STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models

Zach Weiner

CS 598 APK

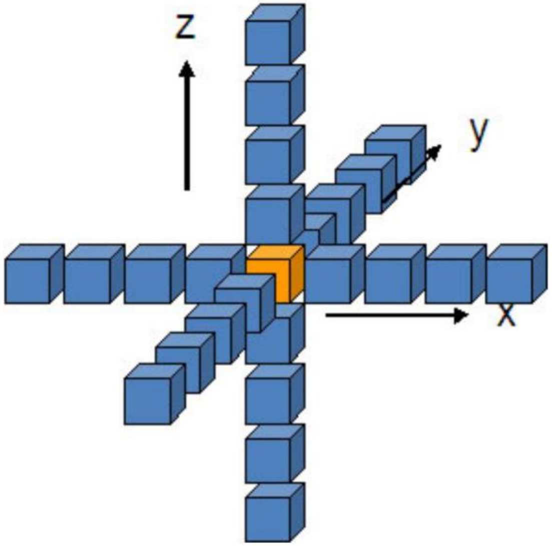
October 12, 2018

The problem

- Solving PDEs on structured grids
  - Atmospheric/climate science
  - Computational fluid dynamics
  - Material science
- Optimizing performance on a variety of (heterogeneous) architectures requires:
  - Loop tiling/blocking
  - Loop fusion
  - Data layout transformations
  - etc.
- All dependent upon specific architecture
- Coding for high performance becomes increasingly difficult

Atmospheric simulations

- Solving Navier-Stokes equations (fluid dynamics) on 3D curvilinear grid
- Runge-Kutta for time integration
- Finite-difference stencils for spatial derivatives



Laplacian pseudo-code

$$\nabla^2\phi = \sum_i \frac{\partial^2\phi}{\partial x_i^2} \approx \sum_i \frac{1}{\Delta x_i}\left[\phi(x+\hat{\imath}) + \phi(x-\hat{\imath}) - 2\phi(x)\right]$$

```
for k = kstart, kend
    for j = jstart-1, jend+1
        for i = istart-1, iend+1
            lap(i,j,k) = phi(i+1,j,k) + phi(i-1,j,k)
                       + phi(i,j+1,k) + phi(i,j-1,k)
                       - 4.0*phi(i,j,k)
```

- Gradients of lap needed for PDE

Performance concerns

- Stencil computation is memory bound (minimal arithmetic)
  - Use data locality: process smaller subdomains which fit into cache
- In naïve implementation, intermediate/temporary fields are stored across the entire domain
- For atmospheric simulations: many different types of stencils, composed stencils

STELLA

- STEncil Loop LAnguage
- DSL: abstracts architecture-dependent implementation details from the solution algorithm
- Handles stencil computation, boundary conditions, and halo-update communication
- As a library, is specific to structured grids and stencils/domain decomposition
  - Still holds broad applicability in many sciences
- “Separation of concerns:” user defines PDEs, STELLA deals with optimization
  - Allows user code to be more concise/resemble underlying mathematical expressions

STELLA

- Currently: portable performance between x86 multicore CPUs and NVIDIA GPUs
  - CPU backend with OpenMP
  - GPU backend with CUDA
  - Xeon Phi backend under development(?)
- Uses standard C++ compilers
- At compile time, DSL is translated into optimized nests of loops
  - Uses C++ template metaprogramming

STELLA usage

- Stencils defined by:
  - Function objects of stencil loop bodies
    - “Stencil stages”
  - DSL which allows multiple function objects to be assembled into one kernel
- Language constructs:
  - Parameters: values to be read/processed throughout stencil
  - Temporaries: buffers for temporary values
    - Optimized layout, alignment, memory footprint
  - Loops: data range, parallelization

STELLA example

- Stencil “stage” definition

```
template<typename Context>
struct LapStage{
    static void Do(Context ctx) {
        ctx[lap::Center()]  = ctx[u::At(iplus1)] + ctx[u::At(iminus1)]
            + ctx[u::At(jplus1)] + ctx[u::At(jminus1)] - 4*ctx[T::Center()];
    }
};
```

$$\begin{aligned}\mathrm{lap(i,j,k)} = {} & \mathrm{phi(i{+}1,j,k)} + \mathrm{phi(i{-}1,j,k)} \\ & + \mathrm{phi(i,j{+}1,k)} + \mathrm{phi(i,j{-}1,k)} \\ & - 4.0 * \mathrm{phi(i,j,k)}\end{aligned}$$

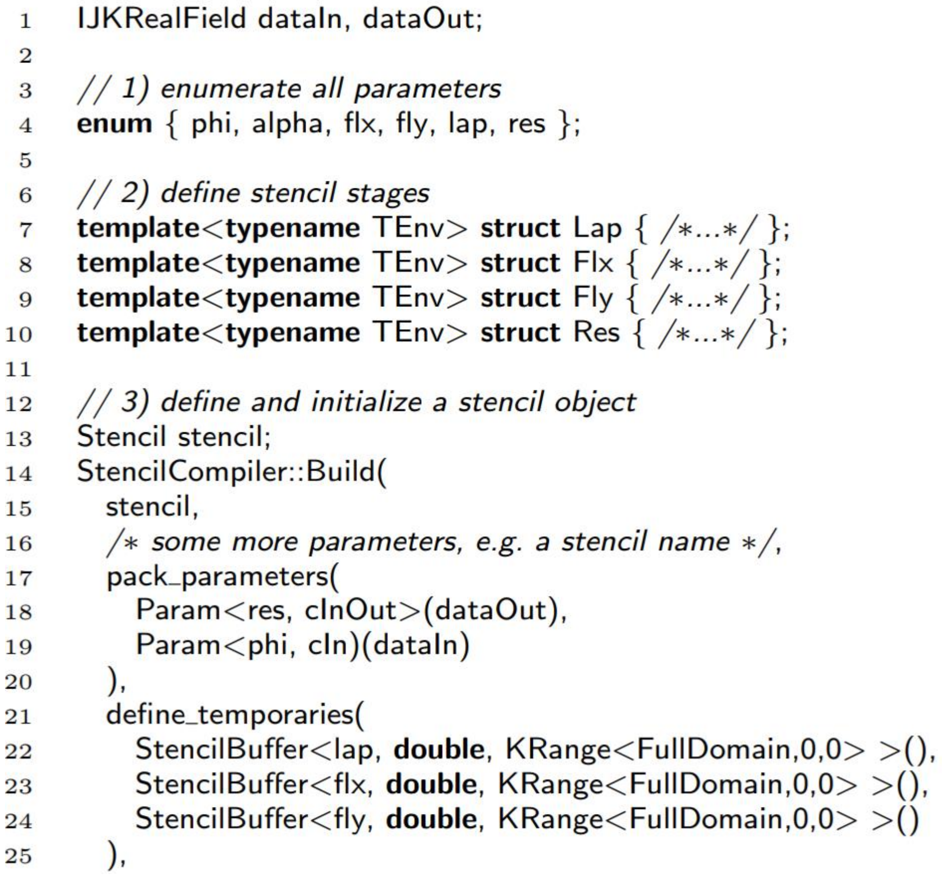
STELLA example (1/2)

Data fields, abstracted memory layout

Define various stages of stencil

Associate parameter packs to placeholders used below

Define temporary storage used in stencil logic

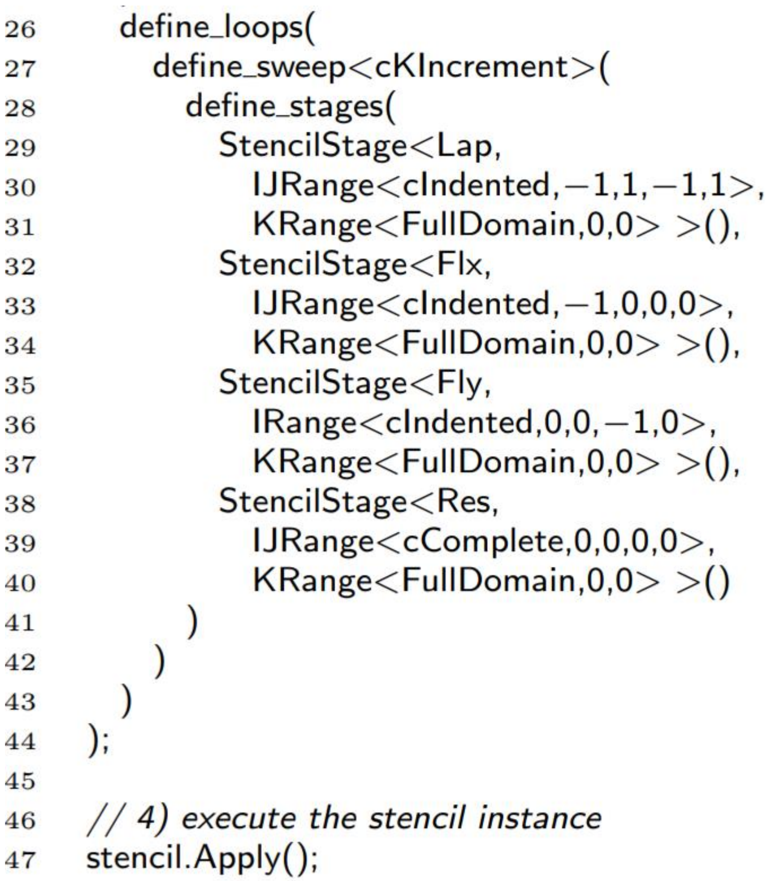


```
IJKRealField dataIn, dataOut;

// 1) enumerate all parameters
enum { phi, alpha, flx, fly, lap, res };

// 2) define stencil stages
template<typename TEnv> struct Lap { /*...*/ };
template<typename TEnv> struct Flx { /*...*/ };
template<typename TEnv> struct Fly { /*...*/ };
template<typename TEnv> struct Res { /*...*/ };

// 3) define and initialize a stencil object
Stencil stencil;
StencilCompiler::Build(
  stencil,
  /* some more parameters, e.g. a stencil name */,
  pack_parameters(
    Param<res, cInOut>(dataOut),
    Param<phi, cIn)(dataIn)
  ),
  define_temporaries(
    StencilBuffer<lap, double, KRange<FullDomain,0,0> >(),
    StencilBuffer<flx, double, KRange<FullDomain,0,0> >(),
    StencilBuffer<fly, double, KRange<FullDomain,0,0> >()
  ),
```

STELLA example (2/2)

Compose stencil stages

Apply stencil



```
  define_loops(
    define_sweep<cKIncrement>(
      define_stages(
        StencilStage<Lap,
          IJRange<cIndented,-1,1,-1,1>,
          KRange<FullDomain,0,0> >(),
        StencilStage<Flx,
          IJRange<cIndented,-1,0,0,0>,
          KRange<FullDomain,0,0> >(),
        StencilStage<Fly,
          IRange<cIndented,0,0,-1,0>,
          KRange<FullDomain,0,0> >(),
        StencilStage<Res,
          IJRange<cComplete,0,0,0,0>,
          KRange<FullDomain,0,0> >()
      )
    )
  )
);

// 4) execute the stencil instance
stencil.Apply();
```

Other functionality

- Software-managed caches
  - Two types: caching of neighbors in 2D parallel plane and caching of levels of the third dimension
  - Can, e.g., buffer temporary values in GPU shared memory
- Boundary conditions: specify boundary handling
- Halo updating
- Domain splitting: distinguish domains with different geometries (cartesian vs. curvilinear)

Implementation

- Compile-time code generation with C++ template meta-programming via Boost MPL library
  - DSL translated into sequence of template instantiations
  - Avoids runtime code generation overhead
  - No auto-tuning
- During compilation: assemble loop logic, instantiate stencil stages, define needed data structures
- At execution: initialize stencil object
- Apply method: thin wrapper around generated loop code

Parallelization

- Coarse-grained blocking with fine-grained threads
  - Makes use of data locality
- Overlapped tiling: (redundant) halo elements are computed when needed so blocks are independent
- Blocks updated in parallel via vector instructions or hardware threads

Backend-dependent decisions

- Array layout

	CPU	GPU
Programming model	OpenMP	CUDA
Storage order (by stride)	$j > i > k$	$k > j > i$

(auto) vectorization over k

memory access coalescing

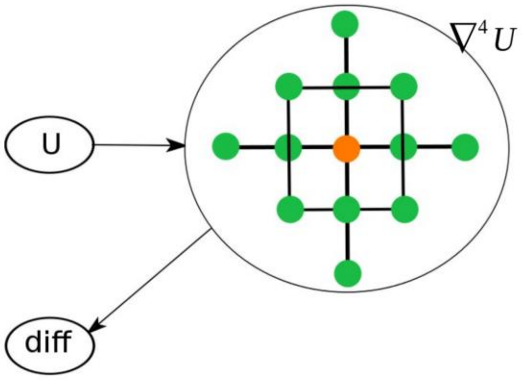
Backend-dependent decisions

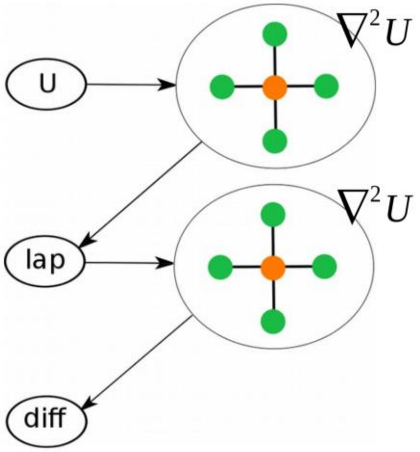
- Loop fusion: compute in two stages (with full temporary array), or nest both stages in one loop?

Architecture	Computation	Communication
E5-2670 (ms)	27.77	21.98
K20X (ms)	9.61	13.38

nested

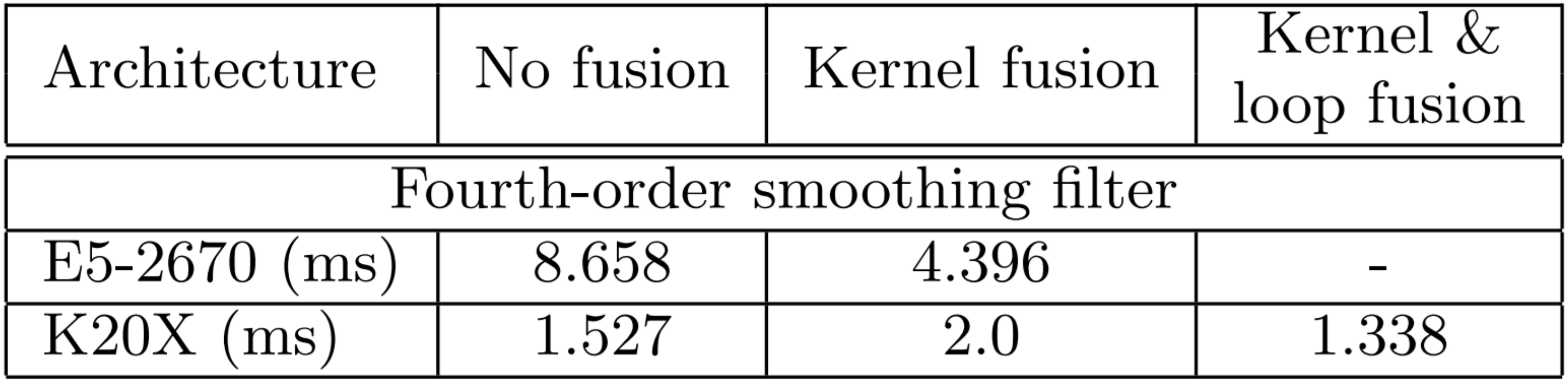
two stages





Kernel fusion

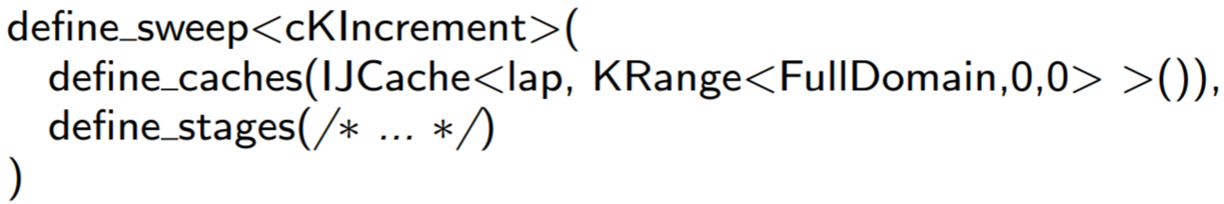
- Reduce off-chip memory traffic by caching reused data
  - Cache intermediate results in shared memory (on GPUs), synchronize block, and compute final result
- Shared memory too small to cache between kernels; CPU L1 cache is large enough
- “Kernel & loop fusion” = all stencil stages in single loop



Architecture	No fusion	Kernel fusion	Kernel & loop fusion
Fourth-order smoothing filter			
E5-2670 (ms)	8.658	4.396	-
K20X (ms)	1.527	2.0	1.338

What does the user choose?

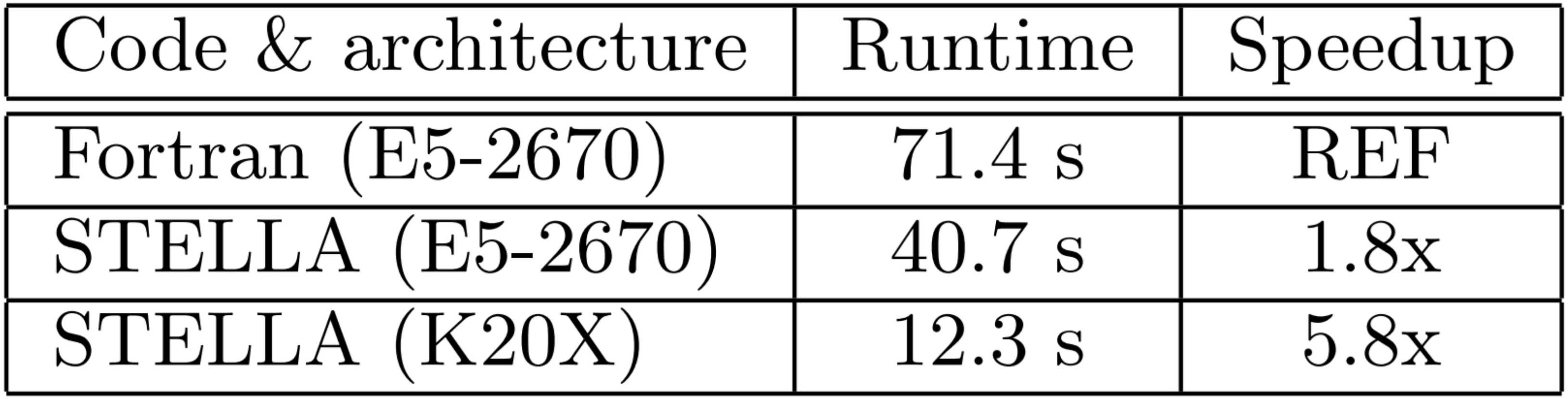
- Kernel and loop fusion
- Caching?
  - “Given this annotation, STELLA’s GPU backend is able to automatically buffer the lap value in shared memory.”



```
define_sweep<cKIncrement>(
   define_caches(IJCache<lap, KRange<FullDomain,0,0> >()),
   define_stages(/∗ ... ∗/)
)
```

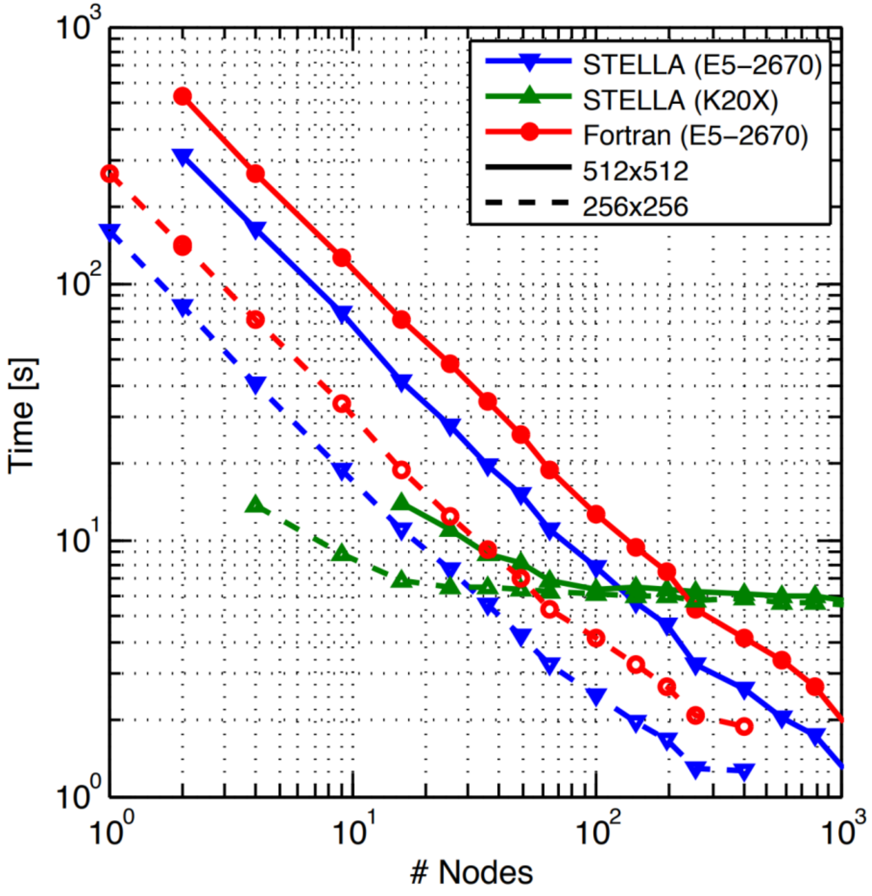
- Whether to parallelize third dimension

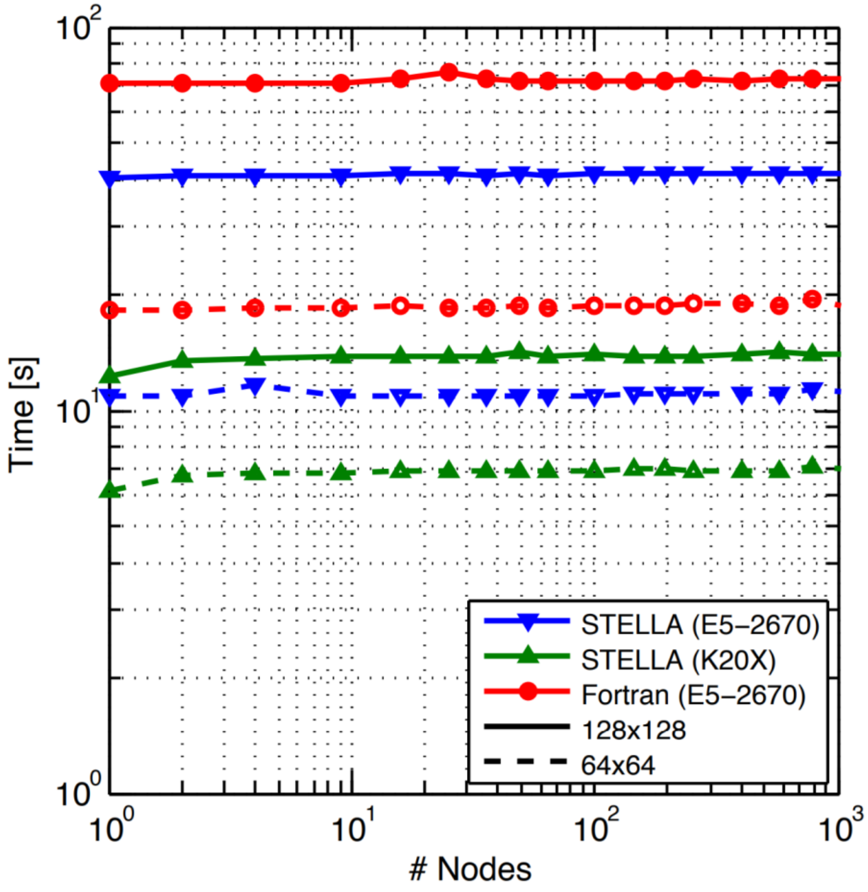
Timing results



Code & architecture	Runtime	Speedup
Fortran (E5-2670)	71.4 s	REF
STELLA (E5-2670)	40.7 s	1.8x
STELLA (K20X)	12.3 s	5.8x

Weak and strong scaling





Future directions

- Improved syntax
- Parallelization in third dimension
- Different geometries
- Performance-model based tuning framework to automate loop/kernel fusion choices

Related work

- Other stencil DSLs all rely on custom compilation/translation toolchains (at the time)
  - Emphasize value of being able to fall back on host language (C++) for non-STELLA kernels
- Patus: stencil kernel generator for CPU and GPU, emphasizes autotuning
- ATMOL: also abstracts solvers
- ICON: mainly abstracts storage order
- Halide (image processing): 2D only
- Pochoi: c++, custom compilation optional, general dimension

Conclusions

- Generates CPU and GPU code
- Abstracted for arbitrary stencil (for a domain which has implements many stencils)
- Aren't specific enough about tests to be sure, but GPU kernels take O(ms) for 256^2*60 gridpoints—are they saturating bandwidth?
  - GPU only ~3.2x faster than CPU (despite >5x bandwidth)
- User must decide whether to use kernel/loop fusion, structure and parallelization of "sweeps"
- Kernel fusion could apply across stencil and integration routines
- Readability: stencil definitions are worse than normal C code