Geometry on GPU

From Set Algebra to AMR Simulation

Sébastien DUBOIS

HPC@Maths Team

December 2025

Outline

I. Context: GPU & Kokkos

GPU Architecture — Massively Parallel

Execution Hierarchy

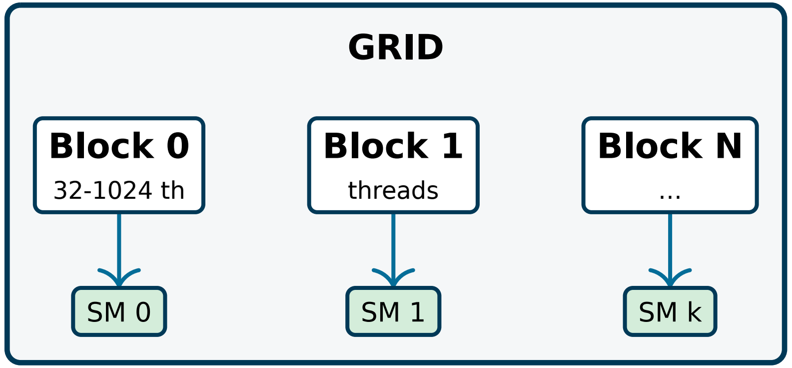
GPU → **SM** ×N → **Warp** ×64 → **Thread** ×32

- **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	**80 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

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

// Copy Host → Device
cudaMemcpy(d_data, h_data,
  n*8, HostToDevice);

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

// Copy Device → Host
cudaMemcpy(h_data, d_data,
  n*8, DeviceToHost);

// Free
cudaFree(d_data);
```

Kokkos

```
// Allocation + auto mirror
View<double*> data("d", n);
auto h_data = create_mirror_view(data);

// Copy Host → Device
deep_copy(data, h_data);

// Parallel (CPU or GPU)
parallel_for(n, KOKKOS_LAMBDA(int i){
  data(i) = compute(i);
});

// Copy Device → Host
deep_copy(h_data, data);

// Automatic cleanup (RAII)
```

Single source code → compiles for OpenMP, CUDA, HIP, SYCL, Serial — specializable if needed

II. Sparse Representation

Example: 2D Sparse Mesh with Intervals

"Smiley" Geometry :-)

```
      Y
      9│ . . . . . . . . . .    (empty)
      8│ . . . . . . . . . .    (empty)
      7│ . . ▓ ▓ . . ▓ ▓ . .    EYES
      6│ . . ▓ ▓ . . ▓ ▓ . .    EYES
      5│ . . . . . . . . . .    (HOLE)
      4│ . . . . . . . . . .    (HOLE)
      3│ . ▓ ▓ . . . . ▓ ▓ .    SMILE
      2│ . . ▓ ▓ . . ▓ ▓ . .    SMILE
      1│ . . . ▓ ▓ ▓ ▓ . . .    SMILE
      0│ . . . . . . . . . .    (empty)
       └─────────────────── X
         0 1 2 3 4 5 6 7 8 9
```

CSR Representation

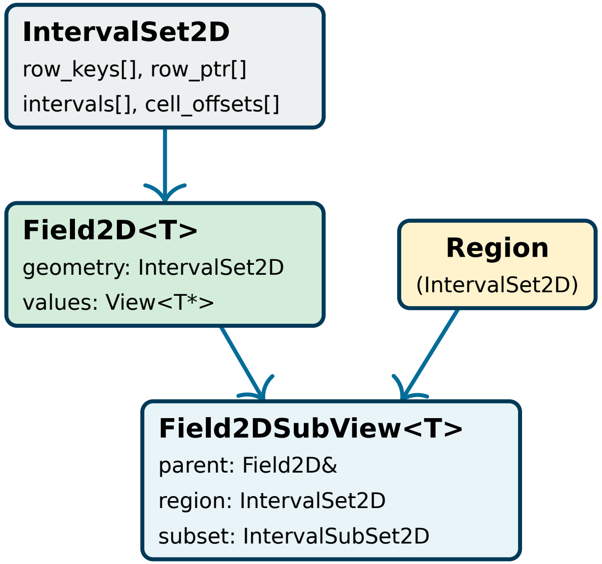
```
// 5 rows, HOLE Y=4,5
row_keys = [1, 2, 3, 6, 7]  // skips 4,5!
num_rows = 5

// 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



IntervalSubSet2D

```
struct IntervalSubSet2D {
  IntervalSet2D parent;      // ref
  interval_indices[];        // which intervals
  x_begin[], x_end[];        // restricted range
  row_indices[];             // Y coords
};
```

SubView: Lazy Intersection

```
// Region = any IntervalSet2D
IntervalSet2DDevice left_bc = make_box_device({0,2,0,ny});
Field2DSubViewDevice<T> sub = make_subview(field, left_bc);

// First op: computes field.geo ∩ region
fill_subview_device(sub, T_inlet, &ctx);

// Time loop: reuses cached intersection
for (int step = 0; step < nsteps; ++step) {
  fill_subview_device(sub, T_inlet);  // fast
}
```

SubView Operations

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

Lazy: intersection computed on first use
Cached: reused for subsequent operations

IntervalSet2D — Complete CSR Structure

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;
};
```

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

Template MemorySpace: Device or Host

Field2D — Field on Sparse Geometry

Definition

Associates a **value** with each sparse cell

```
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

```
// O(1) - when interval index known
T val = field.at(interval_idx, x);

// O(log R + log I) - by coordinates
// (binary search on Y, then X)
bool ok = accessor.try_get(x, y, val);
```

Usage

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

SubSet — Targeted Region Operations

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);

// Iteration: O(1) access per entry
for (e = 0; e < num_entries; ++e) {
  int iv = interval_indices[e];
  for (x = x_begin[e]; x < x_end[e]; ++x)
    field.at(iv, x) = ...;  // O(1)
}
```

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

Field2DSubView — View on Field + Region

Structure

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

Lazy Pattern

```
// 1. Create (no computation)
auto 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 ∩ region

// 3. Next ops reuse cached subset
scale_subview_device(sub, 2.0);  // fast!
fill_subview_device(sub, 1.0);   // fast!
```

Memory Mapping

```
Geometry:   [==A==]    [==B==]    [==C==]
                ↓          ↓          ↓
values[]:  [░░███|░░░░░████░|██░░░░░░]
             ↑          ↑       ↑
          entry 0    entry 1  entry 2
```

░ = skipped █ = accessed by SubSet

Access Formula

`values[ offset[idx] + (x - interval.begin) ]`

O(1) per cell — no coordinate lookup

Workspace & AMR Support

UnifiedCsrWorkspace

Pool of reusable buffers

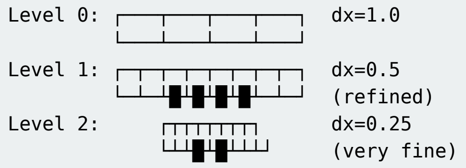
```
struct UnifiedCsrWorkspace {
  View<int*> int_bufs_[5];
  View<size_t*> size_t_bufs_[2];
  View<RowKey2D*> row_key_bufs_[2];
  View<Interval*> interval_buf_0;

  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

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);
  }
};
```

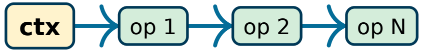


IV. Algorithms

Set Algebra — Binary Operations

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
};
```



Same ctx reused → **zero allocations** after warmup

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 ∪ B
set_union_device(A, B, set1, ctx);

// Compute: set2 = set1 \ 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

Phase 1: Row Mapping

```
A.row_keys:  [y=2, y=5, y=8]
B.row_keys:  [y=3, y=5, y=7, y=8]
                   ↓         ↓
Binary search: A[1] ↔ B[1], A[2] ↔ B[3]
```

```
Output rows:  [y=5, y=8]
row_index_a:  [ 1,   2 ]
row_index_b:  [ 1,   3 ]
```

Phase 2: Interval Merge (per row)

```
A: [===]     [=======]
     2   6       10    18

B:     [=====] [===]
        4    9  12  16

Sweep → max(begin), min(end):
  [4,6] ∩  → output [4,6]
  [10,18] ∩ [12,16] → output [12,16]
```

O(n+m) per row — linear merge

GPU Pattern: Count → Scan → Fill

1. COUNT (parallel per row)

```
row_counts[i] = count_intersection(
    A.intervals[begin_a..end_a],
    B.intervals[begin_b..end_b]);
```

2. SCAN (exclusive prefix sum)

```
row_ptr_out = exclusive_scan(row_counts)
// row_ptr_out[i] = where row i starts
```

3. FILL (parallel per row)

```
fill_intersection(
    A.intervals, B.intervals,
    out.intervals, row_ptr_out[i]);
```

Same pattern for ∪, , ⊕

Field Operations

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
```

Mathematical Morphology & AMR

Dilation / Erosion

```
// N-way union with ±radius offset
row_n_way_union_impl(rows[], radius, out)

// N-way intersection with shrink
row_n_way_intersection_impl(rows[], r, out)
```

```
Original:  ░░░░████████░░░░░░░░
Dilate(1): ░░░█████████░░░░░░░░  (+1 sides)
Erode(1):  ░░░░░██████░░░░░░░░░  (-1 sides)
```

2D Extension

- Consider rows y-r to y+r
- Merge with N-way operation
- Implicit structuring element (square)

AMR Operations

```
// Coarsening: fine → coarse
build_row_coarsen_mapping(fine, ws)
// y_coarse = y_fine / 2, merge X

// Refinement: coarse → fine
refine_level_up_device(coarse, ws)
// [a,b) → [2a, 2b), double Y
```

```
Fine (level 1):          Coarse (level 0):
Y=3: ████████            Y=1: ████████
Y=2: ████████      →          (merge Y=2,3)
Y=1: ░░░░████            Y=0: ░░░░████
Y=0: ░░░░████                 (merge Y=0,1)
```

Field Transfer

```
// Projection fine → coarse (average)
// Prolongation coarse → fine (interp)
build_amr_interval_mapping(coarse, fine)
```

V. Demo

Mach2 Cylinder — Multi-Level AMR Simulation

Description

2D compressible flow simulation:

- **Mach 2** supersonic around a cylinder
- 1st order Godunov scheme + Rusanov flux
- **Dynamic AMR**: up to 6 levels

Subsetix Usage

```
// Fluid geometry = domain - obstacle
auto fluid = set_difference_device(
  make_box_device(domain),
  make_disk_device(cylinder),
  ctx);

// Conserved fields (ρ, ρu, ρv, E)
Field2DDevice<Real> rho(fluid);
Field2DDevice<Real> rhou(fluid);
// ...
```

AMR Architecture

```
┌──────────────────────────────────┐
│  Level 0 (coarse)                │
│                                  │
│  ┌───────────────────────────┐   │
│  │ Flux + Update (dt_coarse) │   │
│  └───────────────────────────┘   │
│                                  │
│        ▲ restrict     │ prolong  │
│  ┌─────┴──────────────▼──────┐   │
│  │  Level 1 (finer)          │   │
│  │  ┌───────────────────┐    │   │
│  │  │ Flux + Update (dt) │   │   │
│  │  └───────────────────┘    │   │
│  │        ▲          │          │
│  │  Level 2 (finest around shock) │
│  └───────────────────────────┘   │
└──────────────────────────────────┘
```

Dynamic Refinement

- Indicator: density gradient
- `expand_device()` for guard zones
- Remeshing every N time steps

Mach2 Cylinder — Results & Visualization

Generated Outputs

```
output/
├── fluid_geometry.vtk
├── obstacle_geometry.vtk
├── level_0_density_0000.vtk
├── level_0_density_0050.vtk
├── level_1_density_0050.vtk
├── level_2_density_0050.vtk
└── ...
```

Execution Command

```
./mach2_cylinder \
  --nx 400 --ny 160 \
  --radius 20 \
  --mach-inlet 2.0 \
  --max-steps 1000 \
  --output-stride 50 \
  --amr
```

Observed Phenomena

- **Bow shock** in front of the cylinder
- Subsonic zone in the wake
- **Von Kármán** vortex street
- Automatic refinement near the shock

Key Technical Points

- CSR stencil: `apply_csr_stencil_on_set_device()`
- Struct-of-Arrays for cache efficiency
- `prolong_guard_from_coarse()`: interpolation
- `restrict_fine_to_coarse()`: conservation
- Multi-level VTK export for ParaView

Sparse: computation only on fluid cells!

Live Demo

Live Demo

Construction

- Box, Disk, Bitmap
- Difference (obstacle)
- CSR display

Operations

- Union / Intersection
- Field algebra
- Stencil

Mach2

- Launch simulation
- ParaView visualization
- AMR in action

Live demonstration...

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

Implementation History

Version	Description	Performance	Status
v1	CPU only, Sparse CSR + Workspaces First sequential implementation	Faster than baseline	✓ Stable
v3	CUDA only GPU set algebra Proof of concept	**Fastest**	✓ PoC validated

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)

Comparison with Native CUDA

Aspect	CUDA	Kokkos

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

Intensive Use of LLMs

Models Used

- **Claude Opus 4** (Anthropic)
- **Claude Sonnet 4** (Anthropic)

Work Pattern

```
1. PLAN
   Architecture and interfaces
   Discussion of alternatives

2. QUESTION
   Implementation details
   Edge cases

3. IMPLEMENTATION
   Code generation
   Review and iteration
```

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

Kokkos

- Website: kokkos.org
- GitHub: github.com/kokkos/kokkos
- Wiki: kokkos.org/kokkos-core-wiki

CUDA

- CUDA Toolkit Documentation
- CUDA C++ Programming Guide

Visualization

- VTK: vtk.org
- ParaView: paraview.org

Mathematical Morphology

- Serra, J. “Image Analysis and Mathematical Morphology” (1982)
- Soille, P. “Morphological Image Analysis” (2003)

Source Code

```
include/subsetix/
├── geometry/      # IntervalSet2D
├── field/         # Field2D
├── csr_ops/       # Algorithms
├── multilevel/    # AMR
└── detail/        # Utilities

examples/mach2_cylinder/
└── mach2_cylinder.cpp  # AMR Demo
```