



# Introduction to YASK

Chuck Yount
Principal Engineer, Intel Corporation
chuck.yount@intel.com

Version 2.09.00
May 18, 2018

# Outline

Introduction

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

Using YASK

- Simple build and test
- Use model
- Run-time options: domain and tile sizes, MPI, etc.
- Compile-time options: stencil, vectorization, ec.
- Where to get help and collaboration opportunities

# Introduction to YASK

# Overview

YASK: Yet Another Stencil Kernel

- Goal: automatically generate HPC stencil kernels from a simple domain-specific language and tune their performance for Intel® Xeon® and Intel® Xeon Phi™ processors supporting the AVX, AVX2, or AVX-512 instruction sets

Features

- Supports trade-off studies for
  - Vector-folding
  - Cache blocking
  - Memory layout
  - Rectilinear scanning-order
  - Temporal wave-front blocking
  - MPI topologies
  - And more
- Creates a library that may be integrated into a larger application

# Version 2 vs. 1 comparison

| Feature | Version 1.0.0 | Version 2.0.0 |
|---|---|---|
| Release date | Aug 24, 2017 | Nov 28, 2017 |
| APIs | None | C++ and Python APIs for both kernel and compiler |
| Libraries | None (monolithic executable) | C++ libs and Python modules for both kernel and compiler |
| Dimensions | Only 3D (x, y, z) stencils allowed | Stencils and grids can be any number of dims with any dim names |
| Auto-tuner | External script only | External script plus internal block-size tuner |

# Version 2.1-2.6 highlights

| Version | Date | Major feature |
|---|---|---|
| 2.01.00 | Dec 1, 2017 | Support for creating efficient Cerjan-style boundary layers with 1-D arrays |
| 2.02.00 | Mar 7, 2018 | APIs now throw C++ and Python exceptions instead of terminating |
| 2.03.00 | Mar 26, 2018 | Added scratch-grids, multi-dimensional temporary vars for stencil equations |
| 2.04.00 | Mar 31, 2018 | Ability to use temporal wave-fronts across multiple MPI ranks (now requires g++ 4.9 or later) |
| 2.05.04 | Apr 12, 2018 | Ability to specify NUMA node for each grid separately |
| 2.06.02 | Apr 28, 2018 | Add compiler APIs for creating sub-domains and manual dependency graphs. |

# Version 2.7+ highlights

| Version | Date | Major feature |
|---|---|---|
| 2.07.03 | May 1, 2018 | Add kernel APIs for setting regions for temporal wave-front tiling and compiler API for specifying full grid-index expressions |
| 2.08.03 | May 7, 2018 | Provided operator overloading in the API interface |
| 2.09.00 | May 18, 2018 | Improvements to decrease compile time and binary size; example stencils now in .cpp (not .hpp) files |

# 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

## Single-node performance*

- Intel® Xeon Phi™ processor 7250 (“Knights Landing”)
  - 16.1 GPoints/sec (68 cores)
- Intel(R) Xeon(R) Gold 6148 CPU (“Skylake”)
  - 9.6 GPoints/sec (2 sockets of 20 cores each)

*Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance measured by Intel, May 2018.
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 (quad/flat mode)

- `make clean; make stencil=iso3dfd arch=knl`
- `bin/yask.sh -stencil iso3dfd -arch knl -d 1024`

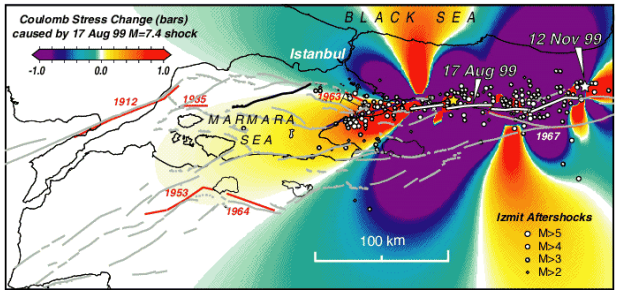
```
…
best-throughput (est-FLOPS):        986.256G
best-throughput (num-points/sec):   16.1681G
```

## 2-socket Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz (SMT on)

- `make clean; make stencil=iso3dfd arch=skl`
- `bin/yask.sh -stencil iso3dfd -arch skl \`
  `-ranks 2 -d 1024`

```
…
best-throughput (est-FLOPS):        590.638G
best-throughput (num-points/sec):   9.68259G
```

# 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

## Single-node performance*

- Intel® Xeon Phi™ processor 7250 (“Knights Landing”)
  - 1.4 GPoints/sec (68 cores)
- Intel® Xeon(R) Gold 6148 CPU (“Skylake”)
  - 857 MPoints/sec (2 sockets of 18 cores each)

*Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Performance measured by Intel, May 2018.
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 (quad/flat mode)

- `make clean; make stencil=awp arch=knl`
- ```
  bin/yask.sh -stencil awp -arch knl \
  -dx 1024 -dy 1024 -dz 128
  …
  best-throughput (est-FLOPS):        406.335G
  best-throughput (num-points/sec):   1.41089G
  ```

## 2-socket Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz (SMT on)

- `make clean; make stencil=awp arch=skl`
- ```
  bin/yask.sh -stencil awp -arch skl -ranks 2 \
  -dx 1024 -dy 1024 -dz 128
  ...
  best-throughput (est-FLOPS):        246.853G
  best-throughput (num-points/sec):   857.13M
  ```

# Using YASK

# Initial build and test

Install

- Download the code from https://github.com/intel/yask
- Install all the prerequisites from the README file

Build and run a test program on a single node

- Follow the recipes from the previous section
- In addition to the architectures “knl” and “skl” used in the recipes, you can use “snb”, “ivb”, or “hsw” depending on your processor’s architecture
- When running, specify “-ranks *N*”, where *N* is the number of sockets on your node
- If it doesn’t build and/or run, look for error message and check the prerequisites

# Typical run and output

Settings are printed

- Stencil name
- Hierarchical tile sizes: rank, region, block, sub-block, cluster, and vector
- Other miscellaneous compile-time and run-time settings

The internal auto-tuner is run to determine a good block size

- The best block-size is printed when done

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 is 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! (This is set with '-v' by default)
- 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

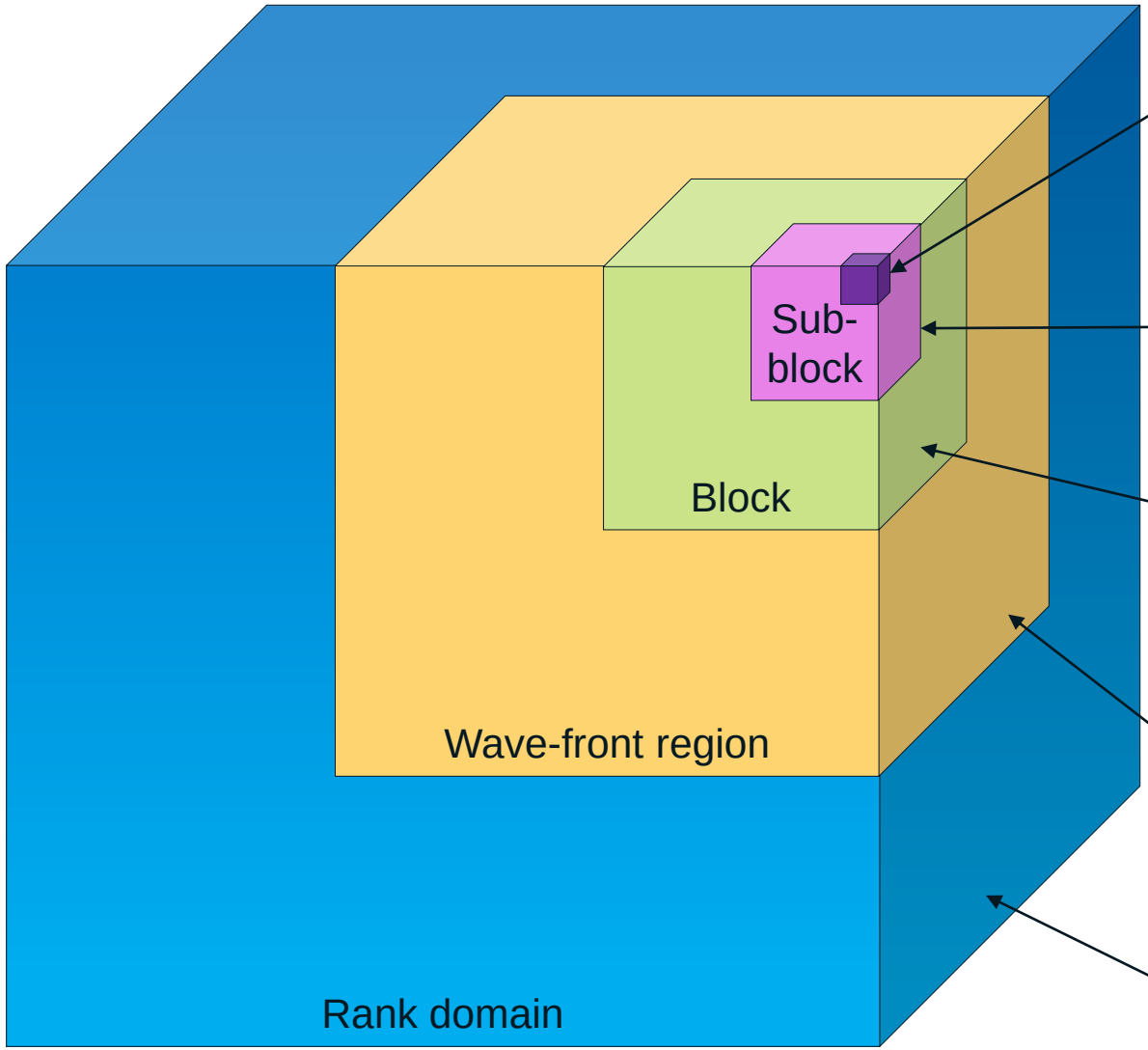
Goals: YASK is a software framework for

- Exploring the stencil design space
- Creating high-performance libraries to link into larger applications

Typical usage model

- Identify stencil(s) used in your application
- Write stencil code using existing stencil(s) in YASK as examples
- Use included YASK script to find well-performing parameters
- Integrate the library into your application using C++ or Python bindings
  - Be aware that YASK performance relies on non-tradition data layout, so your application will need to access data stored in YASK-provided grid objects via the provided APIs

# Tiling applied at multiple levels




Vector-folded blocks are run sequentially by each thread

Sub-blocks are run in parallel by the threads in a team

Blocks are run in parallel by thread teams

Wave-front regions are run sequentially in each rank and allow temporal blocking

Rank domains are run in parallel by MPI ranks

# Common run-time options

Settings controlled from the 'bin/yask.sh' script

- Binary selection via '-stencil' and '-arch' options
- MPI configuration
- 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 (if not using block-size auto-tuning): -b*
- Sub-block size (if not using block-size auto-tuning): -sb*
- Number of trials: -t
- Run a validation test: -v
- Run with '-h' option to get more help on these and other options

# Using 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)
- MPI is recommended for both multiple network nodes and multiple sockets or dies to isolate NUMA nodes

## Usage

- Run "yask.sh" with the appropriate MPI options
  - 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
  - Use the "-mpi_cmd <command>" option and supply rank layout options (-nr*) for all other configurations
    - Example: `bin/yask.sh -mpi_cmd 'mpirun -f hostfile' -stencil iso3dfd -arch snb -nry 2 -nrz 2 -d 512`
- Notes:
  - The -d* options control the domain size *on each rank*, so the overall problem size increases with the number of ranks (weak scaling)
  - To disable MPI in the binary, build with `make mpi=0 …`

# Enabling temporal wave-front blocking

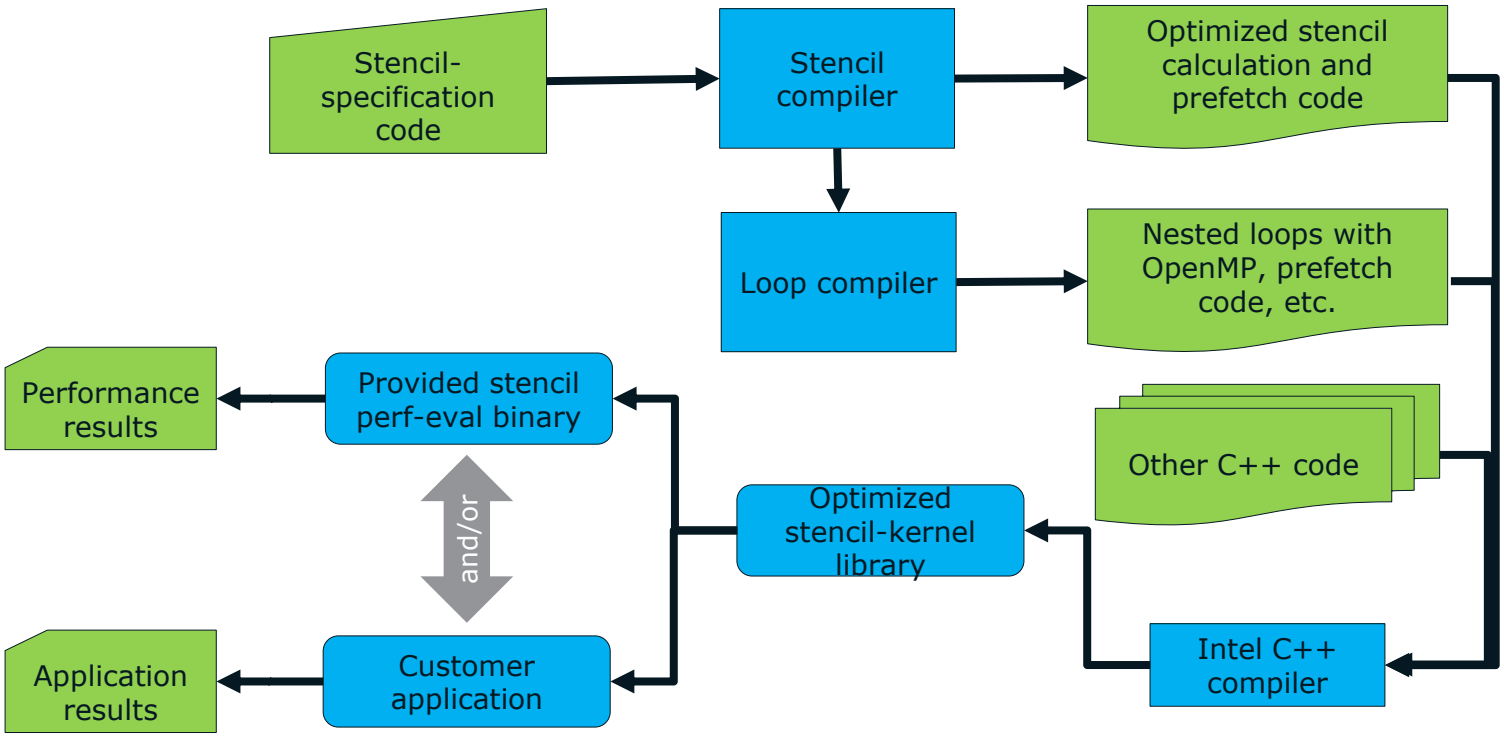
Purpose

- 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 or an Intel® Xeon® processor with a large L3

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
  - If MPI is used, ranks are still evaluated in parallel
- 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, effectively disabling wave-fronts

# High-level software components



# 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 YASK compiler
- Examples of current provided stencils
  - 'iso3dfd': an isotropic acoustic wave equation
  - 'awp': a simplified version of AWP-ODC earthquake simulation stencils provided by Univ. of CA, San Diego (UCSD)
  - 'fsg': a full-staggered grid seismic stencil, provided by Polytechnic University of Catalonia (UPC)
  - '3axis', '9axis', '3plane', and 'cube': common 3D symmetric shapes (as used in the vector-folding paper)
- Write your own by adding a file in src/stencils
  - Implement the StencilBase interface using the *.cpp files as examples
  - Alternatively, implement the StencilRadiusBase interface if you want to inherit the "radius" parameter
  - Older versions of YASK defined stencils in *.hpp files, but the definitions are identical except for the *.cpp files requiring `#define USE_INTERNAL_DSL` at the top

# Stencil customization (cont.)

Stencil size

- Use the 'radius=*n*' argument to the make command
  - The *n* value is passed to the YASK compiler
  - Used only for classes that inherit from StencilRadiusBase
    - Default is different for various stencils
    - See src/kernel/Makefile for current defaults

# Stencil customization (cont.)

Sub-domains (work in progress)

- Most example stencils assume uniformity across the entire 3D grid
- 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

# Example input stencil code

```
class Iso3dfdStencil : public StencilRaduisBase {

    MAKE_STEP_INDEX(t);
    MAKE_DOMAIN_INDEX(x);
    MAKE_DOMAIN_INDEX(y);
    MAKE_DOMAIN_INDEX(z);
    MAKE_MISC_INDEX(r);

    MAKE_GRID(pressure, t, x, y, z);
    MAKE_GRID(vel, x, y, z);
    MAKE_GRID(coeff, r);

public: virtual void define(const IntTuple& offsets) {
    GridValue next_p =
       pressure(t, x, y, z) * coeff(0);
    for (int r = 1; r <= _radius; r++)
       next_p += (pressure(t, x-r, y, z) +
                  pressure(t, x+r, y, z) +
                  pressure(t, x, y-r, z) +
                  pressure(t, x, y+r, z) +
                  pressure(t, x, y, z-r) +
                  pressure(t, x, y, z+r))
       * coeff(r);
    next_p = (2.0 * pressure(t, x, y, z))
       - pressure(t-1, x, y, z)
       + (next_p * vel(x, y, z));
    pressure(t+1, x, y, z) EQUALS next_p;
}
```

Declare temporal and spatial indices

Declare grids: 4D “pressure”, 3D “vel”, 1D “coeff”

Write function to define equation for “pressure” update at step t+1

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

# 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™ Processor
- Combining with loop tiling enables even more speedup

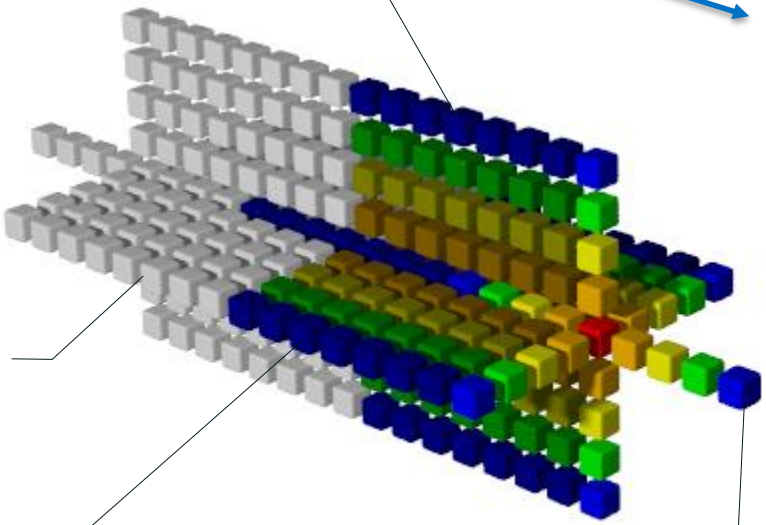
For more information

- 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

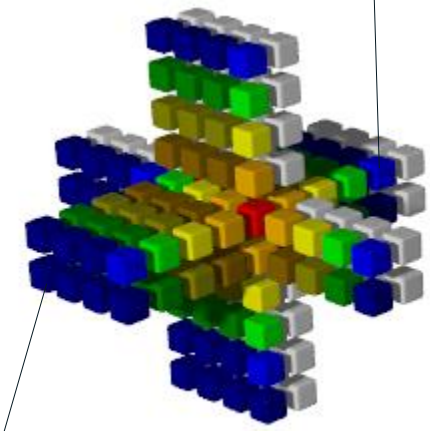
Need to read 17 new cache lines* for each iteration (8 of the 25 vectors overlap in x dimension)

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

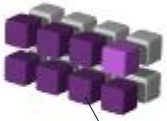
*Assuming cache line size = vector size.

# 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 stencil compiler
  - Example: make fold='x=1,y=2,z=8' generates code using a 1x2x8 fold
  - The product of the fold lengths should equal the vector size of the target architecture and FP precision (single or double)
    - If not, the compiler will adjust the fold using a heuristic algorithm

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 stencil compiler
  - The default is 1x1x1, or one HW vector
  - This essentially implements loop unrolling in multiple dimensions

# Scanning-code customization

The ‘gen_loops.pl’ script generates scanning code for rectilinear solids

- Code is generated for 4 nested solids
  - ‘Outer’ scans break the whole problem into regions
    - Typically, only one region is used unless you’re using wave-fronts
  - ‘Region’ scans break each region into blocks
    - Each block is executed by an OpenMP outer thread “team”
  - ‘Block’ scans break each block into sub-blocks
    - Each sub-block is executed by an OpenMP thread within the block team
  - ‘Sub-block’ scans iterate over each vector cluster in a block
- There are also ‘misc’ scans for other miscellaneous parallel loops

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.
- Specify the *_LOOP_* 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 `pfd_l1=*n*’ and/or ‘pfd_l2=*n*’ to set prefetch distances for L1 and L2; use zero (0) to disable
- Use ‘EXTRA_MACROS='*macro-settings*'’ set other CPP macros
  - See *.hpp for macro definitions
- See the top of the main Makefile and/or run ‘make echo-settings’ to see other make variables

# Debug features

Can enable or disable various output by setting macros via EXTRA_MACROS make var and rebuilding, e.g.,

- CHECK: add lots of bounds-checking assertions
- TRACE: print start of each tile and other status
- TRACE_MEM: print key memory addresses
- TRACE_INTRINSICS: print details of each each permutation and other complex intrinsic functions

All of these will severely impact performance

# APIs

Languages supported

- C++: most efficient
- Python: C++ wrapper generated via SWIG

Documentation

- Open https://rawgit.com/intel/yask/api-docs/html/index.html in any browser
- **The API documentation is also a good reference for how many parts of YASK work, especially grids, even if you're not using the APIs**

Generation

- Run 'make *<normal kernel-building options>* kernel-api' to create the kernel API
- Run 'make compiler-api' to create the stencil-compiler API
  - The compiler API is optional
  - You do not need it if you're writing your stencils like the examples shown earlier
  - It is usually only needed for connecting the YASK compiler to a programmatic stencil-code generator

# Automatic Tuning

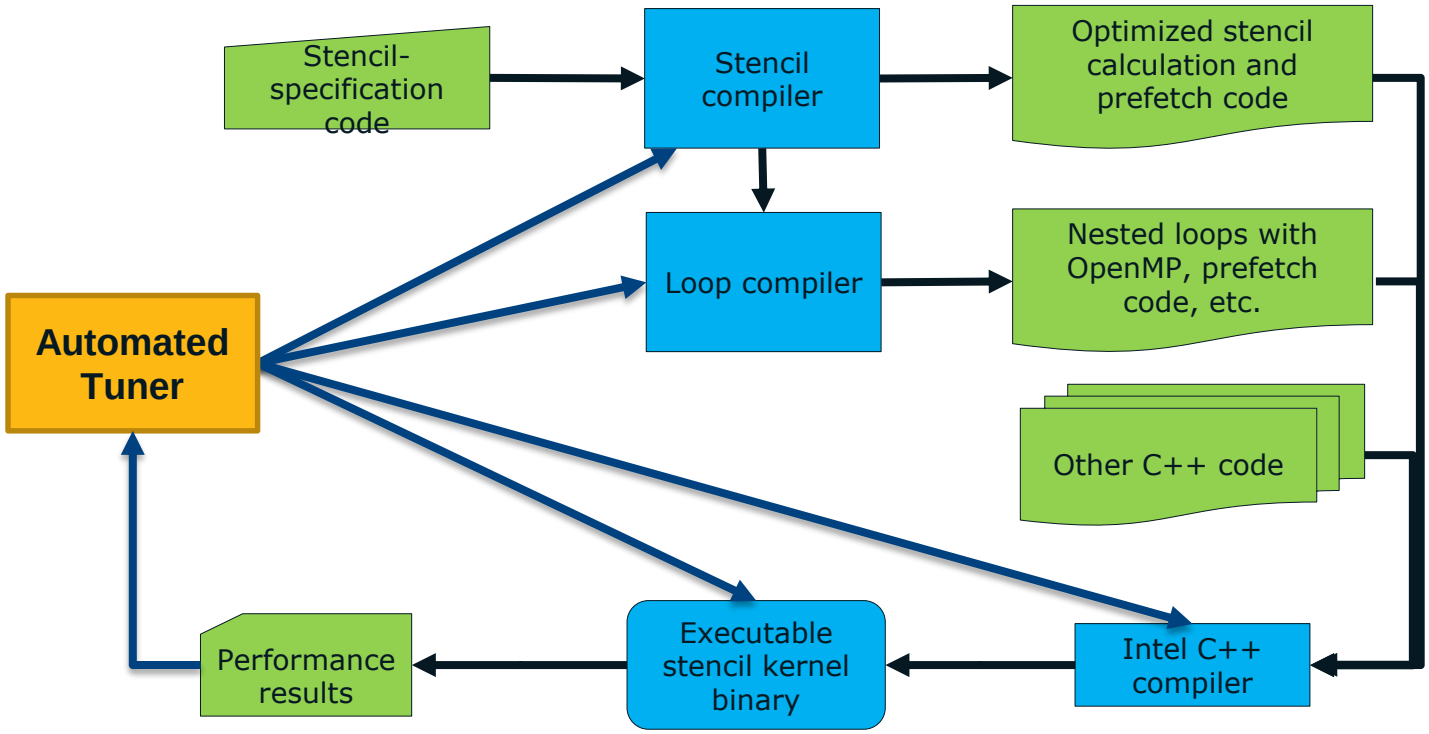
An internal tuner sets the block and sub-block sizes at run-time

- See the binary help and API documentation for information on how to control it

YASK also includes a genetic-algorithm-based tuning utility

- Sets compile-time and run-time options and parameters to maximize performance (points-per-second)
- Run 'bin/yask_tuner.pl -h' to get tuner options and examples
- Output
  - The settings and performance of each trial are saved into a .csv file
  - Use the 'bin/yask_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

# Control and feedback points for tuner



# Getting help or collaborating

Please contact the author via email with

- Questions about usage
- Questions about getting expected performance
- Questions about using YASK in your application
- Requests for academic and conference papers and presentations with more information
- Requests for bibliography entries to use when referencing YASK in papers and presentations
- Offers to collaborate on a project

# Legal Disclaimer & Optimization Notice

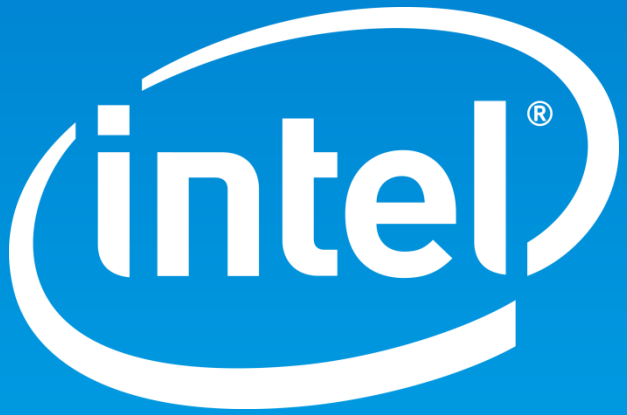
INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel