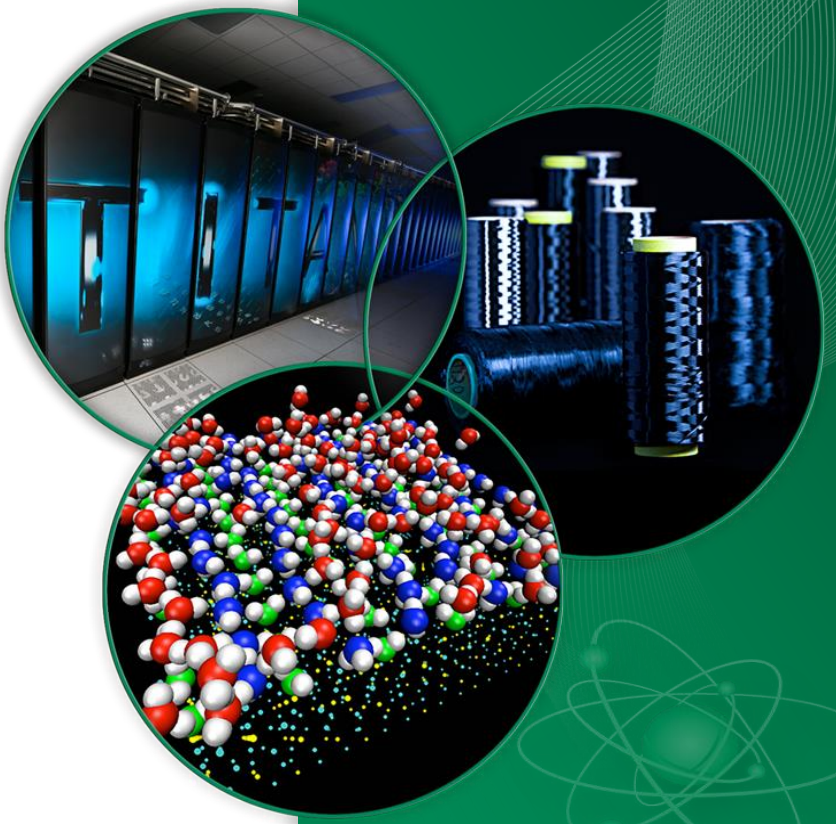
Productive Programming With Descriptive Data ~ Efficient Mesh-Based Algorithm Development in EAVL

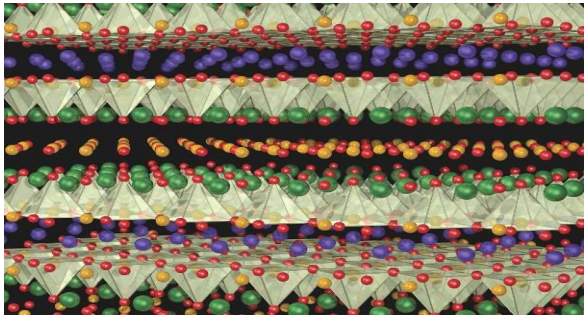


Jeremy Meredith

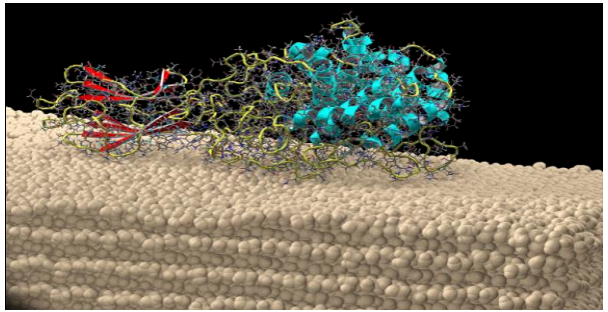
NVIDIA GTC

March 2014

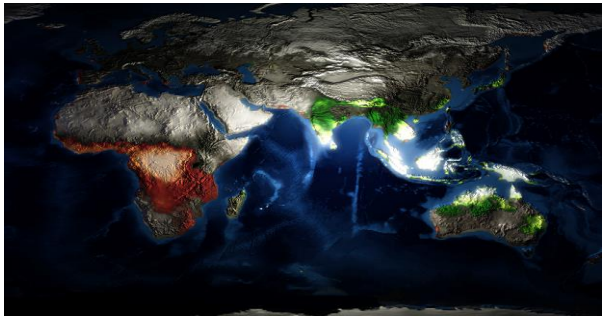
Scientific Discovery through Simulation



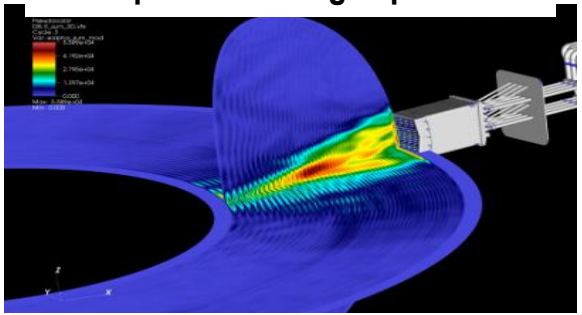
Resolved decades-long controversy about modeling physics of high temperature superconducting cuprates



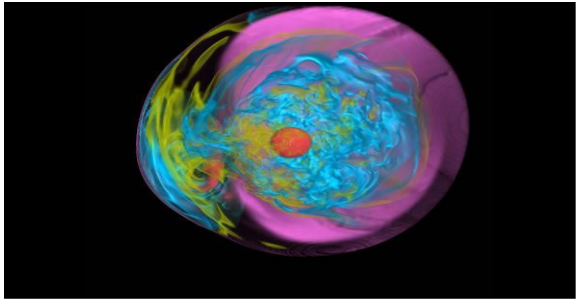
New insights into protein structure and function leading to better understanding of cellulose-to-ethanol conversion



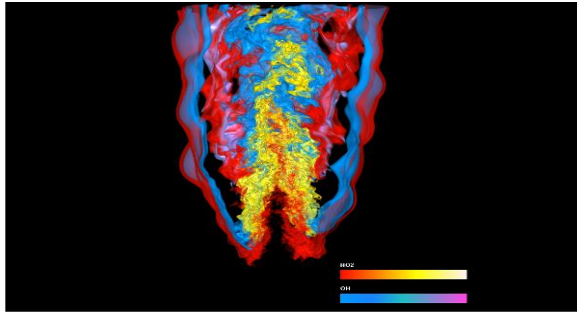
Addition of vegetation models in climate code for global, dynamic CO_2 exploration



First fully 3D plasma simulations shed new light on engineering superheated ionic gas in ITER



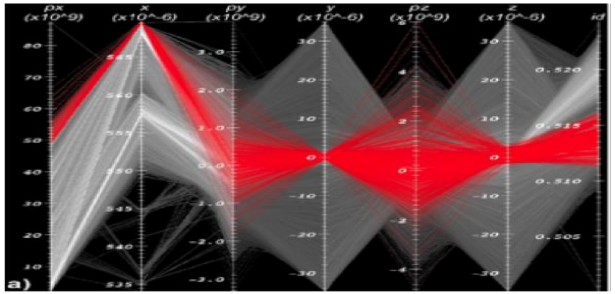
Fundamental instability of supernova shocks discovered directly through simulation

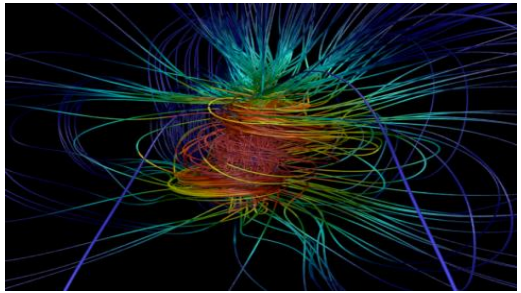


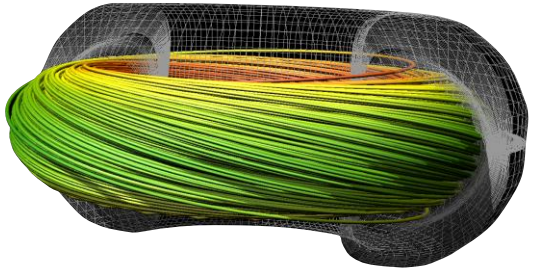
First 3-D simulation of flame that resolves chemical composition, temperature, and flow

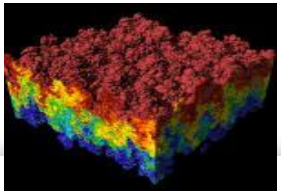
Scientific Visualization and Analysis

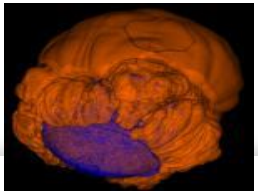
- Extract meaning from scientific data
- Wide variety of scientific domains
  - General purpose libraries: union of domain data models

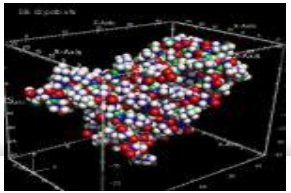


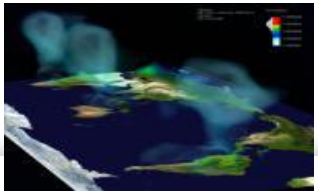


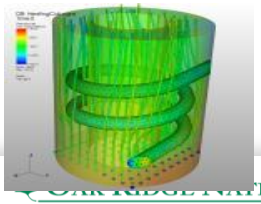


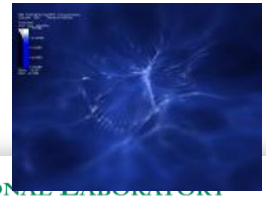












DOE Path to Exascale: I/O vs FLOPS,RAM

Machine	Year	FLOPS Writable to Disk	Whole-System Checkpoint
ASCI Red	1997	0.075%	300 sec
ASCI Blue Pac.	1998	0.041%	400 sec
ASCI White	2001	0.026%	660 sec
ASCI Red Storm	2004	0.035%	660 sec
ASCI Purple	2005	0.025%	500 sec
NCCS XT4	2007	0.004%	1400 sec
Roadrunner	2008	0.005%	480 sec
NCCS XT5	2008	0.005%	1250 sec
ASC Sequoia	2012	0.001%	3200 sec

Relative I/O rates are dropping

- Simulations writing less data
- In situ visualization and analysis
  - Do vis/analysis while the simulation runs
  - Can prevent scientific data loss

DOE Path to Exascale: FLOPS vs RAM

Machine	Year	RAM Bytes / FLOPS
ASCI Red	1997	0.90
ASCI Blue Pacific	1998	1.62
ASCI White	2001	0.49
ASCI Red Storm	2004	0.92
ASCI Purple	2005	0.50
NCCS XT4	2007	0.24
Roadrunner	2008	0.08
NCCS XT5	2008	0.25
ASC Sequoia	2012	0.07

- Memory is precious
  - Available RAM is growing more slowly than FLOPS
  - In situ analysis exacerbates this problem
    - Must share resources (code + data) between simulation and analysis
    - Concurrent analysis helps with use of network communication

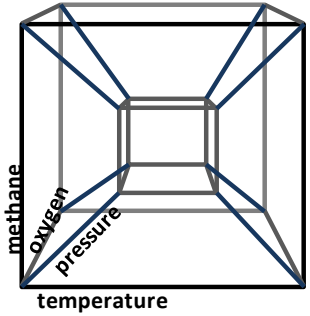
DOE Path to Exascale: Concurrency

Predicted Exascale Machines	
Node Concurrency	1,000 - 10,000
Number of Nodes	1,000,000 - 100,000
Total Concurrency	1 billion

- Massive concurrency *across* nodes
  - In situ analysis codes commensurate with simulation codes' parallelism
- Massive concurrency *within* nodes
  - Thread- and data-level parallelism like GPUs in use today
  - Accelerators with heterogeneous memory spaces

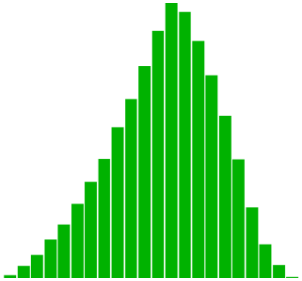
EAVL: Extreme-scale Analysis and Visualization Library

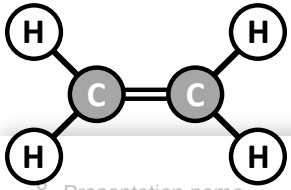
EAVL: Extreme-scale Analysis and Visualization Library

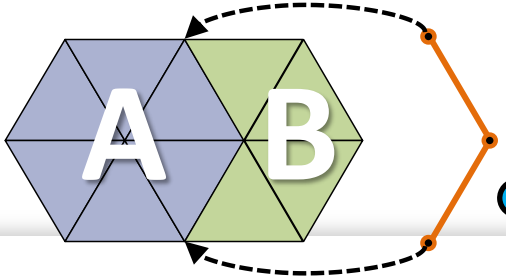


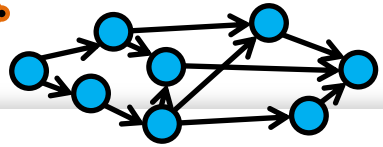
Target approaching hardware/software ecosystem:

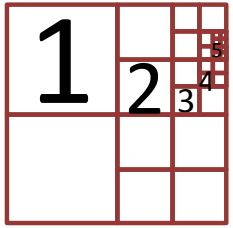
- Update traditional data model to handle modern simulations
- Leverage new data/execution models to achieve efficiency
- Explore means for simulations to enable in situ analysis
- Enable developers to realize efficiency gains through productive programming constructs

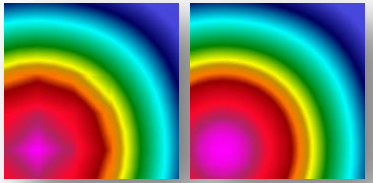








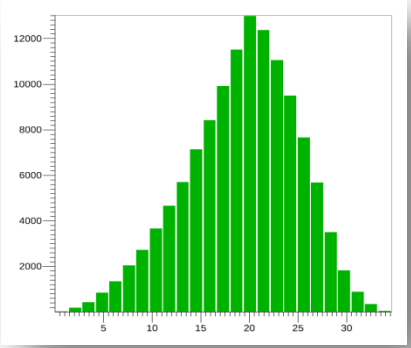


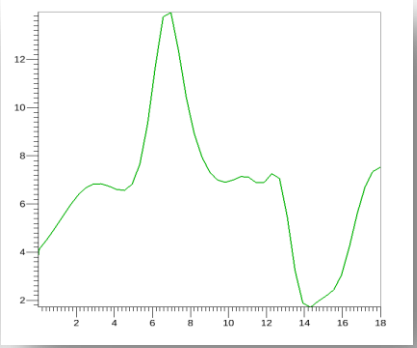


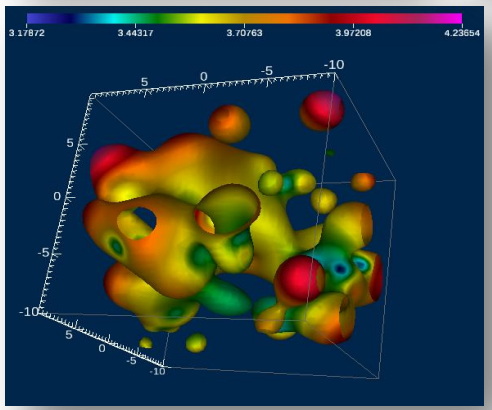
Lightweight + Standalone

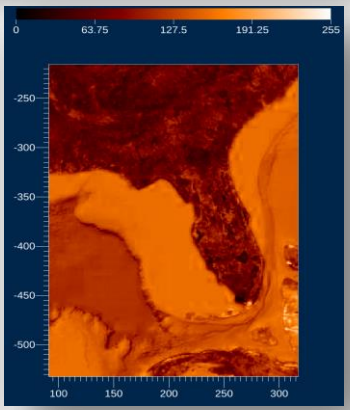
- Small footprint
- <u>No mandatory dependencies</u>
- Header-only 1D, 2D, 3D rendering with annotations
- Optional MPI, CUDA support
- Optional file readers

  (adios, bov, chimera, curve, madness, png, netcdf, pdb, pixie, silo, vtk)

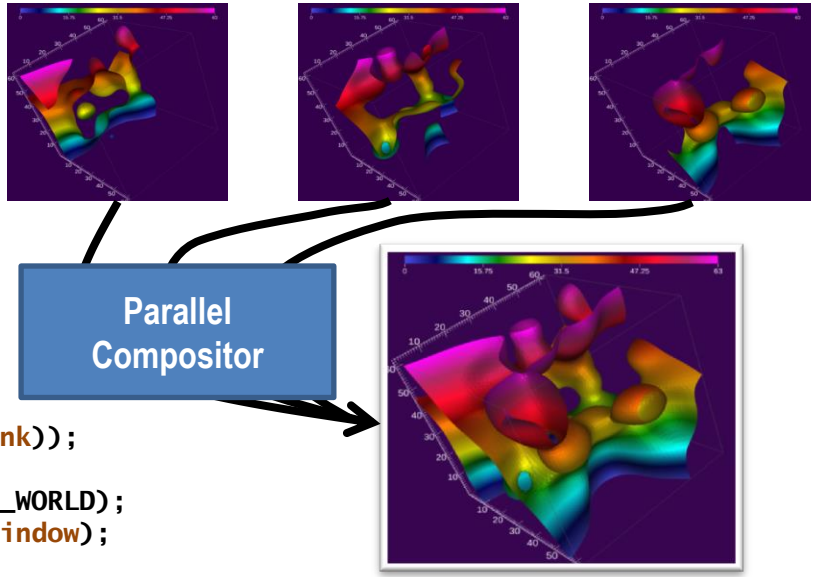








Scalable System Support




```
void main()
{
    int mpirank, nprocs;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    eavlImporter *importer =
        new eavlNetCDFDecomposingImporter(nprocs, "file.nc");
    eavlDataSet *dataset = importer->GetMesh("mesh", mpirank);
    dataset->AddField(importer->GetField("pressure", "mesh", mpirank));

    eavl3DWindow *window = new eavl3DParallelOSMesaWindow(MPI_COMM_WORLD);
    eavlScene *scene = new eavl3DParallelGLScene(MPI_COMM_WORLD, window);
    window->SetScene(scene);
    window->Resize(WIDTH,HEIGHT);

    scene->AddRendering(new eavlPseudocolorRenderer(dataset, “hot_to_cold", “pressure"));
    scene->ResetView();

    window->Paint();
    if (mpirank == 0)
       window->SavePNMImage(“result.pnm");
}
```

Filter Example

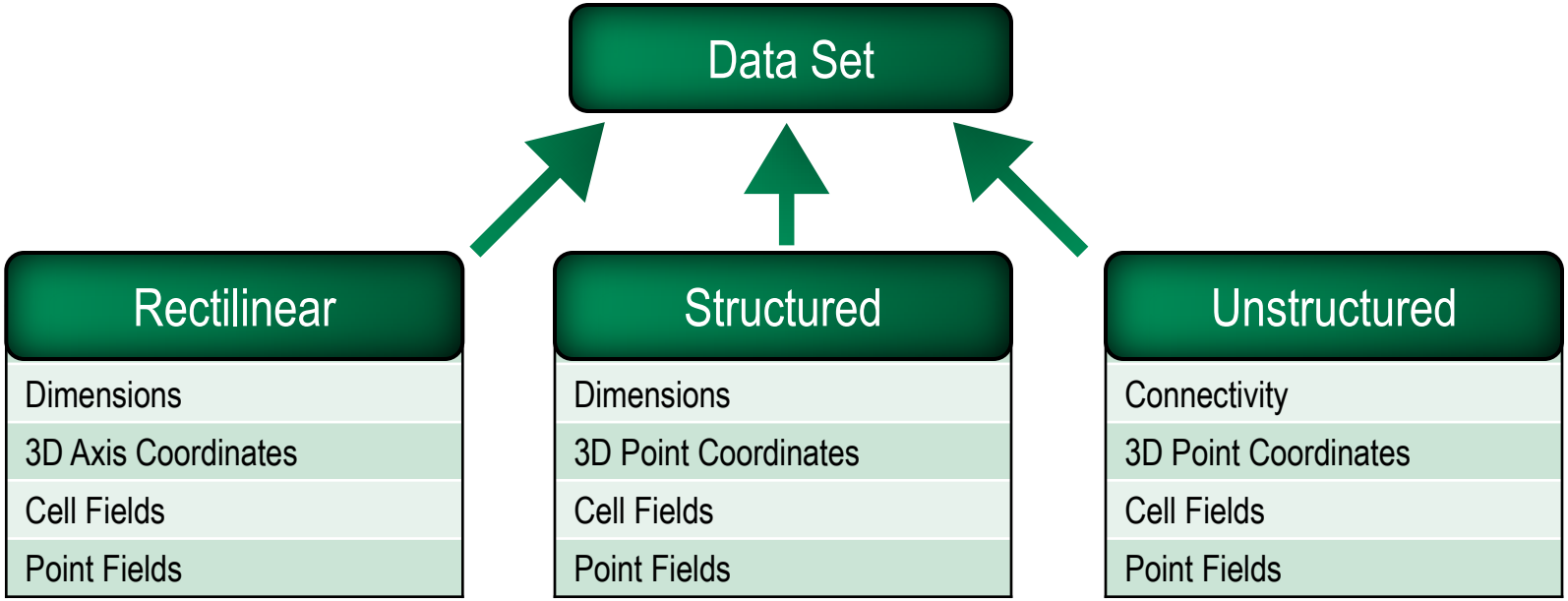
```
#include "eavlImporterFactory.h"
#include "eavlScalarBinFilter.h"
#include "eavlExecutor.h"

int main(int argc, char *argv[])
{
    eavlExecutor::SetExecutionMode(eavlExecutor::ForceGPU);
    eavlInitializeGPU();

    // assume first mesh, single (or first) domain
    int meshindex = 0;
    int domainindex = 0;

    // read the data file (argv[1]) and the given field (argv[2])
    eavlImporter *importer = eavlImporterFactory::GetImporterForFile(argv[1]);
    string meshname = importer->GetMeshList()[meshindex];
    eavlDataSet *data = importer->GetMesh(meshname, domainindex);
    data->AddField(importer->GetField(argv[2], meshname, domainindex));

    // bin given scalar field (argv[2])
    eavlScalarBinFilter *scalarbin = new eavlScalarBinFilter();
    scalarbin->SetInput(data);
    scalarbin->SetField(argv[2]);
    scalarbin->Execute();

    eavlDataSet *result = scalarbin->GetOutput();

    // export the result to a VTK file
    ofstream out("test.vtk");
    eavlVTKExporter exporter(data, 0);
    exporter.Export(out);
    out.close();

    return 0;
}
```

Data Set Creation Example

```
void GenerateRectXY(int ni, int nj)
{
    eavlDataSet *data = new eavlDataSet();
    // set the number of points
    data->SetNumPoints(ni*nj);
    eavlRegularStructure reg;
    reg.SetNodeDimension2D(ni, nj);
    // set the logical structure
    eavlLogicalStructure *log = new eavlLogicalStructureRegular(reg.dimension, reg);
    data->SetLogicalStructure(log);
    // create the coordinate axes
    eavlFloatArray *x = new eavlFloatArray("x", 1, ni);
    eavlFloatArray *y = new eavlFloatArray("y", 1, nj);
    for (int i=0; i<ni; ++i)
       x->SetValue(i, 100 + 10*i);
    for (int j=0; j<nj; ++j)
       y->SetValue(j, 200 + 15*j);
    // add the coordinate axis arrays as linear fields on logical dims
    data->AddField(new eavlField(1, x, eavlField::ASSOC_LOGICALDIM, 0));
    data->AddField(new eavlField(1, y, eavlField::ASSOC_LOGICALDIM, 1));
    // set the coordinates
    eavlCoordinates *coords = new eavlCoordinatesCartesian(log, eavlCoordinatesCartesian::X,
                                                               eavlCoordinatesCartesian::Y);
    coords->SetAxis(0, new eavlCoordinateAxisField("x"));
    coords->SetAxis(1, new eavlCoordinateAxisField("y"));
    data->AddCoordinateSystem(coords);
    // create a cell set implicitly covering the entire regular structure
    eavlCellSet *cells = new eavlCellSetAllStructured("cells", reg);
    data->AddCellSet(cells);
    // print the result
    data->PrintSummary(cout);
}
```

Advanced, Flexible Data Model

A “Traditional” Data Set Model



Gaps in Current Data Models (e.g. VTK)

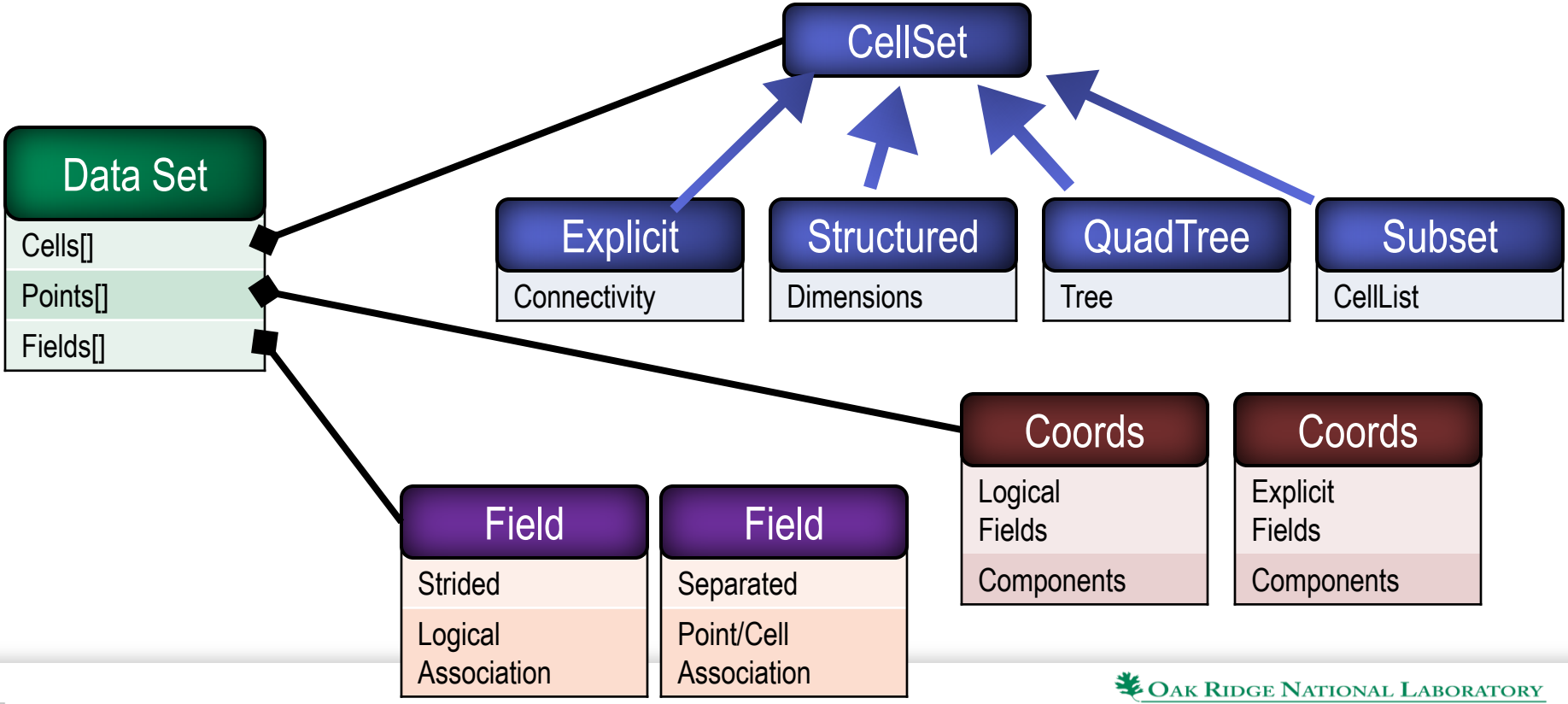
Cells	Coordinates	Point Arrangement		
		Explicit	Logical	Implicit
Structured	Strided	Structured Grid	?	n/a
	Separated	?	Rectilinear Grid	Image Data
Unstructured	Strided	Unstructured Grid	?	?
	Separated	?	?	?

Arbitrary Compositions for Flexibility

Cells	Coordinates	Point Arrangement		
		Explicit	Logical	Implicit
Structured	Strided	✓	✓	✓
	Separated	✓	✓	✓
Unstructured	Strided	✓	✓	✓
	Separated	✓	✓	✓

EAVL Data Set

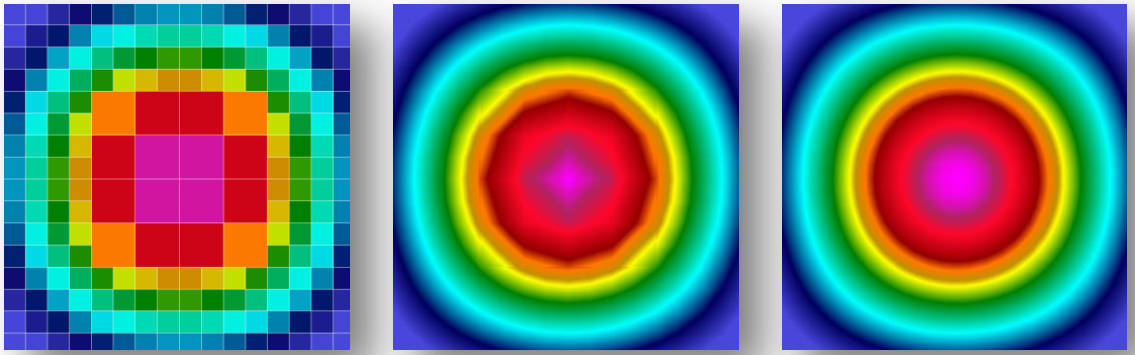
The EAVL Data Set Model



Data Model Gaps Addressed in EAVL

- hybrid mesh types
- 1D/2D coordinate systems
- 4D and higher dimensional data
- non-physical data, e.g. graphs
- face and edge data
- multiple groups of cells in one mesh
  - e.g. subsets, external face sets
- mixed topology meshes
  - e.g. molecules, nodesets, sidesets, embedded surfaces

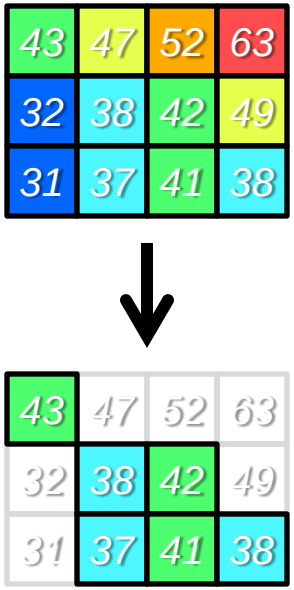
Example: 2nd-order quadtree from MADNESS

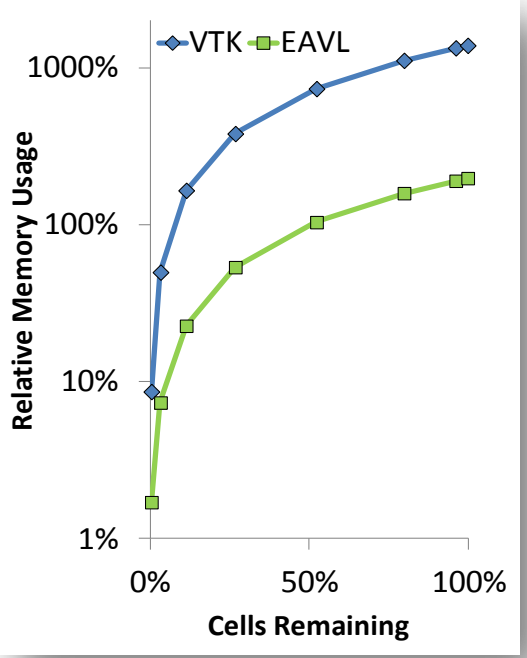


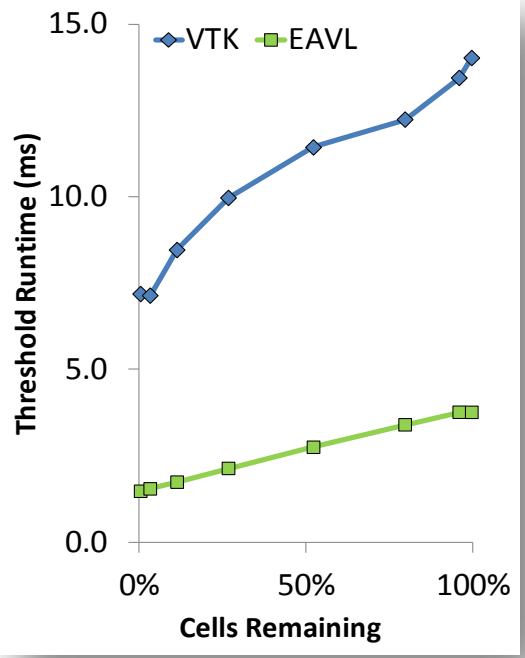
(a) constant (b) bilinear (c) biquadratic

Example: Regular Grid Threshold

- More descriptive data model is more memory efficient
- ... and more computationally efficient!







A Flexible Data Model

- Flexibility and greater descriptive power
  - Broader spectrum of data is representable
  - More efficient representations: memory and computation
  - Zero-copy for tightly coupled in situ processing
- Support for GPUs
  - Heterogeneous memory spaces
    - Zero-copy supporting GPU simulations
  - Both strided and separated array (performance and flexibility)
  - Enable methods for addressing concurrency

Data Parallel Operations With Functors

Map with 1 input, 1 output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
result	6	14	0	2	8	0	0	8	10	6	2	0

```
struct f {
   float operator()(float x) { return x*2; }
};
```

Simplest data-parallel operation. Each result item can be calculated from its corresponding input item alone.

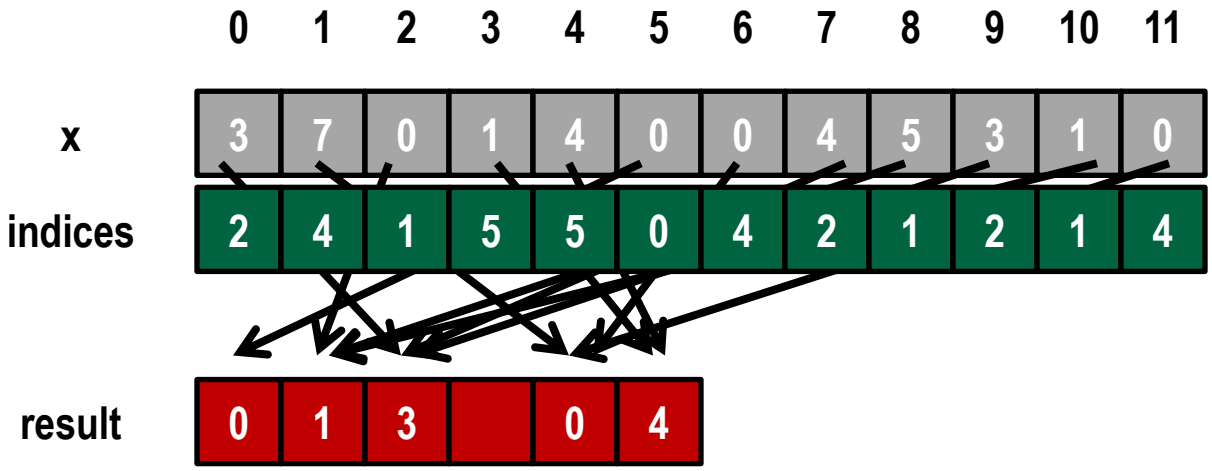
Map with 2 inputs, 1 output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
y	2	4	2	1	8	3	9	5	5	1	2	1
result	5	11	2	2	12	3	9	9	10	4	3	1

```
struct f {
   float operator()(float a, float b) { return a+b; }
};
```

With two input arrays, the functor takes two inputs. You can also have multiple outputs.

Scatter with 1 input (and thus 1 output)

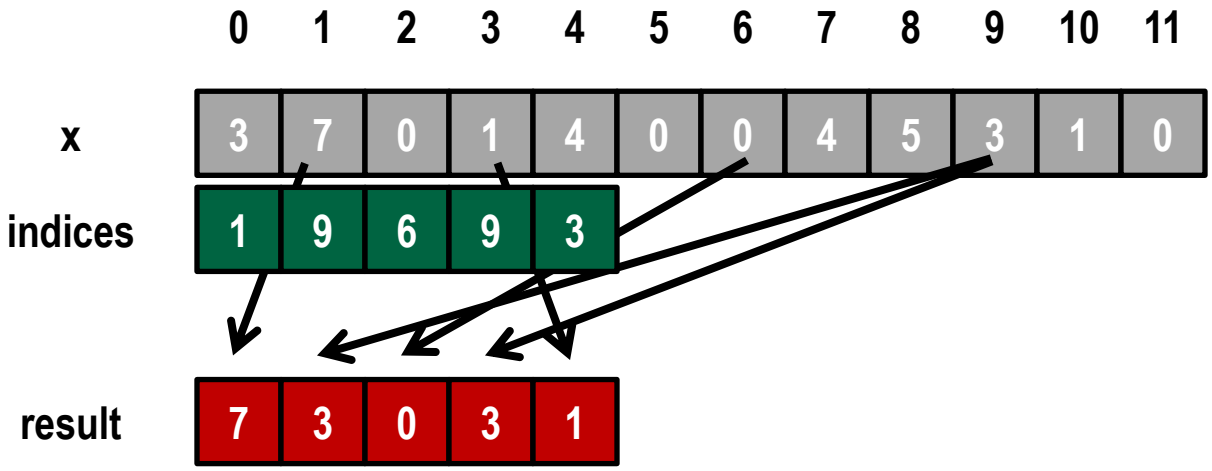



```
No functor
```

Possibly inefficient, risks of race conditions and uninitialized results. (Can also scatter to larger array if desired.)

Often used in a `scatter_if`–type construct.

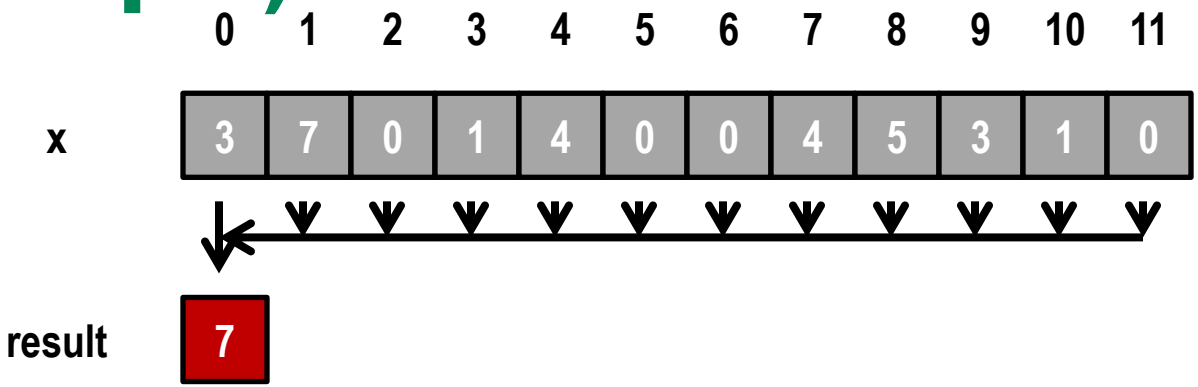
Gather with 1 input (and thus 1 output)




```
No functor
```

Unlike scatter, no risk of uninitialized data or race condition. Plus, parallelization is over a shorter indices array, and caching helps more, so can be more efficient.

Reduction with 1 input (and thus 1 output)




```
struct f {
   float operator()(float a, float b) { return a>b ? a : b; }
};
```

Example: max-reduction. Sum is also common.

Often a fat-tree-based implementation.

Inclusive Prefix Sum (a.k.a. Scan) with 1 input/output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
result	3	10	10	11	15	15	15	19	24	27	28	28

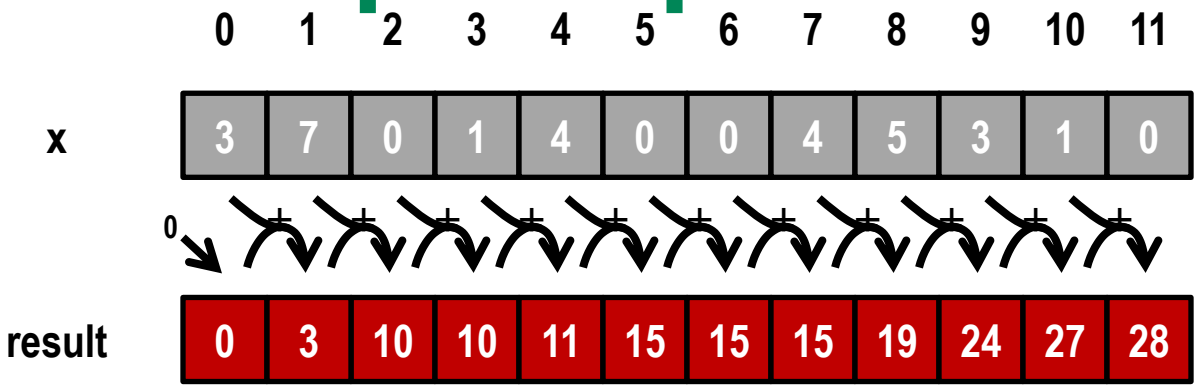
```
No functor.
```

Value at result[*i*] is sum of values x[0]..x[*i*].

Surprisingly efficient parallel implementation.

Basis for many more complex algorithms.

Exclusive Prefix Sum (a.k.a. Scan) with 1 input/output




```
No functor.
```

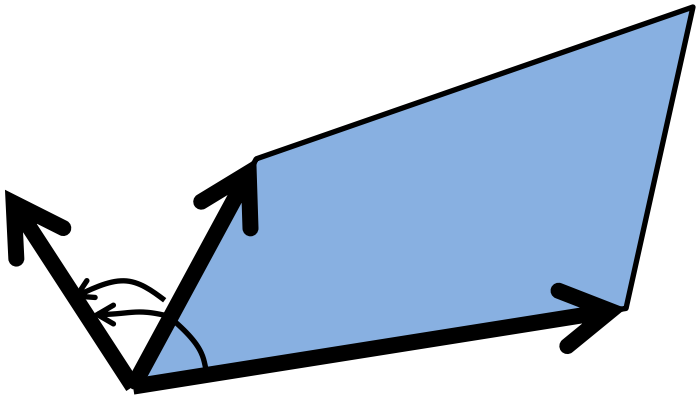
Initialize with zero, value is sum of only up to x[*i*-1].

May be more commonly used than inclusive scan.

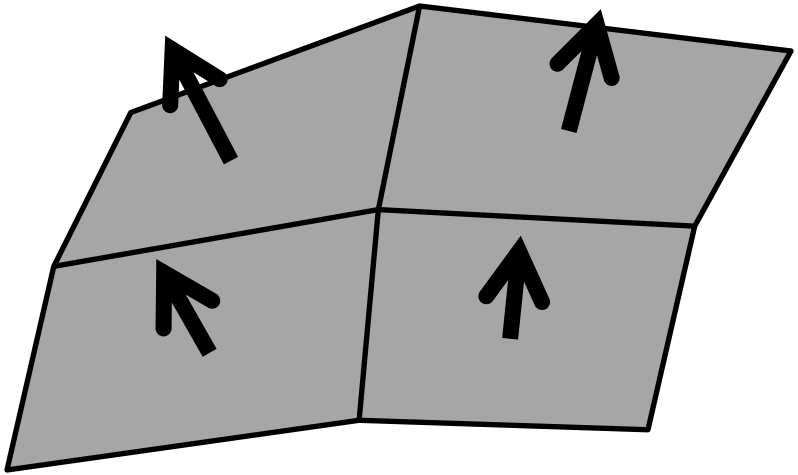
Data Parallelism on Meshes

Example: Surface Normal

- For each 2D cell (i.e. each polygon):
  - Get three adjacent points
  - Pair-wise vector subtract
  - Cross product

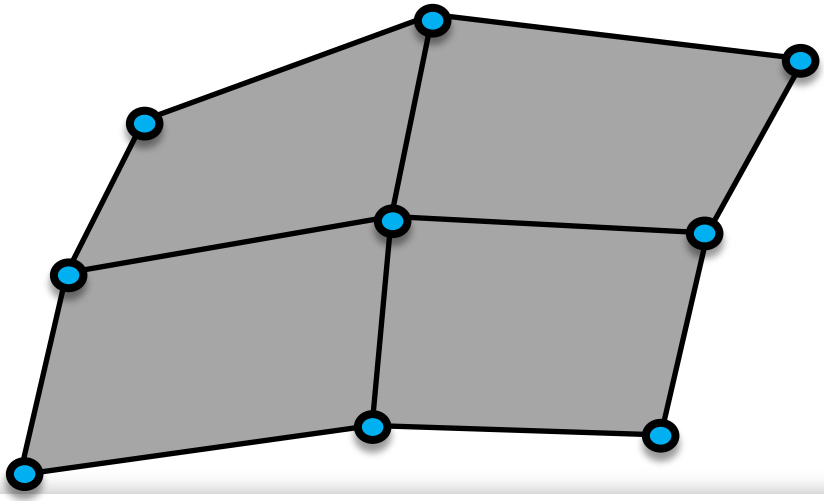


- Data-parallel:
  - Repeat for all cells

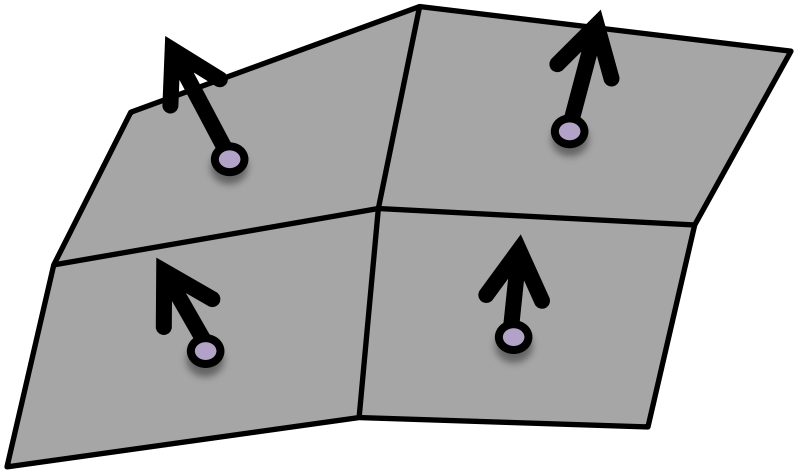


Example: Surface Normal

- INPUT:
  - 3-dimensional coordinates array on the mesh NODES
  - example: length = 9

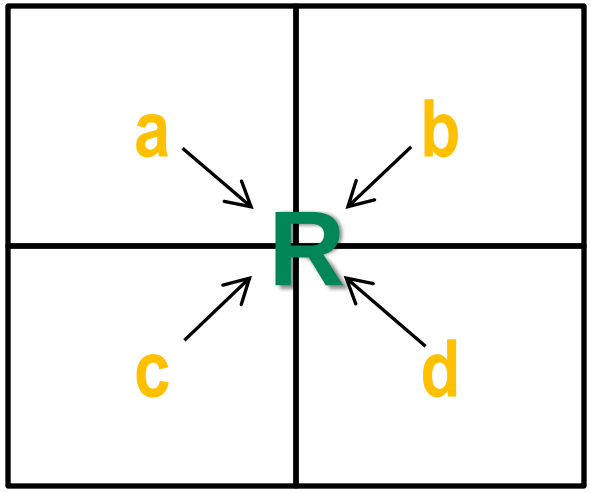


- OUTPUT:
  - 3-component surface normals array on the mesh CELLS
  - example: length = 4



In EAVL: Topology Maps!

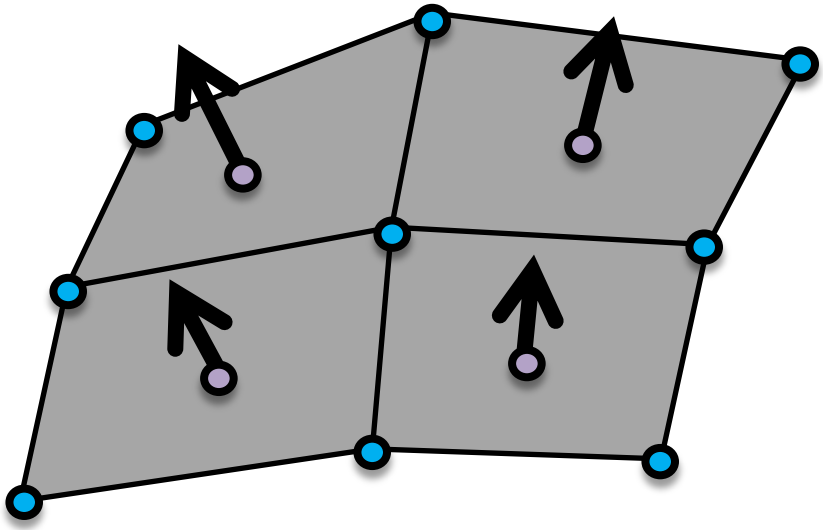
- Map one topology type to another
- Example: cell-to-node map
- Supports cells, nodes, faces, edges
- Supports index lists for sparse or packed maps
- Can be used to obtain array values or topological indices



R = *f*(a,b,c,d)

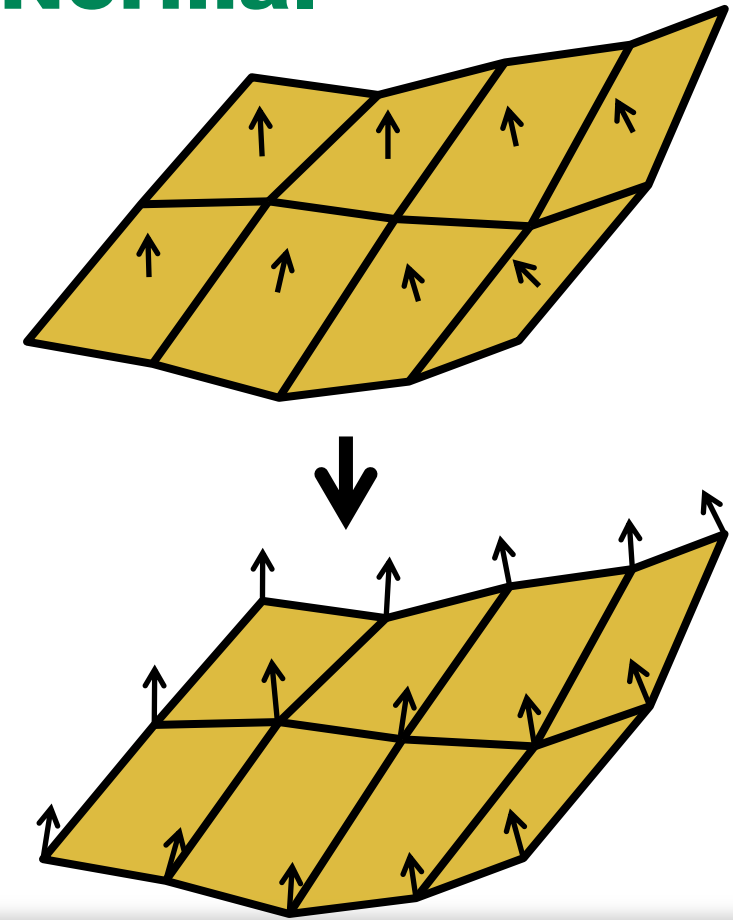
Example: Face Surface Normal

- eavlTopologyMap
  - node-to-cell topology
  - inputs:
    - x, y, z coordinate fields
  - outputs:
    - nx, ny, nz surface normal fields
  - functor:
    - adjacent edge cross product

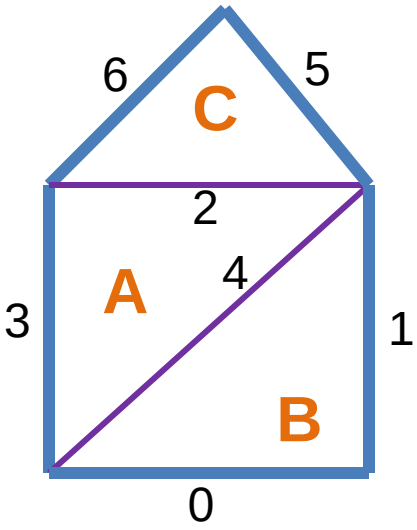


Example: Point Surface Normal

- eavlTopologyMap
  - cell-to-node topology
  - inputs:
    - face surface normal x/y/z
  - outputs:
    - point surface normal x/y/z
  - functor:
    - 3-component average

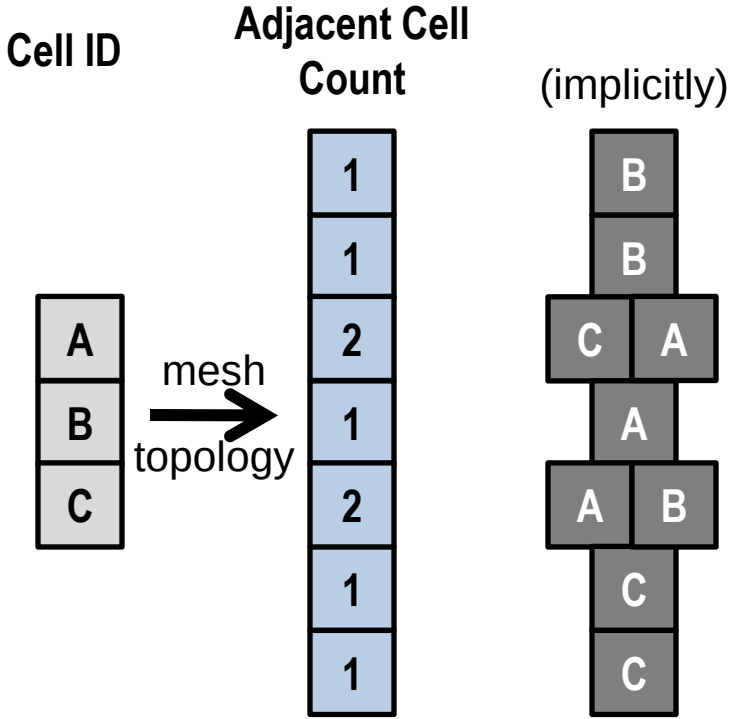


Example: External Edges



Using a simple Cell-to-Edge TopologyMap, count the number of cells adjacent to an edge.

Any edge with adjacency count of 1 is External.



Writing Algorithms in EAVL

Unary Math Operator

```
struct NegateFunctor
{
  float operator()(float val)
  {
      return –val;
  }
};

void DoMathStuff(...)
{
  eavlExecutor::AddOperation(
      new eavlMapOp(eavlOpArgs(a),
                    eavlOpArgs(negative_a).
                    NegateFunctor()));
}
```

Binary Math Operator

```
struct AddFunctor
{
  float operator()(tuple<float,float> vals)
  {
      return get<0>(vals) + get<1>(vals);
  }
};

void DoMathStuff(...)
{
  eavlExecutor::AddOperation(
      new eavlMapOp(eavlOpArgs(a, b),
                    eavlOpArgs(a_plus_b).
                    AddFunctor()));
}
```

Face Surface Normal

```
void CalculateFaceNormals(...)
{
  eavlExecutor::AddOperation(
    new eavlSimpleTopologyMap(inputcells,
                              EAVL_NODES_OF_CELLS,
                              eavlOpArgs(xcoord, ycoord, zcoord),
                              eavlOpArgs(xnormal, ynormal, xnormal),
                              PolyNormalFunctor()));
}
```

Face Surface Normal Functor

```
struct PolyNormalFunctor
{
  template <class IN>
  tuple<float,float,float> operator()(int ids[], IN pts) {
    // get the coords for three adjacent points
    eavlVector3 pt0 = collect(ids[0], pts);
    eavlVector3 pt1 = collect(ids[1], pts);
    eavlVector3 pt2 = collect(ids[2], pts);

    // get two adjacent edge vectors
    eavlVector3 a = pt1 - pt0;
    eavlVector3 b = pt2 - pt1;

    // calculate their cross product
    eavlVector3 c = cross(a,b);
    return tuple<float,float,float>(c.x, c.y, c.z);
  }
};
```

EXAMPLE: THRESHOLD

Starting Mesh

43	47	52	63
32	38	42	49
31	37	41	38

We want to threshold a mesh based on its density values (shown here).

	0	1	2	3	4	5	6	7	8	9	10	11
density	43	47	52	63	32	38	42	49	31	37	41	38

If we threshold

35 < density < 45, we want this result:

43	47	52	63
32	38	42	49
31	37	41	38

Which Cells to Include?

Evaluate a Map operation with this functor:

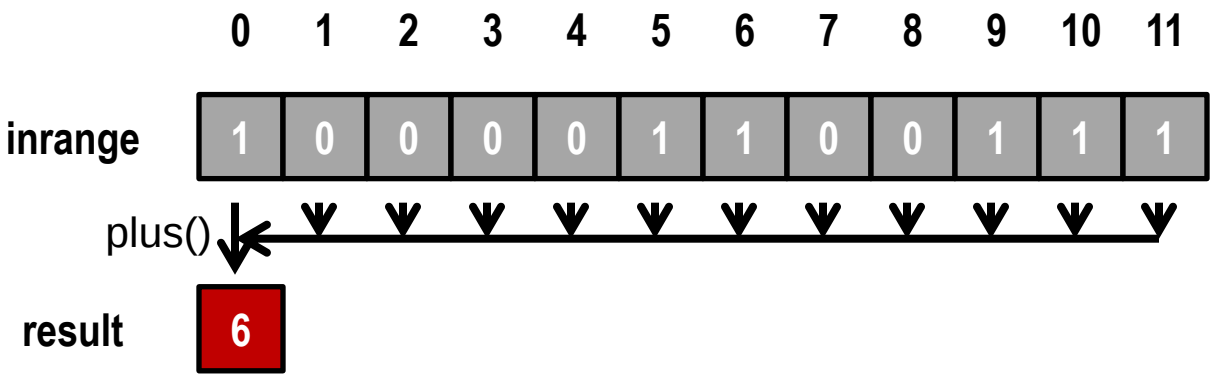
```
struct InRange {
 float lo,hi;
 InRange(float l, float h) : lo(l),hi(h) {}
 int operator()(float x){return x>lo && x<hi;}
}
```

	0	1	2	3	4	5	6	7	8	9	10	11
density	43	47	52	63	32	38	42	49	31	37	41	38
InRange()	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
inrange	1	0	0	0	0	1	1	0	0	1	1	1

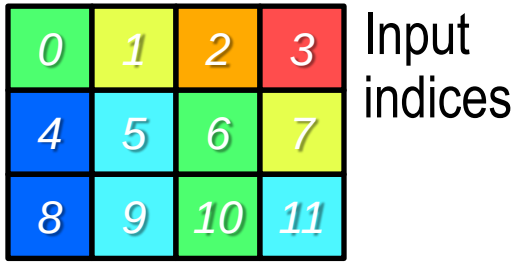
How Many Cells in Output?

Evaluate a Reduce operation using Add<> functor.

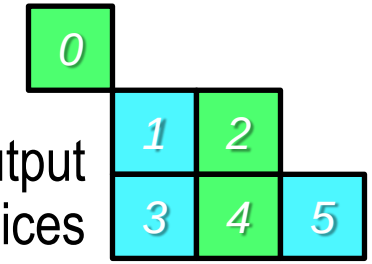
We can use this to create output cell length arrays.



Where Do the Output Cells Go?



Input indices

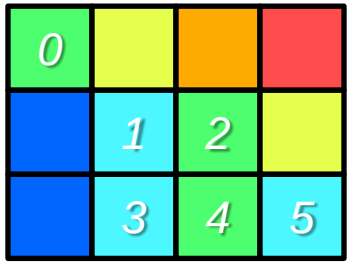


Output indices

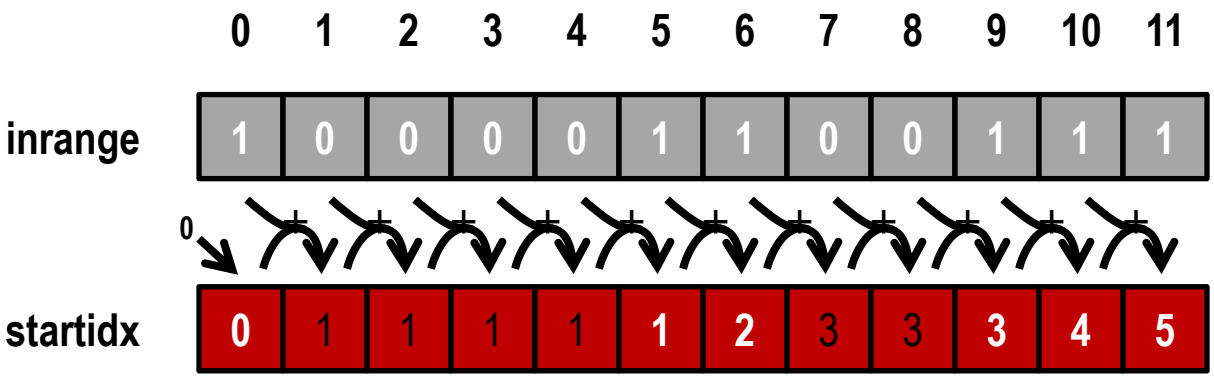
input cell	0	1	2	3	4	5	6	7	8	9	10	11
output cell	0					1	2			3	4	5

How do we create this mapping?

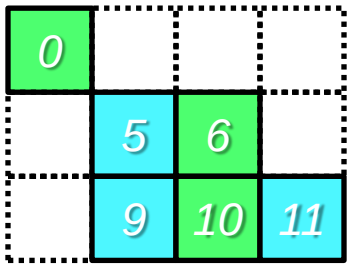
Create Input-to-Output Indexing



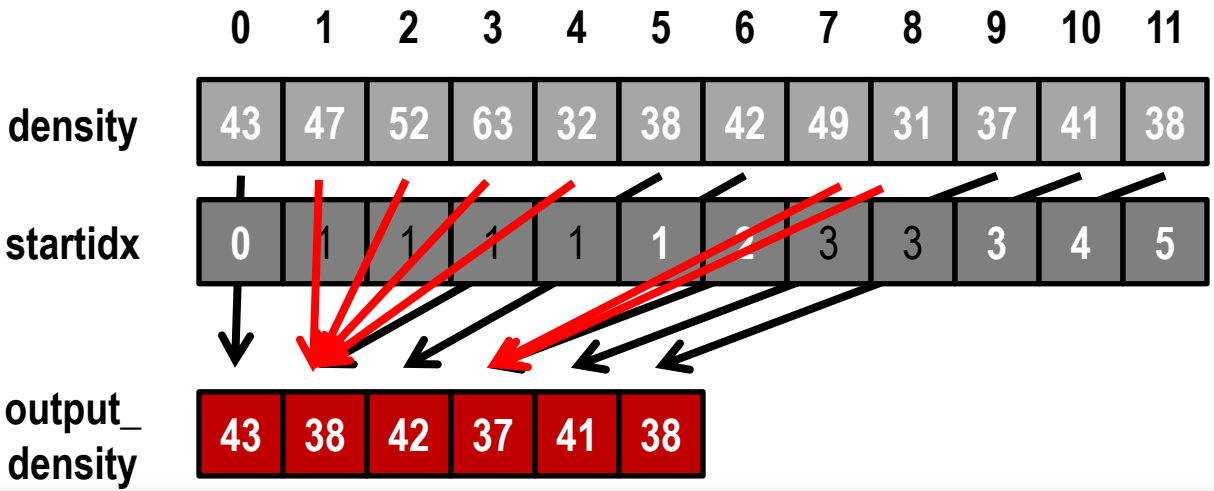
Exclusive Scan (exclusive prefix sum) gives us the output index positions.



Scatter Input Arrays to Output?

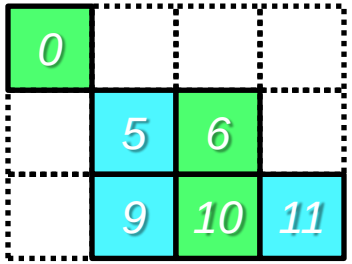


NO. We can do this, but scatters can be risky/inefficient. Assuming we have multiple arrays to process, we can do something better....

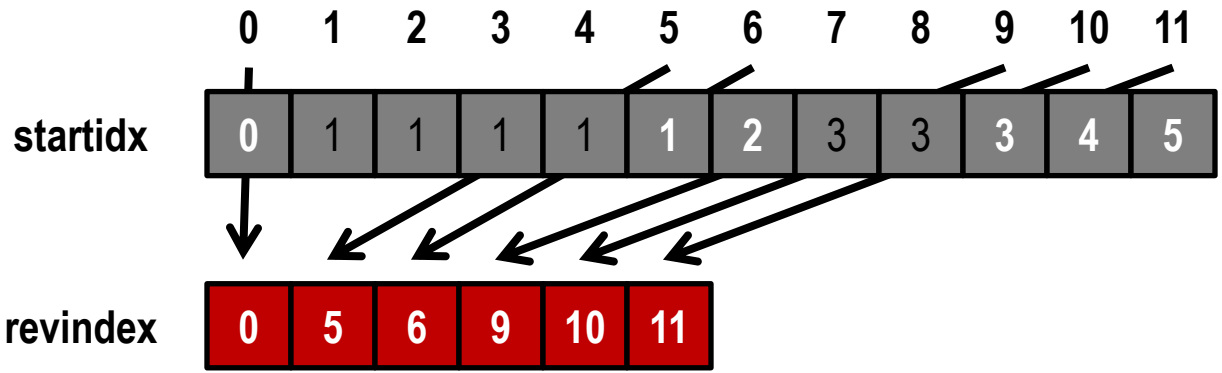


Race condition unless we add a mask array!

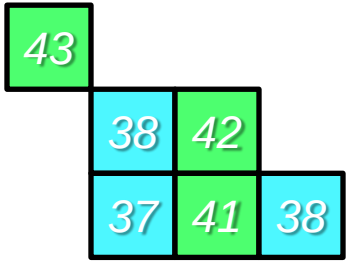
Create Output-to-Input Indexing



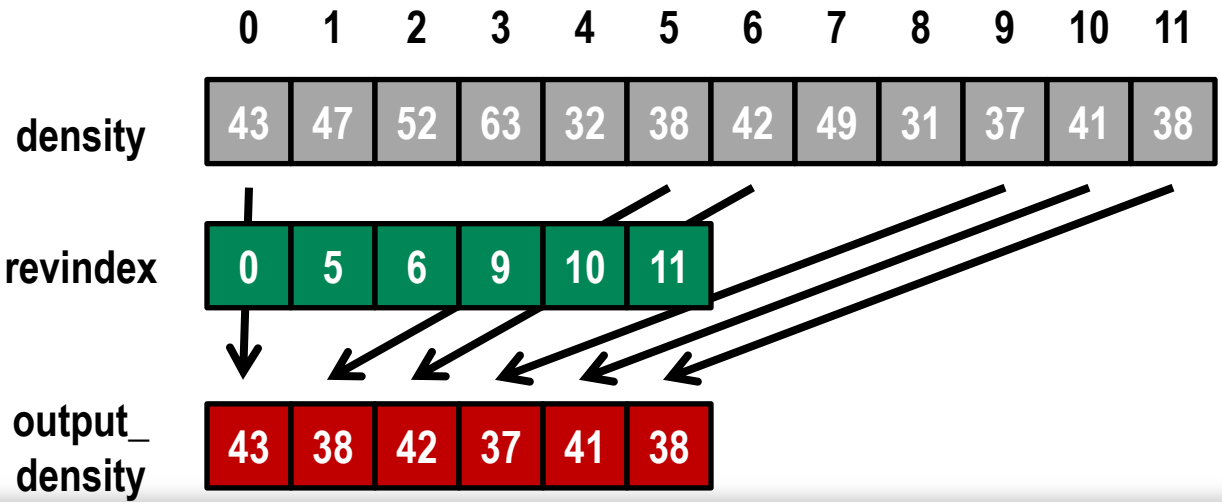
We want to work in the shorter output-length arrays and use gathers. A specialized scatter in EAVL creates this reverse index.



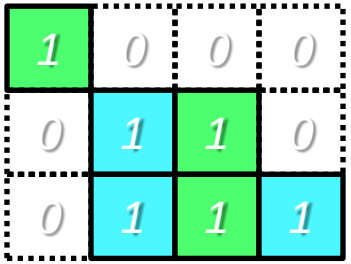
Gather Input Mesh Arrays to Output



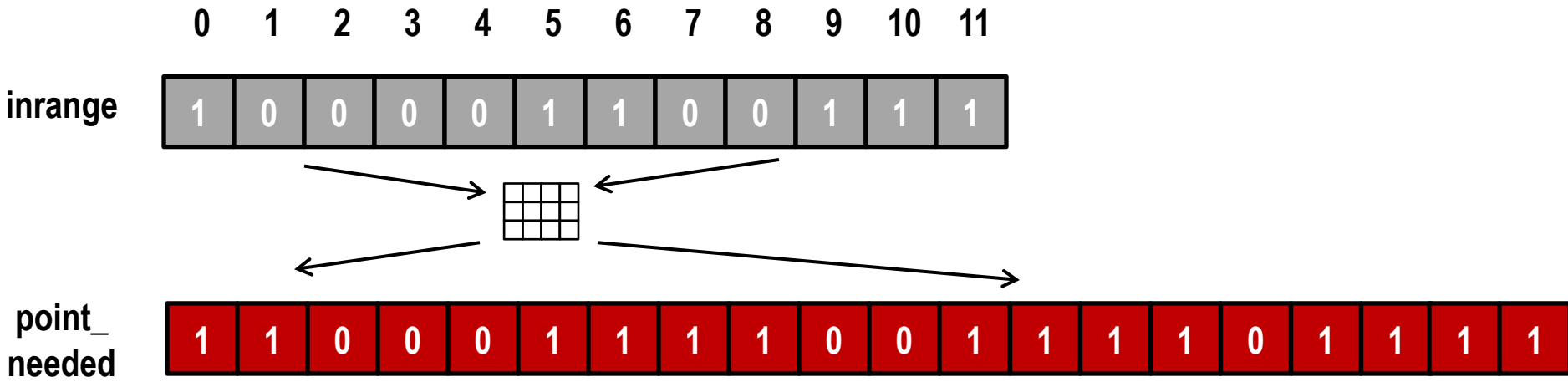
We can now use simple gathers to pull input arrays (density, pressure) into the output mesh.



Tag Input Points (optional compaction)



Use a simple TopologyMap with cell-to-node topology, input field “inrange”, and Max<> functor.



TECHNICALITIES

Multiple Back-end Implementations

- Data-parallel operations implemented in CUDA, OpenMP
  - GPU and multi-core CPU support
- Single functor supports all back ends
  - Write your functors once
  - Supports data model within functors
    - eavlVector and other math operations
- Template meta-programming, requires C++
  - Functors inlined at compile time, optimizes well

Data Model Support

- All operations support all EAVL mesh types
  - unstructured versus structured cell arrangements
  - implicit versus explicit arrays/coordinates
  - strided versus separated arrays

Example: separate x/y arrays (SoA)

```
eavlMapOp( eavlOpArgs(x,y), ... )
```

Example: x/y interleaved in strided coord array (AoS)

```
eavlMapOp( eavlOpArgs( eavlIndex(coord,0), eavlIndex(coord,1)), ... )
```

CUDA Support

- EAVL's entire data model supports CUDA
- Internally (for EAVL coders)
  - Data-parallel operations written once for both CPU and GPU
  - E.g. can access cell connectivity within GPU using standard functions
- For algorithm developers
  - If data parallel patterns don't work for you, writing standard CUDA kernels is well supported

Runtime Array Types

- A pure compile-time implementation is “optimal”
- ... but only supports single code, e.g. in situ analysis
- EAVL supports post-processing analysis
  - e.g. end-user tools read data off disk, unknown if float vs double, etc.
- For arrays not known at compile time
  - EAVL generates all type options at compile time
  - Selects proper code path at runtime
- Yet generates single code path for types known at compile time

Runtime Scheduling

- eavlExecutor keeps a stack of operations
- Execution triggered when needed (or when you say “::go()”)
- Smart scheduling, e.g. “PreferGPU” algorithm
  - tries GPU, falls back to CPU on failures like out-of-GPU-memory
  - transparent to algorithm developer, user
- Transparent heterogeneous memory space support
  - algorithm developer is oblivious to memory space; just uses eavlArray

Conclusion

Summary

- EAVL is a lightweight visualization and analysis library
- Advanced data model provides
  - flexibility for many mesh arrangements and data types
  - memory efficiency and computational efficiency
  - straightforward in situ support
- Algorithm development API
  - efficient parallel primitives on flat arrays and mesh structures
  - productive programming for multi- and many-core architectures
  - advanced scheduling for scalable heterogeneous systems

Thanks!

- Project website at http://ft.ornl.gov/eavl
- Source code, docs at https://github.com/jsmeredith/EAVL
- Email team at eavl-dev@email.ornl.gov
- My email: jsmeredith@ornl.gov
- ASCR (SDAV, PSI), ORNL LDRD