Y.A.S.K.

Yet Another Stencil Kernel

Chuck Yount, Principal Engineer
Intel Corporation

April 21, 2017

chuck.yount@intel.com

Outline

Introduction

- Overview
- Example stencils and performance on Intel® Xeon® and Intel® Xeon Phi™ processors

Using YASK

- Build and test
- Output
- Use model
- Run-time options: hierarchy sizes, MPI, wave-front blocking
- Stencil, vectorization, loop, and advanced customization
- Collaboration opportunities

YASK Software architecture

- Vector folding and the fold builder
- Loop-code generator
- Memory accessor
- Debug output

Introduction to YASK

Overview

YASK: Yet Another Stencil Kernel

- Goal: facilitate exploration of the stencil-performance design space for Intel® Xeon® or Intel® Xeon Phi™ processors supporting the AVX, AVX2, or AVX-512 instruction sets

Features

- Supports trade-off studies for coding options for
  - Vector-folding
  - Cache blocking
  - Memory layout
  - Loop construction
  - Temporal wave-front blocking
  - MPI halo exchanges
  - And more
- Is a collection of C++ code, code-generators and other scripts

Example 1: Iso3dfd stencil

Description

- Finite-difference code found in seismic-imaging software used by energy-exploration companies to predict the location of oil and gas deposits

Performance* on 1024^3 problem size per node

- Intel® Xeon Phi™ processor 7250 ("Knights Landing")
  - 17.3 GPoints/sec on one node (68 cores)
- Intel® Xeon® processor E5-2697 v4 ("Broadwell")
  - 5.36 GPoints/ses on one node (2 sockets of 18 cores each)
- Intel® Xeon® processor E5-2699 v3 ("Haswell")
  - 4.55 GPoints/sec on one node (2 sockets of 18 cores each)

*Observed performance for illustration and comparison; not guaranteed.
Image from https://commons.wikimedia.org/wiki/File:PlatformHolly.jpg. Public domain--U.S. DoE.

Iso3dfd build and run "recipes"

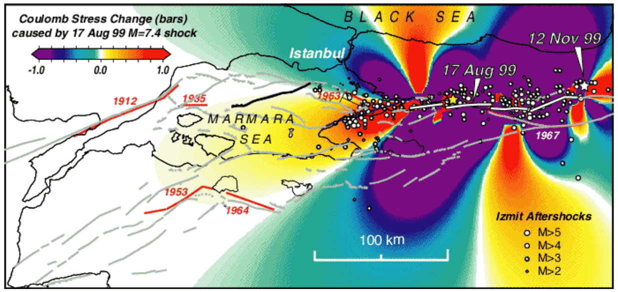
Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz, 16GB MCDRAM flat mode

- `make clean; make stencil=iso3dfd arch=knl`
- ```
  bin/yask.sh -stencil iso3dfd -arch knl -- -d 1024
  …
  best-throughput (point-updates/sec):    17.3832G
  best-throughput (est-FLOPS):            1.06038T
  ```

2-socket Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz

- `make clean; make stencil=iso3dfd arch=hsw`
- ```
  bin/yask.sh -stencil iso3dfd -arch hsw -ranks 2 -- \
  -dx 512 -dy 1024 -dz 1024
  …
  best-throughput (point-updates/sec):    5.36613G
  best-throughput (est-FLOPS):            327.334G
  ```
- Note that half of the x-domain-size is specified for each of the two ranks so that the overall size is 1024^3
- The same recipe may be used for the Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz (output not shown)

Example 2: AWP stencil



Description

- Primary compute kernel for the Anelastic Wave Propagation—ODC earthquake simulator: http://hpgeoc.sdsc.edu/AWPODC
  - Consists of 26 grids in a staggered-grid formulation

Performance* on $1024^2 \times 128$ problem size per node

- Intel® Xeon Phi™ processor 7250 (“Knights Landing”)
  - 23.3 GPoints/sec on one node (68 cores)
- Intel® Xeon® processor E5-2697 v4 (“Broadwell”)
  - 7.56 GPoints/sec on one node (2 sockets of 18 cores each)
- Intel® Xeon® processor E5-2699 v3 (“Haswell”)
  - 6.35 GPoints/sec on one node (2 sockets of 18 cores each)

*Observed performance for illustration and comparison; not guaranteed.
Image from https://commons.wikimedia.org/wiki/File:Izmit_11-12-99.gif. Public domain--U.S.G.S.

AWP build and run "recipes"

Intel® Xeon Phi™ CPU 7250 @ 1.40GHz, 16GB MCDRAM flat mode

- `make clean; make stencil=awp arch=knl`
- ```
  bin/yask.sh -stencil awp -arch knl -- \
  -dx 1024 -dy 1024 -dz 128
  …
  best-throughput (prob-size-points/sec): 1.55341G
  best-throughput (point-updates/sec):    23.3012G
  best-throughput (est-FLOPS):            419.421G
  ```

2-socket Intel® Xeon® CPU E5-2697 v4 @ 2.30GHz

- `make clean; make stencil=awp arch=hsw`
- ```
  bin/yask.sh -stencil awp -arch hsw -ranks 2 -- \
   -dx 512 -dy 1024 -dz 128
  ...
  best-throughput (prob-size-points/sec): 504.039M
  best-throughput (point-updates/sec):    7.56059G
  best-throughput (est-FLOPS):            136.091G
  ```
- Note that half of the x-domain-size is specified for each of the two ranks so that the overall size is $1024^2 \times 128$
- The same recipe may be used for the Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz (output not shown)

Using YASK

Initial build and test

Install

- Download the code from the 'GIT REPO' link at https://01.org/yask
- Install all the prerequisites from the README file

Build and run the default test program

- Type 'make -arch *name* -stencil *name*' per the README file
- Run the program using the bin/yask.sh script
  - Run natively on Xeon™ or Xeon Phi™ processors (KNL)
  - Use the -mic option to run on a Xeon Phi™ coprocessor (KNC)
  - Run under SDE to emulate hardware you don't have
- If it doesn't build and/or run, check the prerequisites

Typical run and output

Settings are printed

- Stencil name
- Hierarchical sizes (spatial and temporal): rank, region, block, cluster, and vector
- Other miscellaneous compile-time and run-time settings

A number of trials (default=3) is run

- Time and throughput (points per sec) are printed
- FP-rate is estimated based on number of FP ops in original scalar spec
- Best result across the trials is re-printed

Validation

- If the '-v' option was used, a non-vectorized, non-tiled version of the code is run, the results are compared, and 'PASSED' or 'FAILED' is printed
- Validation is slow; run with a small problem size!
- If you get near-miss errors during validation, it may be due to rounding error instead of a bug; try building with "real_bytes=8" to check

Use model

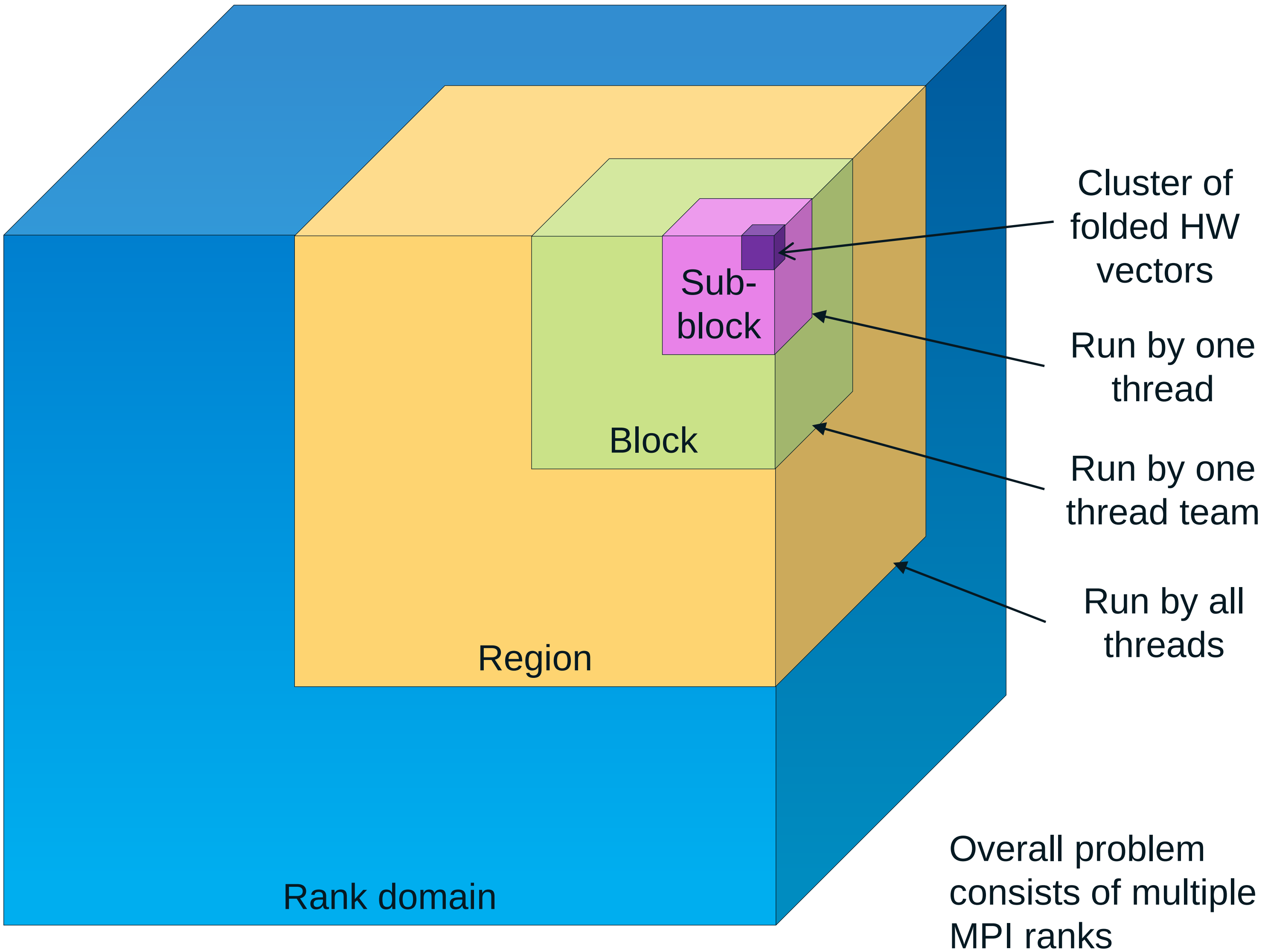
Goal

- YASK is a tool for exploring the stencil design space
- It is not a library (yet)

Typical usage model

- Identify stencil(s) used in your application
- Use existing stencil(s) in YASK or write your own
- Use YASK to find well-performing parameters
- Integrate the stencil code back into your application

Tiling applied at multiple levels



Common run-time options

Settings controlled from the ‘yask.sh’ script

- Binary selection via ‘arch’ option
- Number of MPI ranks
- Which Xeon Phi coprocessor or other host to use
- Run with ‘-h’ option to get help

Settings controlled from the ‘yask.*stencil.arch*.exe’ binary (passed through from the yask.sh script)

- Spatial/temporal rank-domain size (overall problem if not using MPI): -d*
- Spatial/temporal region size (used for temporal wave-front blocking): -r*
- Block size: -b*
- Sub-block size: -sb*
- Padding: -p*
  - Used to fine-tune data alignment across rows and columns
  - The specified value is added to the halo size during memory allocation
- Number of trials: -t
- Enable validation: -v
- Run with ‘-h’ option to get more help on these and other options

Enabling MPI

Scope

- The MPI implementation in YASK exchanges halos with all neighbors in all directions
  - For 3D problems, this can be up to 26 neighbors (3^3-1)
  - For 4D problems, this can be up to 80 neighbors (3^4-1)

Usage

- Compile with MPI enabled using "make mpi=1 …"
- Prefix "yask.sh" with the appropriate MPI command, e.g., mpirun -n 4 -ppn 1
  - Use the "-ranks <n>" option to "yask.sh" as a shortcut to run more than one rank on a single node in the x-dimension only
- Note: the -d* options control the rank size, so the overall problem size increases by the number of ranks (weak scaling)

Enabling temporal wave-front blocking

Purpose

- The temporal blocking in YASK is designed to exploit large shared caches, e.g., when using the Intel® Xeon Phi™ processor in MCDRAM cache mode

Usage

- Temporal wave-front blocking is done using the “regions” level of the hierarchy shown earlier
  - Spatial blocks within each region are evaluated in parallel using OpenMP
  - The time slices within each region are evaluated sequentially to reuse memory
  - Regions are evaluated sequentially to increase shared-cache locality
- Executable run-time options
  - -rt <n> sets the number of time slices in each region
  - -r* <n> sets the spatial size of each region
  - Example: bin/yask.sh -stencil 3axis -arch knl -d 1920 -dt 50 -r 768 -rt 25
  - Note: the default setting of -rt is one (1), and the default setting of -r is zero (0), which means the region size is the same as the rank size.

Stencil customization

Stencil Type

- Use the ‘stencil=*stencil-name*’ argument to the make command to select a stencil (required)
  - The *stencil-name* string is passed to the foldBuilder tool
- Examples of current provided stencils
  - ‘iso3dfd’: an isotropic acoustic wave equation
  - ‘3axis’, ‘9axis’, ‘3plane’, and ‘cube’: common 3D symmetric shapes (as used in the vector-folding paper)
  - ‘ave’: the simple 27-pt stencil from the miniGhost benchmark
  - ‘awp’: a simplified version of AWP-ODC earthquake simulation stencils
- Write your own by modifying code in src/foldBuilder
  - Implement the StencilBase interface using the stencils/*Stencil.hpp files as examples

Stencil customization (cont.)

Stencil size

- Use the ‘radius=*n*’ argument to the make command
  - The *n* value is passed to the ‘foldBuilder’ tool
  - Default is different for various stencils and ignored for some
    - See Makefile for current defaults
- Write your own by modifying code in src/foldBuilder
  - Follow the existing examples to pass the ‘radius’ parameter to your stencil code if applicable

Other parameters

- If you’re developing your own stencil, you can add more parameters similar to the ‘radius’ one

Stencil customization (cont.)

Advanced

- The provided stencils assume uniformity across the entire 3D grid
  - The ‘foldBuilder’ tool evaluates the stencil code only from the origin to the extent of a vector
- Some stencil applications require special code at boundaries or other conditions
  - A mechanism for handling conditions that specify sub-domains is under development
  - See the awp_elastic stencil for an example
  - Since this is under development, the syntax may change

Vector-folding introduction

Concept

- Store small 2D or 3D block of data into each vector
- Pros: reduces memory BW requirements compared to traditional 1D in-line vectors
- Cons: requires data pre-conditioning (element rearranging) and additional shift and blend operations preceding SIMD arithmetic operations

Results

- Significant speedup shown on Intel® Xeon Phi™ Coprocessor
- Combining with loop tiling enables even more speedup

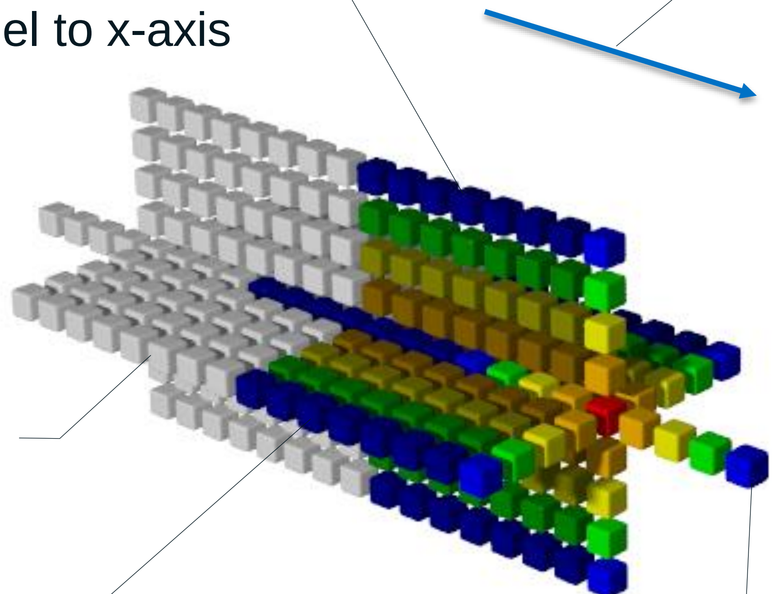
For more information

- Refer to paper on Vector Folding from HPCC 2015

Traditional in-line 1D vectorization

25-point 3D stencil input vectorized using 8-element vectors, each parallel to x-axis

Inner 3D loop iterates in x direction, i.e., *same dimension* as vectorization



Previous iteration

Current iteration

One 8-element vector output per iteration

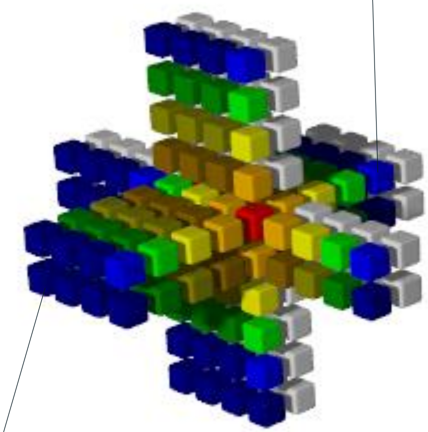
Need to read 17 new cache lines* for each iteration (8 of the 25 vectors overlap in x dimension)

Steady-state memory BW = **17** new cache lines input to calculate each vector of output

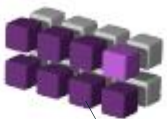
*Assuming cache line size = vector size.

Reduce BW via vector folding

25-point 3D stencil input vectorized using 8-element vectors, *each containing a 4x2 grid in the x-y plane*



Inner 3D loop iterates in z direction, i.e., *perpendicular* to 2D vector



One 8-element (4x2) vector output per iteration

Need to read only 7 new cache lines for each iteration (vectors overlap in x-y dimensions within an iteration and in z dimension between iterations)

Steady-state memory BW = **7** new cache lines input to calculate each vector of output: **2.4x lower** than in-line

Vector-folding customization

Vector fold

- Use the fold='x=*n*,y=*n*,z=*n*' argument to the make command to control how much vectorization is done in each dimension
  - The values are passed to the ‘foldBuilder’ tool
  - Example: make fold='x=1,y=2,z=8' generates code using a 1x2x8 fold
  - Make sure the product of the fold lengths equals the vector size of the target architecture and FP precision (single or double)

Vector cluster

- Use the cluster='x=*n*,y=*n*,z=*n*' argument to the make command to control how many vectors are calculated and output simultaneously
  - The values are passed to the ‘foldBuilder’ tool
  - The default is 1x1x1, or one HW vector
  - This essentially implements loop unrolling in multiple dimensions

Loop-structure customization

The ‘gen-loops.pl’ script creates the loop-control code

- There are 4 loop-control codes
  - ‘Outer’ loops break the whole problem into OpenMP regions (typically, only one OpenMP region is used)
  - ‘Region’ loops break each OpenMP region into cache blocks
  - ‘Block’ loops iterate over each vector cluster in a cache block
  - ‘Halo’ loops are used for copying data for halo exchanges

Usage

- See the Makefile for default invocations or run ‘make -n’
- Run ‘./gen-loops.pl’ without any parameters to get help on more options: index ordering, OpenMP scheduling, etc.
- Run the script before the make command or specify the *LOOP_ARGS variables in the make command to override

Misc. advanced customization

More compile-time options to the make command

- Use ‘real_bytes=*n*’ to set the size of a float: n=4 for single-precision or n=8 for double-precision (default=4; 8 for ‘ave’ stencil)
- Use ‘EXTRA_MACROS='*macro-settings*'’ set other CPP macros
  - ‘PFDL1=*n1* PFDL2=*n2*’ to change the prefetch distances; only used in the prefetch code generated from ‘gen-loops.pl’, not in compiler-generated prefetch code
  - Example: ‘make EXTRA_MACROS=‘PFDL2=15'’
  - See *.hpp for macro definitions
- Run ‘make echo-settings’ to see other make variables

Automatic Tuning

YASK includes a genetic-algorithm-based automatic tuning utility

- Tunes compile-time and run-time options and parameters to maximize throughput
- Run ‘stencil-tuner.pl -h’ to get tuner options
- Example run:
  - Tune the parameters for the ‘iso3dfd’ stencil on the KNL architecture, constraining the memory usage between 14 and 16 GiB
  - ./stencil-tuner.pl -stencil=iso3dfd -arch=knl -mem=14-16
- Results
  - The settings and throughput of each trial is saved into a .csv file
  - Use the ‘stencil-tuner-summary.csh’ script to find the best result
    - Can use after the tuner exits or anytime while it’s running
  - Each result includes the “make” and “run” commands
    - Often these commands will include parameter settings which are not strictly necessary to obtain the best results
    - You may wish to experiment manually with the best outcome to find simpler commands that produce similar results

Collaboration

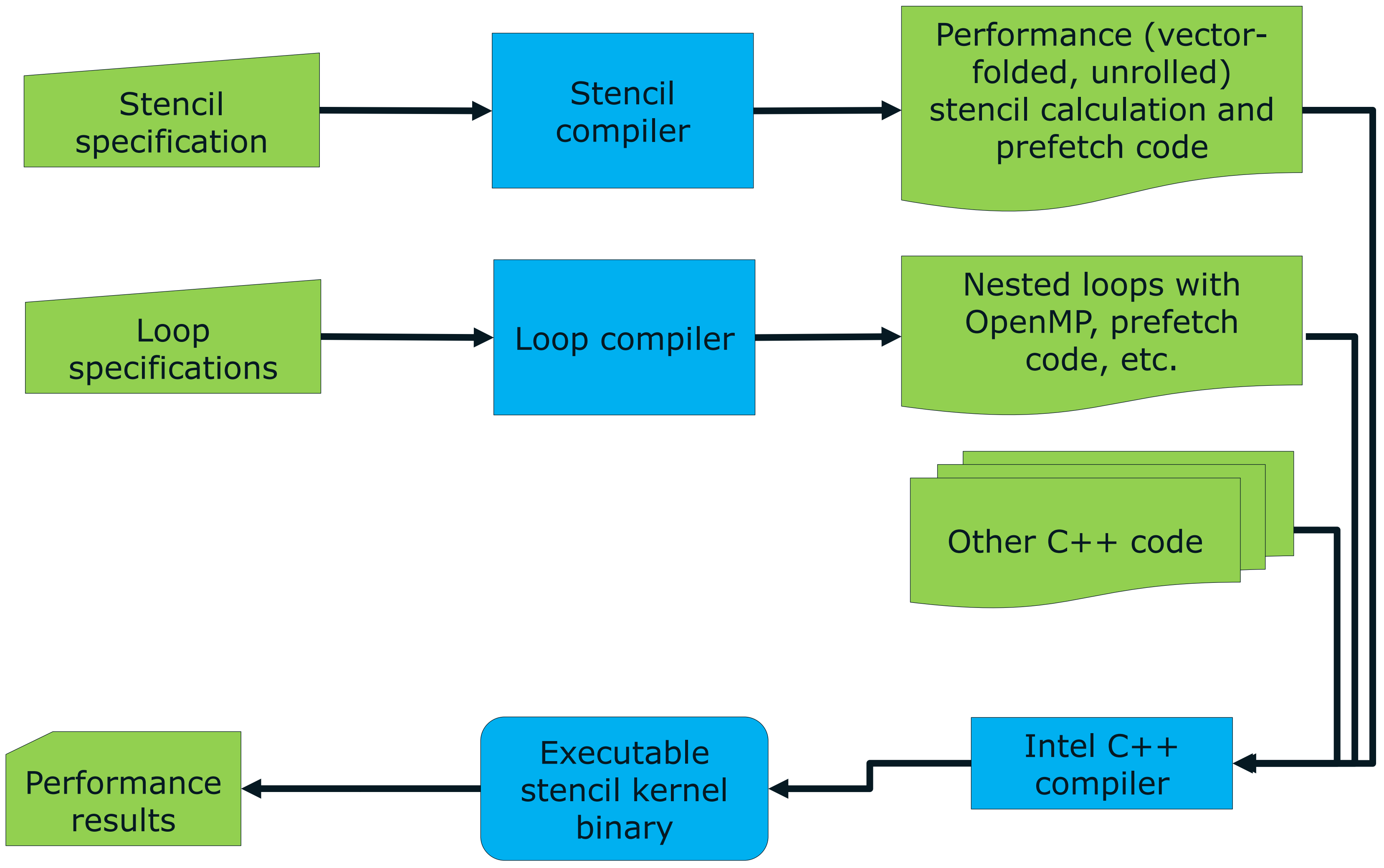
Use the blog at https://01.org/yask to ask and/or answer questions

Submit useful changes for review via github

Contact the author of this presentation for further collaboration opportunities

YASK Software Architecture

High-level components



Fold-builder code generator

Goal: automate the tedious and error-prone process of creating high-performance stencil code

Input

- Inherit from a C++ abstract ‘StencilBase’ class to create a new stencil type
- Define the grid(s) to be used and the names of their dimensions, e.g., “t”, “x”, “y”, “z”
- Implement the ‘define’ method to define how one point in each grid is calculated from others
- Use loops, functions for coefficients, recursion, etc.

Process

- Compile code into fold-builder executable
- Run executable, specifying any stencil parameters (e.g., order), target architecture, etc.
- Code generator evaluates the ‘define’ method to create an abstract syntax tree (AST)
- AST is traversed, and optimized code is output

Output

- Efficient function to calculate stencil
  - Unrolled loops, intrinsics to construct unaligned vectors of points, etc.
  - Calls to memory accessor object
- Functions for prefetching to L1 and L2

Example input stencil code

```
class Iso3dfdStencil : public StencilOrderBase {
…
 INIT_GRID_4D(pressure, t, x, y, z),
 INIT_GRID_3D(vel, x, y, z)
…
 virtual void define(const IntTuple& offsets) {
…
        // start with center value multiplied by coeff 0.
        GridValue v = pressure(t, x, y, z) * coeff(0);

        // add values from x, y, and z axes multiplied by the
        // coeff for the given radius.
        for (int r = 1; r <= _order/2; r++) {

            // Add values from axes at radius r.
            v += (
                  // x-axis.
                  pressure(t, x-r, y, z) +
                  pressure(t, x+r, y, z) +

                  // y-axis.
                  pressure(t, x, y-r, z) +
                  pressure(t, x, y+r, z) +

                  // z-axis.
                  pressure(t, x, y, z-r) +
                  pressure(t, x, y, z+r)

                  ) * coeff(r);
        }

        // finish equation, including t-1 and velocity components.
        v = (2.0 * pressure(t, x, y, z))
            - pressure(t-1, x, y, z) // subtract pressure from t-1.
            + (v * vel(x, y, z));       // add v * velocity.

        // define the value at t+1 to be equal to (really, an approximation of) v.
        pressure(t+1, x, y, z) EQUALS v;
    }
```

Declare 2 grids: 4D “pressure” and 3D “vel”

Write function to define equation for “pressure” update

The final equation uses declarative (not imperative) style via the EQUALS macro

Loop-code generator

Script that generates code for nested loops

- Input: Very simple DSL (domain-specific language)
  - omp loop(y) { loop(x, z) { calc(stencil); } }
  - Can easily change loop types, index ordering
- Output: C++ code to be included in function bodies
  - Loops annotated with OMP as requested
  - Inner loop might generate several loops, e.g.,
    - Prefetch L2
    - Prefetch L1
    - Calculate stencil and prefetch L2 and L1
    - Calculate and prefetch L1 only to avoid over-prefetching L2

Memory accessor

- C++ classes to allocate and access 3D & 4D matrices of vectors of floats or doubles
  - Construction specifies ‘w, x, y, z’ dimensions and padding sizes; padding includes halos (use ‘w’ for 4D problems)
  - Read and write access via methods: per vector for highest speed; per element for scalar code.
- Actual memory layout is encapsulated and defined via inheritance
  - Map indices to 1-D mem address
    - 24 simple permutations of minor-to-major ordering in 4D
    - More complex mappings possible, e.g., tiling, space-filling curves
  - ‘t’ dimension is used for time and is template for performance
- Uses concrete inheritance to allow inlining
  - Gives compiler full access to memory-layout formula
  - Allows common sub-expression elimination and other optimizations

Debug features

Can enable or disable various output by setting macros and rebuilding, e.g.,

- TRACE: print each stencil calculation
- TRACE_MEM: print each matrix read, write, prefetch, eviction
- TRACE_INTRINSICS: print before-and-after each permutation

Built-in memory-access tracker

- Models an infinite L1 or L2 cache
- Tracks reads, writes, prefetches, evictions
- Reports any un-prefetched read or un-read prefetch
- Reports summary stats
- Very useful for debugging prefetch code

Example cache-model output

```
modeling cache...
cache L2: redundant prefetch of 0x2aaabfa45a40 at line 193 after a read at line 85.
cache L2: redundant prefetch of 0x2aaabfa45a80 at line 193 after a read at line 85.
cache L2: redundant prefetch of 0x2aaabfa45a40 at line 195 after a prefetch at line 193.
cache L2: redundant prefetch of 0x2aaabfa45a40 at line 196 after a prefetch at line 195.
…
done modeling cache...
cache L2: read of 0x2aaabf9c3240 from line 85 without any eviction.
cache L2: read of 0x2aaabf9c3280 from line 85 without any eviction.
…
cache L2: prefetch of 0x2aaabfa53b00 from line 318 without any read.
cache L2: prefetch of 0x2aaabfa53b40 from line 318 without any read.
…
cache L2 stats:
 cur size = 324714 lines (19.8190 MB).
 max size = 324714 lines (19.8190 MB).
 ave size = 185126 lines (11.2992 MB).
 num reads = 722400.
  num reads of missing lines = 0.
  num lines read but never evicted = 321700.
 num prefetches = 1458800.
  num prefetches of lines never subsequently read = 3014.
  num prefetches of lines already in cache = 404686.
 num evictitions = 0.
  num evictions to non-existant lines = 0.
 num prefetches into L1 = 729400.
  num prefetches into L1 of missing lines = 0.
```

Legal Disclaimer & Optimization Notice

Optimization Notice

intel