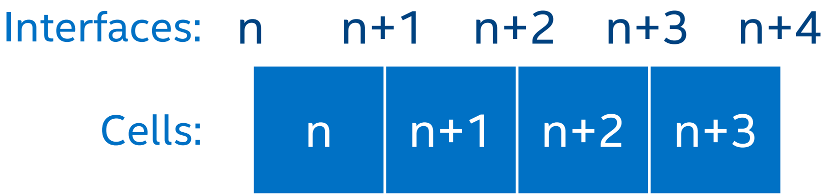
Generalizing a DSL for Structured Dependency ("Stencil-like") Codes to OpenMP* Loops

John Pennycook and Jason Sewall, Intel HPC Ecosystem and Applications Team

DOE COE Performance Portability Meeting, April 2016, Arizona

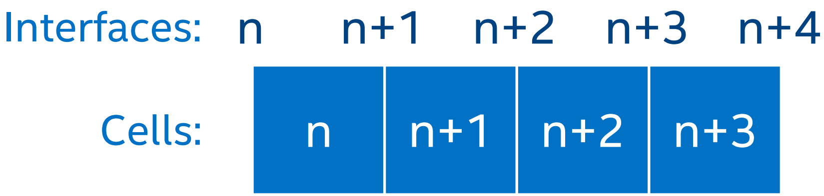
A Simple Example (1/3)



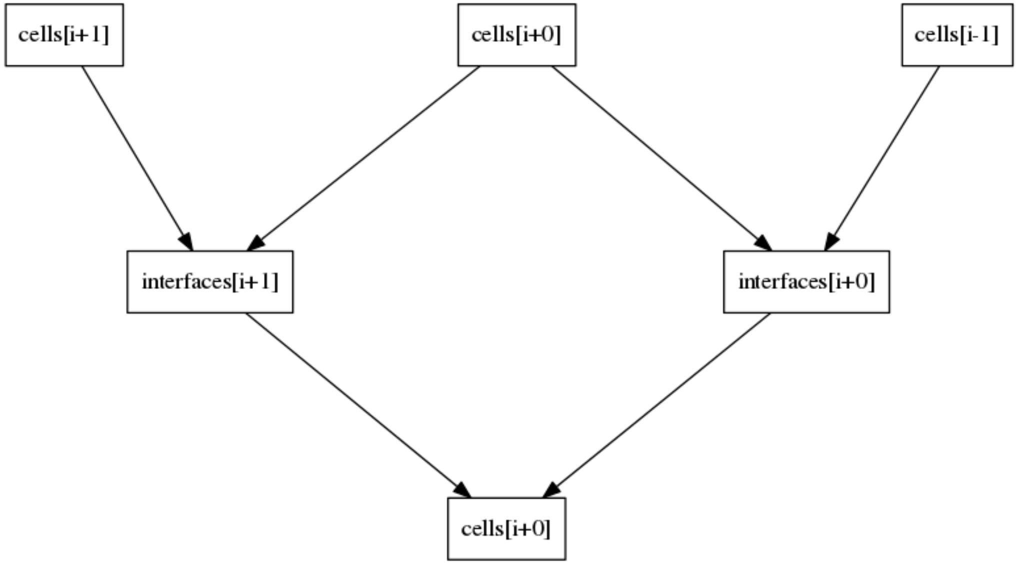

```
// Two simple kernels
void flux(double prev_cell, double next_cell, double &interface);
void integrate(double prev_interface, double next_interface, double &cell);

// Typical parallel implementation (in 1D)
#pragma omp parallel for simd
for (uint32_t itf = first_itf; itf < last_itf; ++itf)
{
     flux(cells[itf-1], cells[itf], interfaces[itf]);
}
#pragma omp parallel for simd
for (uint32_t c = first_cell; c < last_cell; ++c)
{
     integrate(interfaces[c], interfaces[c+1], cells[c]);
}
```

A Simple Example (2/3)



Data-flow DAG:

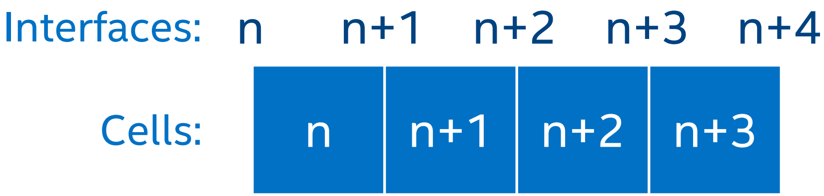


3 cell values as input

2 interface values as intermediates

1 cell value as output

A Simple Example (3/3)




```
// Two simple kernels
void flux(double prev_cell, double next_cell, double &interface);
void integrate(double prev_interface, double next_interface, double &cell);

// Rolling update implementation (in 1D)
#pragma omp parallel for
for each tile (first_cell, last_cell)
{
     double tmp_itf[2];
     flux(cells[first_cell-1], cells[first_cell], tmp_itf[0]); // Prologue
     for (uint32_t c = first_cell; c < last_cell; ++c)          // Steady-state
     {
          flux(cells[c], cells[c+1], tmp_itf[1]);
          integrate(tmp_itf[0], tmp_itf[1], cells[c]);
          tmp_itf[0] = tmp_itf[1];
     }
}
```

This forward dependency prevents auto-vectorization.

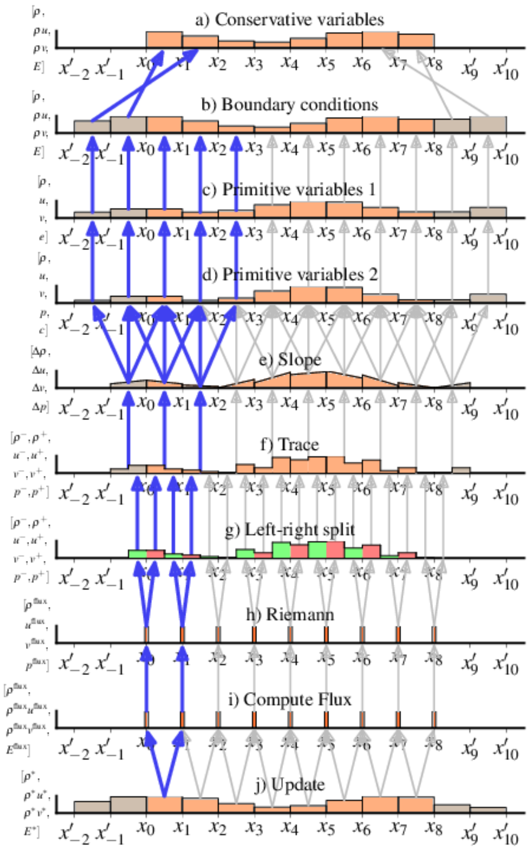
A Complicated Example – CEA's Hydro2D

- Implements 9 "parallel kernels" as:

```
for all cells in a slab:
  function();
synchronize



make_boundary();
constoprim();
equation_of_state();
slope();
trace();
qleftright();
riemann();
cmpflx();
updateConservativeVars();
```



Why a DSL/Code Generator?

Data dependency analysis is error-prone and time-consuming.

- Needs to be repeated each time application functionality is changed.
- Application functionality may change many times during optimization studies.
- Dependency analysis for proxy applications won't match the legacy application.

Rolling update code follows a pattern => copy-paste errors.

- Very easy to get a temporary index, or the rolling buffer size, wrong.

Rolling update loops have very real data dependencies.

- Compiler cannot vectorize the code (at all).
- Explicit vectorization requires intrinsics, SIMD classes or ugly OpenMP code.

Input Parameters

- Kernel description(s):

```
integrate:
        declaration: integrate(flux_t lf, flux_t rf, cell_t &ic);
        inputs: |
            lf : flux(q?[j?][i?])
            rf : flux(q?[j?][i?+1])
        outputs: |
            ic : integrated(q?[j?][i?])
```

Kernels are elemental functions representing production rules. "?" can be substituted.

- Axiom(s):

```
double cell[j?][i?]
```

Axioms exist before any kernels. Infinite extent assumed.

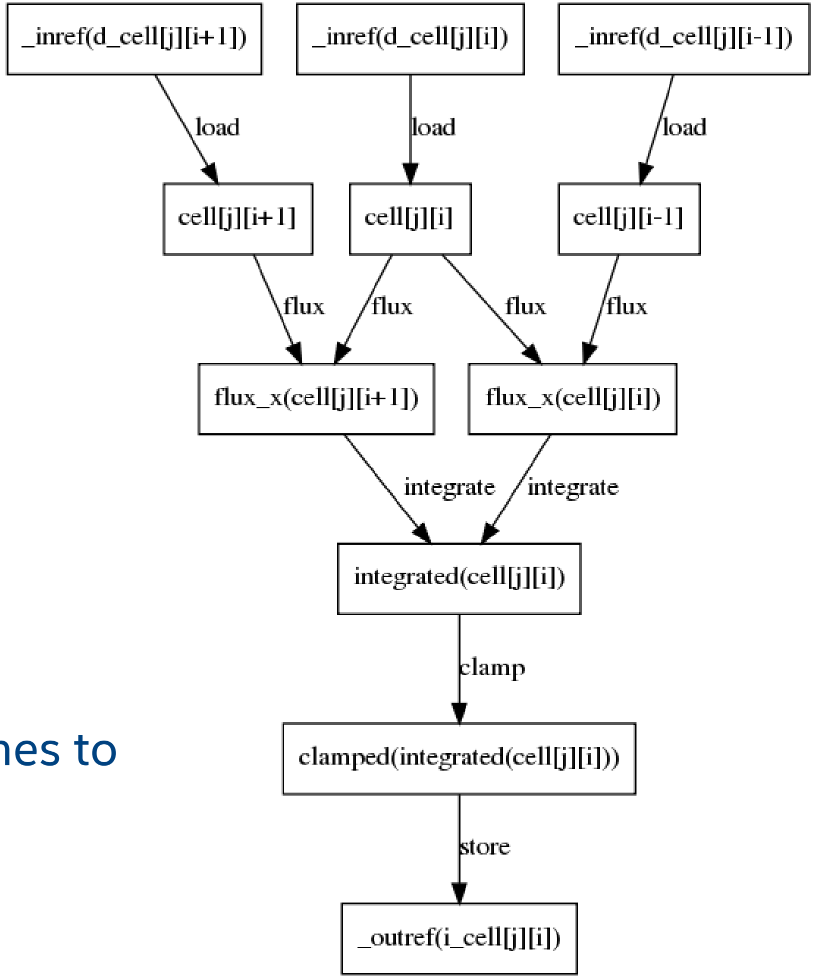
- Goal(s):

```
integrated(cell[j][i]) => double cell[j][i]
```

Inference works backward from goals to compute specified index.

Stage 1: Inference

- Start at a goal (e.g. i_cell[j][i]).
- Repeatedly apply rules and substitutions until we reach:
  - An axiom (e.g. d_cell[j][i]); or
  - A node already in the DAG.
- We have prefixed function names to ensure unique variable names.



Stage 2: Loop Nest Optimizations

- Serves two purposes:
  1. Identifies functions with a spatial relationship and fuses them; and
  2. Aggressively fuses loop nests (where safe to do so).

- In many cases, it is safe to fuse all loops – only “concavity” prevents it:

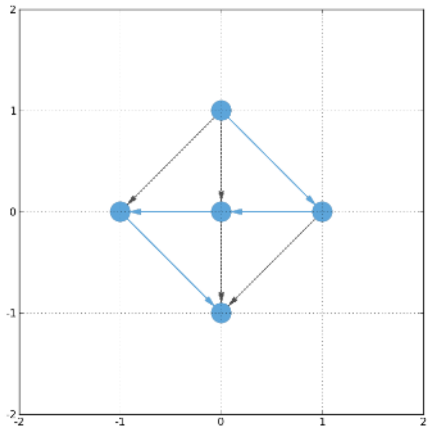
```
#pragma omp parallel for simd reduction(+:sum)
for (uint32_t i = 0; i < N; ++i)
{
    sum += f(input[i]);
}
#pragma omp parallel for simd
for (uint32_t i = 0; i < N; ++i)
{
    output[i] = g(sum);
}
```

Using the reduced value requires all iterations of the first loop to be executed.

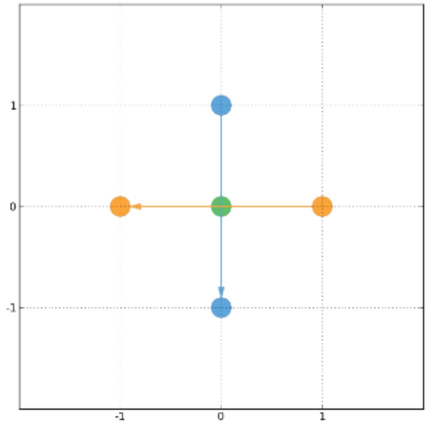
Stage 3: Rolling Analysis

- All spatial references to a variable can be visualized as (another) “reuse” DAG.
  - Vertices = Spatial references
  - Edges = Child node is “reachable” from parent node, using given loop order and stride

- Vertices with input degree of zero are the first time that point in the iteration space has been visited for this variable; others can be loaded from intermediate storage.



2D Laplace, Stride 1



2D Laplace, Stride 2

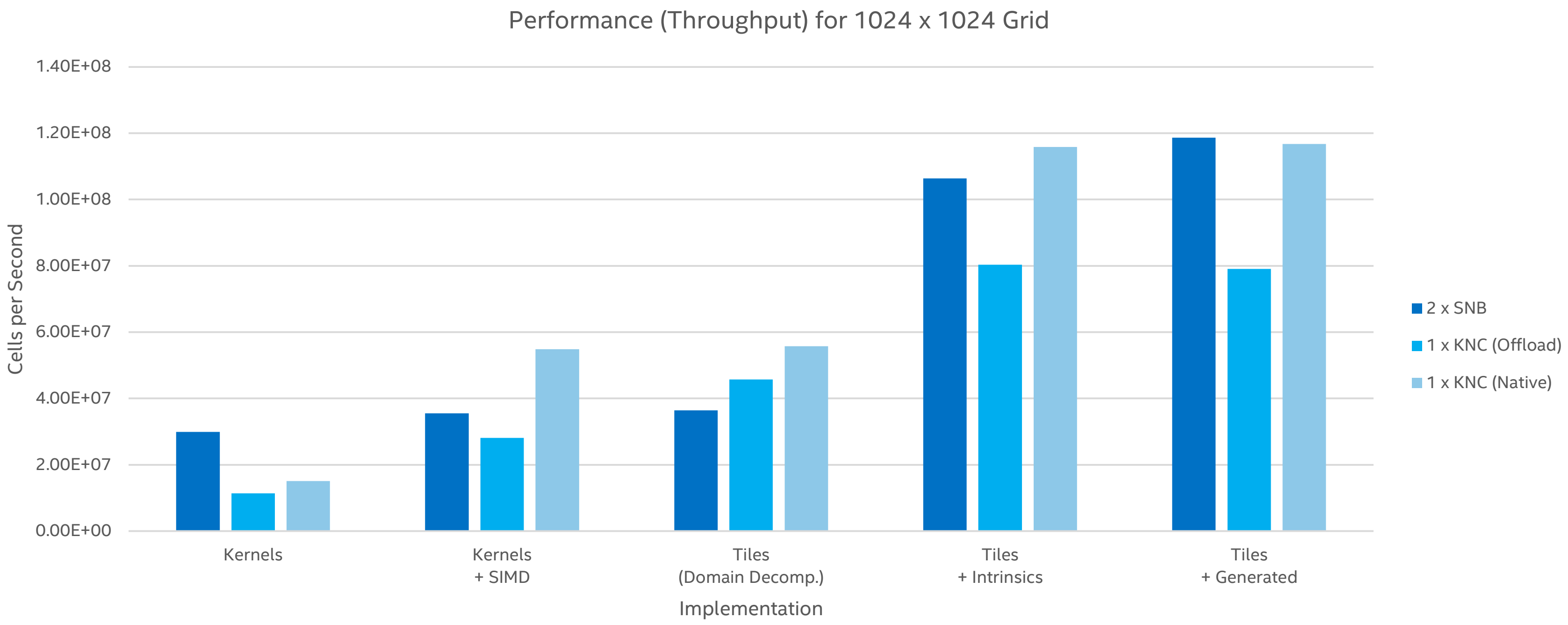
Stage 4: Code Generation

- For each loop nest:
  - Open loop nest.
  - Generate code for children (other loop nests and/or kernel calls).
  - Close loop nest.

- For each variable:
  - Map from global to temporary (e.g. flux[j][i] => tmp_flux[i-istart])

- For vectorization:
  - "#pragma omp simd" if loop has no dependencies; otherwise
  - Strip-mine (and interchange, if necessary) with intrinsic function to rotate buffers

Case Study: CEA's Hydro2D



Generalizing to OpenMP*

- Why are we interested in language extensions?
  - Our YAML framework is ugly and not user-friendly.
  - User code already contains information that must be re-specified in our framework (*e.g.* loop bounds, boundary conditions, program order, variable names)
  - Adoption rate of language extensions appears much higher than that of frameworks (*e.g.* OpenMP SIMD extensions vs Threading Building Blocks)

- We propose to extend OpenMP 4.5 task syntax to specify dependencies between iterations of different loops.

Proposed Language Extensions – Draft

```
#pragma omp pipeline \
depend(inout:cell) intermediate(flux_x) \
iterators(j:j_itf,j_cell,i:i_itf,i_cell)
{

  #pragma omp pipeline block \
   depend(inout:cell:*,*)
  initialize_boundary_conditions(cell);

  #pragma omp pipeline loop simd collapse(2) \
   depend(in:cell:j,i-1) depend(in:cell:j,i) depend(out:flux_x:j,i)
  for (int j_itf = jstart; j_itf < jend; ++j_itf)
  {
    for (int i_itf = istart; i_itf < iend+1; ++i_itf)
    {
      flux(cell[j_itf][i_itf-1], cell[j_itf][i_itf], flux_x[j_itf][i_itf]);
    }
  }

  #pragma omp pipeline loop simd collapse(2) reduction(max:maxCell[j]) \
   depend(in:flux_x:j,i) depend(in:flux_x:j,i+1) depend(out:cell:j,i)
  for (int j_cell = jstart; j_itf < jend; ++j_itf)
  {
    for (int i_cell = istart; i_itf < iend; ++i_itf)
    {
      integrate(flux_x[j_cell][i_cell], flux_x[j_cell][i_cell+1],
                cell[j_cell][i_cell]);
      maxCell[j_cell] = max(maxCell[j_cell], cell[j_cell][i_cell]);
    }
  }
}
```

pipeline
A region containing one or more pipeline stages.

pipeline loop/block
Marks a loop or structured block as a pipeline stage.

intermediate(*list*)
Declares one or more storage locations used only to pass data between pipeline stages.

depend*(dependence-type : list : vec)*
Enforce constraints on the scheduling of loop iterations in different stages of the same pipeline region.

New/extended keywords highlighted in red.

Summary

- Produced a prototype analysis + code generation tool for “rolling updates”.
- Impressive performance results for real-life benchmark.
- Future work:
  - Optimization heuristics (e.g. kernel fusion, redundant compute, halo size)
  - Compiler/language integration

- If your code matches the following criteria, please e-mail us (or talk to me):
  - Multiple parallel/vector loops over a single domain.
  - Local, known (i.e. structured) dependencies between domain elements.

Legal Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel, Xeon, Xeon Phi and the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

intel®