Subsetix: Sparse 2D Geometry on GPU

From Set Algebra to AMR Simulation

Sébastien DUBOIS

HPC@Maths Team

December 2025

Outline

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

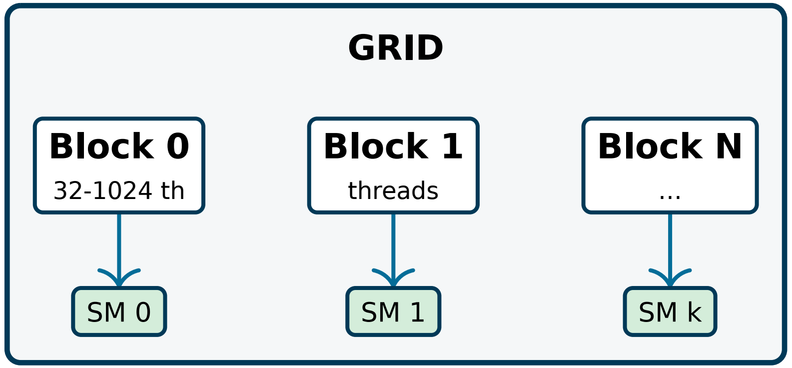
GPU → SM ×N → Warp ×64 → Thread ×32

- **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	**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 :-)

```
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

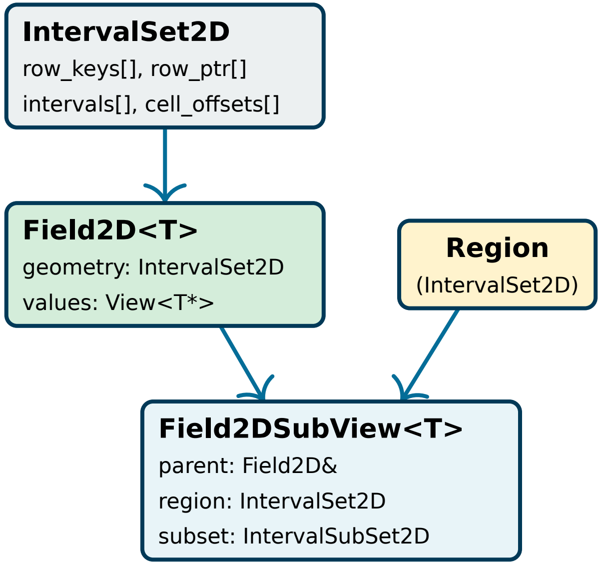
```
// 5 rows, HOLE Y=4,5
row_keys = [1, 2, 3, 6, 7]  // skips 4,5!
num_rows = 5

// 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

```cpp
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

```cpp
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});
  }
};
```

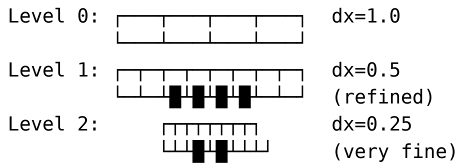
Avoids repeated GPU allocations
Crucial for chained operations

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

MultilevelGeo (AMR)

Multi-resolution grids

```cpp
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 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_{row}) — I = intervals in row

Combined: Cell Lookup

```
bool try_get(Coord x, Coord y, T& out) {
  // Step 1: find row
  int row = find_row_by_y(row_keys, y);
  if (row < 0) return false;

  // Step 2: find interval in row
  int iv = find_interval_by_x(
    intervals, row_ptr[row], row_ptr[row+1], x);
  if (iv < 0) return false;

  // Step 3: compute offset
  out = values[offsets[iv] + (x - intervals[iv].begin)];
  return true;
}
```

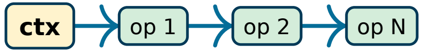
Total: **O(log R + log I)**

Invariant: Data must stay sorted!
(enforced by construction)

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

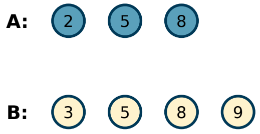
Row Mapping — Why and How

GPU Constraint

1 thread = 1 output row

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

Which Rows Participate?



Op	Output rows	Count
A ∩ B	{5, 8}	2
A ∪ B	{2,3,5,8,9}	5
A \ B	{2}	1

Row Mapping per Operation

Intersection — rows in both A and B

```
// Binary search A rows in B
map[i] = binary_search(B.row_keys, A.row_keys[i]);
// Keep only matched rows (map[i] >= 0)
```

Union — rows in A or B (merge sorted)

```
// 1. Search A→B and B→A
// 2. Flag-Scan-Compact: extract B-only rows
// 3. Interleave A + B-only maintaining order
```

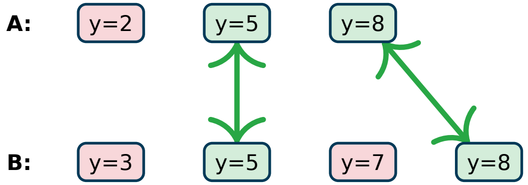
Difference — rows in A (check B for each)

```
// Output = A.row_keys (same structure)
// map[i] = search A.row_keys[i] in B
// If found: must subtract intervals
```

Result: output row list + per-row mapping
→ enables `parallel_for(num_output_rows, ...)`

Intersection — How It Works

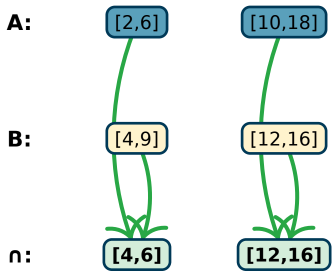
Phase 1: Row Mapping



Binary search: O(log n) per row

Phase 2: Interval Merge (per row)

Row y=5 — 2 intervals each



O(n+m) sweep — `max(begin)`, `min(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 → 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 **∪**, \, ⊕

Stencil Access — Sparse Geometry

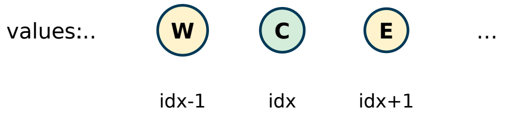
The Problem

On dense grid: `north = values[i + nx]`

On sparse CSR: row y+1 may not exist, or have different intervals!

Horizontal: Easy

Same interval → contiguous in memory



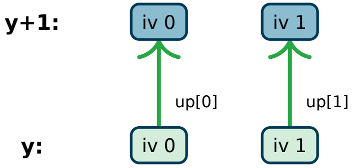
`west = values[idx-1]` **O(1)**

Vertical: Pre-computed Mapping

```
struct VerticalIntervalMapping {
  View<int*> up_interval;
  View<int*> down_interval;
};
```

Built once, maps each interval to its neighbor in y±1

Vertical Mapping Diagram



Condition: same interval count per row

Access North

```
int up_idx = up_interval[center_idx];
Interval iv = intervals[up_idx];
size_t off = offsets[up_idx] + (x - iv.begin);
return values[off];  // O(1)
```

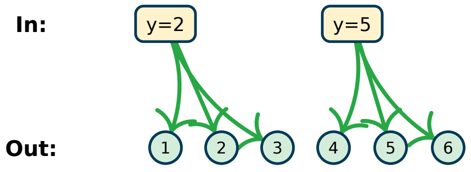
User Interface

```
// Stencil functor receives:
CsrStencilPoint<T> p;
p.center(); p.west(); p.east();
p.north();  p.south();  // All O(1)
```

Expand (Dilation) — How It Works

Phase 1: Row Mapping

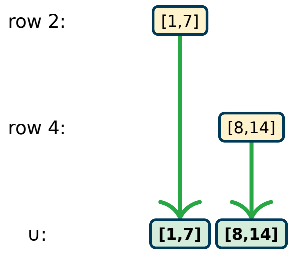
Input row y → output rows [y-ry, y+ry]



ry=1: row 2 → {1,2,3}, row 5 → {4,5,6}

Phase 2: N-way Union (per row)

Output row y=3 — inputs: rows 2,3,4



Expand: [b,e) → [b-rx, e+rx)

Interval Expansion

Original: [4,8]

rx=2: **[2,10]**

`[begin-rx, end+rx)`

GPU Pattern: Count → Scan → Fill

1. COUNT — N-way union count

```
row_counts[i] = n_way_union_count(
    rows[map_start[i]..map_end[i]], rx);
```

2. SCAN

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

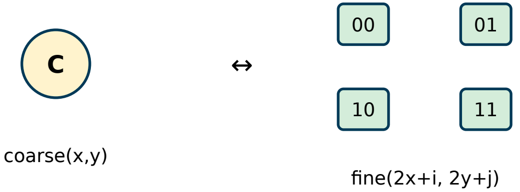
3. FILL — N-way union write

```
n_way_union_fill(
    rows[map_start[i]..map_end[i]], rx,
    out, row_ptr[i]);
```

Shrink: N-way ∩, [b+rx, e-rx)

AMR — Restrict & Prolong

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

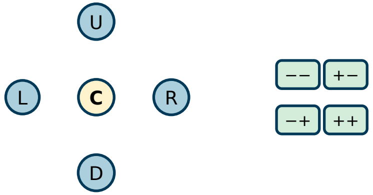
```
// Handles negative coordinates correctly
floor_div2(x) = (x>=0) ? x/2 : (x-1)/2;
ceil_div2(x)  = (x>=0) ? (x+1)/2 : x/2;
// Example: floor_div2(-3) = -2, not -1
```

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

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**: 4 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

```
mach2_cylinder/
├── fluid_geometry.vtk
├── obstacle_geometry.vtk
├── refine_mask_lvl{1,2,3}.vtk
├── fine_geometry_lvl{1,2,3}.vtk
├── step_00001_density.vtk
├── step_00001_l0_density.vtk
├── step_00001_l1_density.vtk
├── step_00001_mach.vtk
├── step_00001_pressure.vtk
└── ...
```

Execution Command

```
./mach2_cylinder \
  --nx 400 --ny 160 \
  --radius 20 \
  --mach-inlet 2.0 \
  --max-steps 5000 \
  --output-stride 50 \
  --amr --amr-levels 4
```

Observed Phenomena

- **Bow shock** in front of the cylinder
- Density/pressure gradient captured
- AMR refinement follows the shock

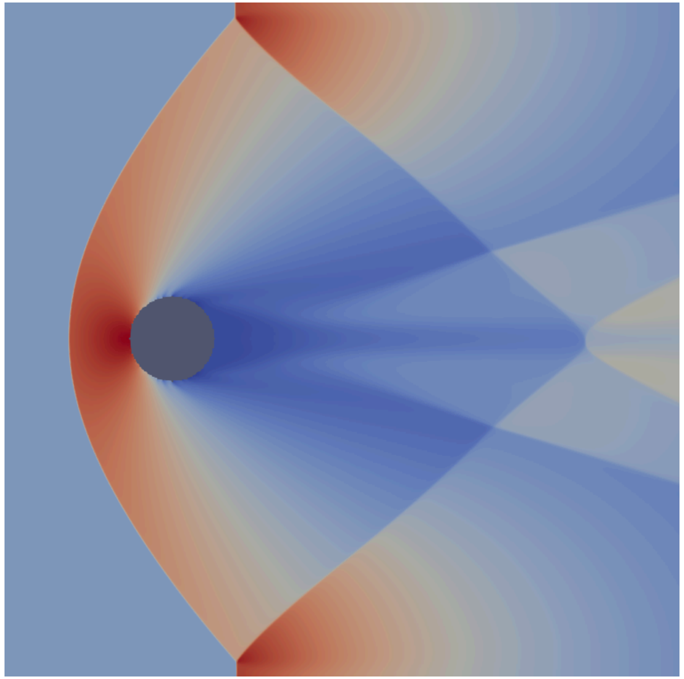
Key Technical Points

- 1st order Godunov + Rusanov flux
- Struct-of-Arrays: `ConservedFields` (ρ, ρu, ρv, E)
- `threshold_field()` → detect shock gradient
- `expand_device()` → guard cells around refined zone
- `restrict_fine_to_coarse()` / `prolong_guard_from_coarse()`
- `write_multilevel_field_vtk()` for ParaView

Sparse: computation only on fluid cells!

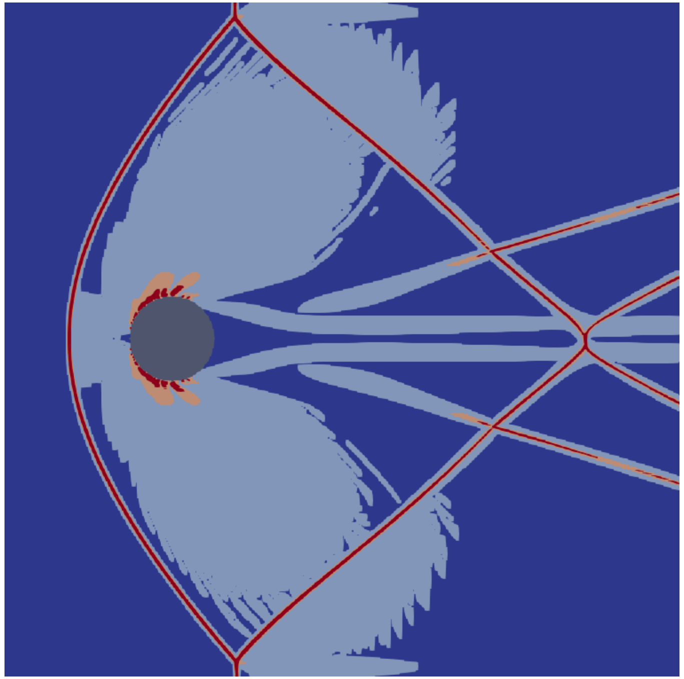
Mach2 Cylinder — Visual Results

Density Field



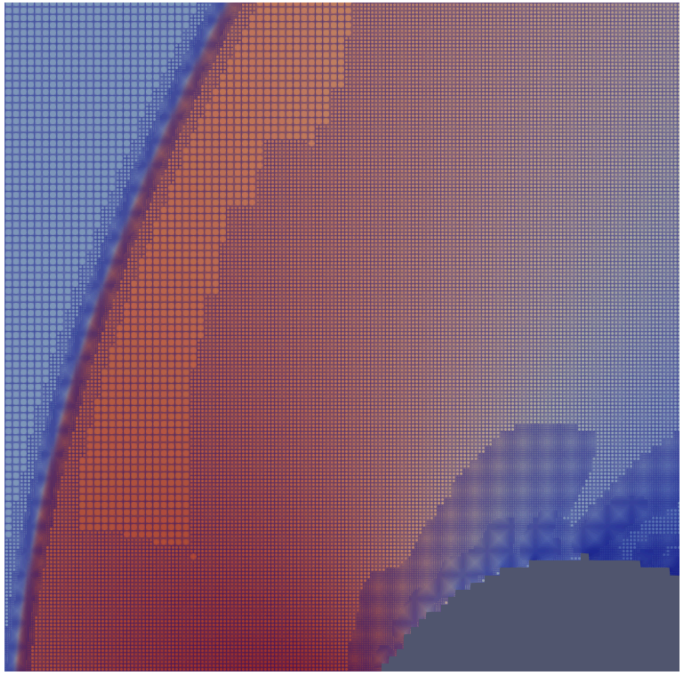
Bow shock in front of cylinder
Colormap: blue (low) → red (high)

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

PLAN
Architecture & interfaces
Alternatives

↓

QUESTION
Impl details
Edge cases

↓

IMPLEMENTATION
Code generation
Review & 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