Developing Stencil Code using the YASK Framework

This tutorial applies to YASK version 3.00.00 (and later versions in most cases)

Chuck Yount
Intel Americas, Inc.

June 21, 2019
© 2019 Intel Corporation
NOTICES AND DISCLAIMERS

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

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. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks.

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. For more complete information visit http://www.intel.com/benchmarks.

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

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.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate. Intel, the Intel logo, and Intel Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as property of others.

© 2018 Intel Corporation.
Outline

Introduction
- What is YASK?
- Motivation and example YASK application
- Scope of this presentation

Basic features and usage
- Download, build, and test
- High-level flow
- Writing a simple stencil in the DSL
- Building the YASK library
- Testing and tuning your stencil
- Multi-node usage via MPI

Using the APIs
- Access, terms, and basic usage
- Advanced APIs and exceptions

Advanced stencils and tuning
- Multiple stencils
- Boundary regions (sub-domains)
- Temporal conditions
- Vector folding
- Cluster, bundles, and packs
- FP precision and prefetching
- 3 layers of temporal tiling
- Nested OpenMP in sub-blocks
- More on the auto-tuners

Wrap-up
- Current work
- Further reading and call to action
INTRODUCTION
YASK: “Yet Another Stencil Kit”

YASK is a software framework for the rapid development of HPC stencil-based applications

- Stencil: an iterative kernel that updates elements in one or more N-dimensional vars using a fixed pattern of computation on neighboring elements
- Fundamental algorithm in many scientific simulations, e.g., finite-difference-method (FDM) approximations of differential equations describing various physical phenomena

![Image Processing](https://commons.wikimedia.org)

![Seismic Modeling](https://commons.wikimedia.org)

![Weather Simulation](https://commons.wikimedia.org)

Fun fact: prior to version 3, “YASK” expanded to “Yet Another Stencil Kernel"

Images from [https://commons.wikimedia.org](https://commons.wikimedia.org)
Technical and business motivation

Rapid Development

- Stencils in YASK are coded in a simple DSL (domain-specific language)
  - YASK programmer only needs to describe what to do, not how to do it
  - YASK compiler generates high-perf code from the DSL description
  - Can easily and quickly change the stencil and generate new code
  - Supports arbitrary dimensions, complex stencils, boundary conditions, and more

- Can easily and quickly try different tuning features and parameters without recoding
  - Many complex optimization techniques are available immediately
  - Supports cluster scaling, spatial and temporal tiling, vector folding, and more

- Generated code compiles into a library with documented C++ and Python* APIs to facilitate integration into real HPC applications

Performance Portability

- Can re-target stencil for different Intel® platforms by generating multiple libraries from single DSL description
- Future YASK features and supported platforms can be leveraged immediately without recoding
Example application of YASK

AWP-ODC: Anelastic Wave Propagation-Olsen, Day, Cui

- Software that simulates seismic wave propagation after a fault rupture
- Widely used by the Southern California Earthquake Center (SCEC) community

AWP-ODC-OS

- First ever open source release in 2016 (BSD-2 license), including port to Intel Xeon Phi processor, under development by San Diego Supercomputer Center (SDSC) at Univ. of CA, San Diego (UCSD)
- Demonstrated on >9000 nodes of Cori supercomputer

- CyberShake Study 15.4 hazard map for 336 sites around Southern California
- Warm colors represent areas of high hazard

Content on this slide courtesy of UCSD
Scope of this presentation

Goals
- Give a practical introduction to using YASK—a tutorial
- Provide an overview of the major YASK performance techniques
- Deliver enough information for someone already familiar with the application of stencil codes to start using YASK

Non-goals
- Not providing a tutorial on
  - Finite-difference methods, seismic modeling, brain imaging, etc.
  - C++, Linux*, OpenMP*, MPI*, github*, etc.
  - Intel® instruction sets, microprocessor architecture, etc.
- Not explaining in-depth how the YASK performance techniques work
- Not providing performance data on all the various trade-offs
- Not describing the internal software architecture
- Not a reference manual for the APIs
BASIC FEATURES AND USAGE
Download, build, and test

Code access
- Download from Intel's github* project
  - `git clone https://github.com/intel/yask`
  - MIT open-source license
- Builds and runs are made from the top-level directory: `cd yask`

Pre-requisites
- See README.md for complete list
- Only supported OS is Linux (no specific distribution recommended)
- Intel® C++ compiler needed for performance
  - Install “Intel® Parallel Studio XE Cluster Edition for Linux”
  - Binary built from gcc* is consistently lower performance
- Common utilities (gcc, perl, awk, etc.)
Example 1: Iso3dfd stencil

Description
- Isotropic 3D finite-difference code found in seismic-imaging software used by energy-exploration companies to predict the location of oil and gas deposits
  - Simple stencil with only one updated variable

Recipes for building and running
- Makefile and run script will automatically determine architecture
  - `make clean; make stencil=iso3dfd`
  - `bin/yask.sh -stencil iso3dfd -g 1024`
- The `-g 1024` option sets the global-domain size to a cube, 1024 elements on a side

Example 2: AWP stencil

Description

- Primary compute kernel for earthquake simulator described in the introduction
  - More complex problem that consists of 26 vars in a staggered-grid formulation

Recipes for building and running

- Makefile and run script will automatically determine architecture
  - make clean; make stencil=awp
  - bin/yask.sh -stencil awp -gx 1024 -gy 1024 -gz 128

- The \(-g^*\) options set the global-domain size to specific sizes in each dimension

High-level tool-chain flow

Stencil-specification code → Stencil compiler → Loop compiler → Nested loops with OpenMP pragmas, etc. → Other C++ code → Intel C++ compiler

Provided stencil perf-eval utility → and/or → Optimized stencil-kernel library

Performance results → and/or → Application results → Customer application → and/or → Provided stencil perf-eval utility

Variable declarations and optimized stencil calculation code
Stencil specification

Stencil specification code → Stencil compiler → Variable declarations and optimized stencil calculation code

Loop compiler → Nested loops with OpenMP pragmas, etc.

Provided stencil perf-eval utility → Optimized stencil-kernel library → Intel C++ compiler

Performance results → and/or → Application results

Other C++ code

Customer application

Intel C++ compiler
Example simple 25-point 3-D stencil

25 points from 3D var u(t)

...as specified by the RHS of this finite-difference equation

\[ u(t + 1, i, j, k) = c_0 u(t, i, j, k) \]
\[ + \sum_{r=1}^{4} c_r [u(t, i - r, j, k) + u(t, i + r, j, k) + u(t, i, j - r, k) \]
\[ + u(t, i, j + r, k) + u(t, i, j, k - r) + u(t, i, j, k + r)] \]

...are used to compute 1 point in u(t+1)

\[ u(t) \rightarrow u(t + 1) \]
Stencil will be applied over entire problem domain

$u(t) \rightarrow u(t+1)$

Repeat for $u(t+2)...$

“Halo” data regions

Entire problem domain—typically millions of points

“Halo” data regions
Example stencil DSL code

```
#include "yask Compiler_utility_api.hpp"
using namespace yask;

class MyStencil : public yc_solution_with_radius_base {
  public:
    MyStencil(int radius=4) :
      yc_solution_with_radius_base("my_stencil", radius) { }
    virtual void define() {
      auto t = new_step_index("t");
      auto x = new_domain_index("x");
      auto y = new_domain_index("y");
      auto z = new_domain_index("z");
      yc_var_proxy u("U", get_soln(), { t, x, y, z });
      auto i = new_misc_index("i");
      yc_var_proxy c("C", get_soln(), { i });

      auto nu = c(0) * u(t, x, y, z);
      for (int r = 1; r <= get_radius(); r++)
        nu += c(r) * (u(t, x-r, y, z) + u(t, x+r, y, z) +
                       u(t, x, y-r, z) + u(t, x, y+r, z) +
                       u(t, x, y, z-r) + u(t, x, y, z+r));

      u(t+1, x, y, z) EQUALS nu;
    }
};

static MyStencil MyStencil_instance;
```

Derive new class from yc_solution_base (or yc_solution_with_radius_base to get radius parameter)

Overload define() method

Declare 1 time and 3 space indices and 4D “u” var

Declare misc index and 1D “c” array for coefficients

Write expression for value at step t+1

Use EQUALS operator to define u(t+1)—this is not an assignment!

Register this stencil with the YASK compiler utility

---

Mathematical description of finite-difference approximation implemented in DSL code

\[
\begin{align*}
    u(t+1, i, j, k) &= c_0 u(t, i, j, k) \\
    + \sum_{r=1}^{4} c_r [ u(t, i-r, j, k) \\
    + u(t, i+r, j, k) \\
    + u(t, i, j-r, k) \\
    + u(t, i, j+r, k) + u(t, i, j, k-r) \\
    + u(t, i, j, k+r) ]
\end{align*}
\]
Building a stencil

- Stencil specification code
- Stencil compiler
- Loop compiler
- Nested loops with OpenMP pragmas, etc.
- Other C++ code
- Optimized stencil kernel library
- Provided stencil perf-eval utility
- Performance results
- Application results

- Variable declarations and optimized stencil calculation code
- Intel C++ compiler

Performance results and/or Application results

YASK tutorial
Building your example stencil

Build your stencil and the YASK compiler

▪ Put the code from a couple of slides back into a new file `src/stencils/MyStencil.cpp`
  – If you really don’t want to copy the example stencil during this tutorial, just substitute `iso3dfd` for `my_stencil` in the example commands from here on
  – If you want to write your own stencil, see the examples in `src/stencils` and the full DSL documentation for the YASK Compiler as discussed in the “Using the APIs” section later in this tutorial

▪ Optional: rebuild and run the YASK compiler manually
  – Build the compiler: `make -j compiler`
  – Run the compiler: `bin/yask_compiler.exe -h`
    – “my_stencil” should be listed as a valid parameter to the `-stencil` option
    – Lots of other options shown; most of these are passed automatically from the `Makefile`

Build the YASK kernel

▪ Build the kernel: `make -j stencil=my_stencil radius=4`
  – Targets the instruction-set architecture of your current platform by default
    – If you want to cross-compile, specify `arch=isa-code`, e.g., `knl`, `avx2`
  – Uses all the default parameters for static options such as data layout and prefetching
  – Dynamic options such as problem size are specified at run-time

▪ If you skipped the step of building the compiler above, it will be done here for you automatically
Running a stencil

Stencil specification code → Stencil compiler → Loop compiler

Optimized stencil kernel library

Variable declarations and optimized stencil calculation code

Nested loops with OpenMP pragmas, etc.

Other C++ code

Intel C++ compiler

Performance results

Provided stencil perf-eval utility

and/or

Application results

Customer application

YASK tutorial
Running your example stencil on one node

Run the provided test utility

- `bin/yask.sh -stencil my_stencil -g 512 -no-pre_auto_tune`
  - Problem size is specified as 512 in the 3 spatial dimensions (512^3 points) using `-g` option ("g" for global-domain)
  - Other parameters are set to their defaults, except the auto-tuner is disabled
  - The utility prints lots of stats about the stencil, memory usage, performance, etc.
  - Runs 3 trials for about 10 sec each and reports the best and median time and throughput (rate) stats
  - A log file is kept in the "logs" directory, which includes some compilation and platform information for posterity

Use the auto-tuner

- `bin/yask.sh -stencil my_stencil -g 512`
  - As above, but with the block-size auto-tuner enabled (by default)
    - The block size determines the amount of work done by each OpenMP "parallel for" iteration
    - The utility prints each block size that the auto-tuner tries and its measured performance
  - Compare performance of the two runs

If your node has more than one socket, try an optional communication method

- `bin/yask.sh -stencil my_stencil -g 512 -use_shm`
  - As above, but uses shared-memory buffers to speed up inter-socket communication
  - This is not the default setting because not all nodes are configured to allow enough shared memory for large problems
    - You will get an out-of-memory error (or sometimes MPI will just exit) when trying to allocate memory if this is the case
Tuning the size parameters

Block size
- From the log file of the last run, find the best block size found by the auto-tuner
  - For example, auto-tuner: best-block-size: t=1 * x=96 * y=64 * z=128
- Specify this manually to bypass the auto-tuner, e.g.,
  - bin/yask.sh -stencil my_stencil -g 512 -no-pre_auto_tune \
    -bx 96 -by 64 -bz 128
  - You should get similar performance
  - Play around with block size to see its effect
  - Contribution opportunity: github ticket #159 to save and reuse the settings automatically

General usage of size-parameter options
- Most domain-dimension sizes can be specified in two ways
  - Spatial dims set separately as we did with block size, e.g., -bx, -by, -bz
  - Leave off the dimension names to set all sizes as we did with domain size, e.g., -g
- Play around with various problem sizes to see the effect on memory and performance
Getting a list of options

The YASK test utility is composed of a script driver and a binary

- **Print script options:** `bin/yask.sh -h`
  - There is only a short list of options here
  - These are just the ones that control how the binary is launched from the script

- **Print binary options:** `bin/yask.sh -stencil my_stencil -help`
  - Need to specify the stencil because there are different binaries for each
    - If you don't specify a stencil, the script will show the ones that have been compiled
  - **Warning:** the long list might be overwhelming; we'll point out the most important ones in this tutorial 😊
Scaling out to multiple nodes

Domain decomposition

- Many HPC problem sizes won’t fit in the DRAM memory of one node
- Most HPC problems can be divided spatially across multiple nodes

YASK implementation

- Exchanges required inter-node data via two-sided MPI* communication
- **Important:** strong vs. weak HPC scaling
  - Global-domain size parameters \((-g)\) apply to the overall problem size
    - This implements a *strong-scaling* model, where the overall problem size remains constant regardless of the number of nodes
  - Local-domain size parameters \((-l)\) apply to the size on each MPI rank
    - This implements a *weak-scaling* model, where the overall problem size increases with the number of nodes
Conceptual view of 2D, 2×2 rank domains

Due to the read-radius of stencils, halo data must be read to calculate some points in the domain.

Similarly, at rank boundaries, each rank needs to read data from its neighbors.

This data changes every time-step and must be exchanged between ranks every time-step to keep the data consistent.
Schematic of 2D, 2×2 rank y-edge halo exchanges

Halos are exchanged at beginning of run and after each time-step.

Can get more complex when there are multiple dependent stencils.

Data calculated in rank 1 at time-step $n$...

...is needed by rank 3 at time-step $n+1$.

YASK tutorial
Schematic of 2D, 2×2 rank x-edge halo exchanges

Width of halos is based on radius of stencil in each dim.
Corner exchanges are only needed for halos that have diagonal points.

For 3D stencils, exchanges may be needed along faces, edges, and corners of the rectangular solids.

This concept can be extended to any number of dims.
Running on multiple nodes

Running the test utility with an MPI command

- bin/yask.sh  -mpi_cmd 'mpirun -f $PBS_NODEFILE -ppn 2' -stencil my_stencil -g 1024 -no-pre_auto_tune
  
  - This example uses $PBS_NODEFILE host file created by IBM Platform LSF*
    - Use the technique for your scheduler, explicit -hosts option, or whatever is used on your cluster
    - Important: specify how many processes to run on each node, e.g., with -ppn, usually corresponding to the number of NUMA nodes
  
  - Use the -nr* options to control the topology of the MPI ranks
    - Example: -nr 2 specifies 2 ranks in each spatial dimension
    - Example: -nrx 2 -nry 4 -nrz 8 specifies 2 ranks in the 'x' dimension, 4 in the 'y', and 8 in the 'z', for 64 total ranks
  
  - Auto-tuning is turned off in this example, but it can also be used across multiple nodes
  
  - You can also specify -use_shm: it will automatically be applied only between ranks that share a virtual address space

Implementation optimization

- For typical configurations, YASK overlaps communication with [most] computation using this sequence:
  1. Only the data needed by another rank is calculated on each rank for a given time-step
  2. Asynchronous (non-blocking) MPI receive and send requests are initiated
  3. The remaining data on each rank is calculated while the MPI data is in-flight
  4. The MPI requests are completed before the next time-step is begun

- You can disable this feature via -no-overlap_comms to measure the impact
USING THE APIS
Making a stencil application

Stencil specification code → Stencil compiler → Loop compiler → Nested loops with OpenMP pragmas, etc. → Other C++ code

Performance results

Provided stencil perf-eval utility

Optimized stencil-kernel library

Customer application

Application results

Variable declarations and optimized stencil calculation code

Intel C++ compiler
API Overview

Purpose
- Define the DSL (Domain-Specific Language) used by the YASK compiler
- Facilitate inclusion of stencil code generated by YASK into real applications

Target languages
- C++ natively
- Python* interface generated via SWIG*

Design principles
- Consistent interface
- Stable interface with backward-compatibility maintained except in rare cases
- Documented
- Hidden implementation via C++ pure-virtual classes in most cases
- “Factory” pattern: use small number of factory objects to create more objects
API documentation

From https://github.com/intel/yask, follow the “API documentation” link in the Overview section of README.md

There are four main tabs

- **Main Page** contains an overview of the YASK workflow and API sets
  - Read this first
- **Modules** lists the API sets
  - This is the best place to start looking for the API you want
  - **YASK Compiler** APIs are used to define stencil solutions, i.e., the “DSL”
    - Objects in the compiler API start with “yc_”
  - **YASK Kernel** APIs are used to create your own application from a YASK library
    - Objects in the kernel API start with “yk_”
  - **YASK Common Utilities** contains some common utility classes like output streams
    - Objects here start with “yask_”
- **Classes** lists all the C++ classes in alphabetical order
- **Files** lists the C++ header files
Access to the APIs

C++

- The standard method of writing YASK stencils uses the compiler API library
- When you build a kernel as in the previous examples, you have implicitly made a kernel API library
  - A shared object is created in the lib directory, one for each stencil and architecture
  - The performance-test utility we've been using is just a wrapper around the library
- The header files are in the include directory
  - #include "yask_kernel_api.hpp" in your application
  - See src/kernel/tests/yask_kernel_api_test.cpp for an example

Build the APIs for Python* only if needed

- Download src code as before: git clone https://github.com/intel/yask
  - See README.md for SWIG* version requirement
- From yask directory, run make -j api
- Python APIs are not documented separately
- Performance-wise, the stencil calculations will be the same speed as in C++, but data access is slower
Main YASK kernel process steps

1. Create a YASK “solution” object
2. Set problem size and other parameters
3. “Prepare” the solution: allocate memory and sync meta data
4. Run solution for $n$ time-steps
5. Initialize data in YASK variables
6. Inject data representing external stimulus if needed, etc.
7. Extract results from YASK variables
8. Done?
Creating a YASK kernel via the APIs

Bootstrap

- Declare an object of type `yk_factory`
- Call `yk_factory::new_env()` to create an “environment” object
  - This can be used to provide an existing MPI communicator if desired
- Call `yk_factory::new_solution()` to create a YASK “solution” object
  - The solution object provides methods to configure the problem size, access YASK variables, and run the stencil calculations

What’s with all the output?

- Calling many APIs results in debug output written to a stream
  - By default, this stream is standard output
  - Handy for development, but you probably don’t want this in your final application
- Call `yk_solution::set_debug_output()` to change it to a file or discard it
  - Use a `yask_output_factory` to create new output streams
  - NB: this mechanism was used instead of raw C++ streams for Python usage
Key terms related to domain sizes

Overall (global) and rank (local) domain sizes

- As with the test utility, each MPI rank can define the local-domain size or the global-domain size in each dimension
  - The size that is not specified will be calculated automatically
  - All ranks in a given row must have the same rank-domain height, etc.
- Also, each rank can specify the number of ranks in each dimension or let the run-time assign them automatically

YASK-variable dimensions and sizes

- When YASK variables are defined, they can have three types of dims
  - **Step dim**: usually “t” for “time”
  - **Domain dim**: usually used for spatial dims like “x” and “y”
    - Generally, can be any domain-size parameter that may be decomposed over ranks
  - **Misc dim**: an index that is known at YASK-compile time like a coefficient index
- The size of the domain will be used when allocating each variable
  - Read the “Detailed Description” section of the `yk_var` API page for an explanation of each size
Global view of 2D problem size

The “domain” is the region where the stencils are calculated and the YASK vars are updated.

Halo regions are not considered part of the domain.

The “domain” is the region where the stencils are calculated and the YASK vars are updated.

Halo regions are not considered part of the domain.

Global-domain size in y dim

Local-domain size in x dim in ranks 1 and 3

YASK tutorial
Per-rank view of 2D problem sizes

Local-domain size in y dim in ranks 0 and 1

Halo size—can be different for each variable, dimension, and side of domain

YASK may add extra padding to ensure vector data-alignment, etc. to each variable.

Application programmers usually don't need to set the size of halos or padding—YASK computes them for you.
Prepare the kernel solution

Set up the problem sizes and ranks

- Call `yk_solution::set_overall_domain_size()` to set the global-domain sizes
  or `yk_solution::set_rank_domain_size()` to set the local-domain sizes
- Call `yk_solution::set_num_ranks()` to explicitly set the number of ranks if desired
- Call `yk_solution::set_rank_index()` to explicitly set the position of each rank if desired

Optional: Set other solution parameters

- Call `yk_solution::set_block_size()` to use the best block size you found during the tuning process
- Alternatively, or in addition, you can use the block-size auto-tuner
  - The auto-tuner can be controlled with APIs
  - There are several ways to use the auto-tuner, discussed in the advanced section
Allocate data and synchronize info across ranks

Call `yk_solution::prepare_solution()`

- Shares MPI positions across ranks or calculates a default position for each rank
  - Since `prepare_solution()` uses MPI calls, it is critical to call it from each rank
- Ensures domain-size consistency across ranks
- Allocates data for each YASK variable based on halo sizes determined by the YASK compiler and the local domain sizes
- Determines MPI buffer requirements and sizes and allocates data for them
Initialize data

Access YASK variables
- Get a list via `yk_solution::get_vars()` or find a specific var by name via `yk_solution::get_var()`
- Either one returns pointers to variables of type `yk_var`
  - Used to find all meta data about each var
  - Use `get_*_index()` APIs to determine valid indices
  - Use `idx_t` as a type for indices: `size_t` and `int` are not good alternatives.

Data access
- YASK uses a non-standard tiled data layout ("vector folding" discussed later)
  - Thus, it does not support simple overlay with native row-major or column-major arrays of floating-point numbers
  - However, github issue #147 requests to support this under certain restrictions for easier integration with legacy applications
- There are several APIs for writing one or more FP elements from and to the variables
  - Start with `set_element()`, `set_elements_in_slice()`, and `set_all_elements_same()` in `yk_var`
  - The "slice" versions are threaded for performance and convert to and from row-major layout
    - But you'll probably want to avoid copying whole vars this way to avoid wasting memory
YASK-variable indices

Important: var indices are always global!

Note: some may be negative.

Tip: use an API to determine each index instead of adding sizes or doing similar math.
Notes on initializing data

Of course, load data into the domain areas.

Load data into the outer halos or initialize them to a safe value.

No need to load data into the halos between ranks because they will be copied from the neighbors during the first halo exchange.

No need to load data into the extra padding area because it is never read.
Make the stencil calculations and get results

Advance the simulation through a series of time-steps

- Call `yk_solution::run_solution()`
  - General form takes first and last time indices, e.g.,
    - `run_solution(0, 9)` runs the first 10 time-steps
    - `run_solution(10, 19)` runs the next 10

- If needed, between time-steps, you can access var data, e.g., for source injection

Access results

- Use appropriate var methods analogous to the data-writing ones listed earlier
  - Start with `get_element()` and `get_elements_in_slice`
- Call `yk_solution::get_stats()` to collect some performance stats if desired
- Call `yk_solution::end_solution()` when all done to release memory, etc.
Advanced APIs

YASK Compiler APIs

- Objects in the compiler API start with “yc_”
- Very similar functionality to the DSL and the YASK compiler utility
  - Methods to make dims, vars, expressions, spatial and temporal conditions, etc.
  - Also provides operator overloading in C++ and Python to enable readable expressions (except for logical `and`, `or`, and `not` in Python)
- Only needed if you want to write a program to create a YASK stencil
  - However, a planned feature of YASK version 3 will replace the [undocumented] DSL with these APIs (with a conversion utility)

Misc interesting APIs (not [remotely] exhaustive)

- Call `yk_var::set_numa_preferred()` to set the NUMA node for a given variable—useful for explicit placement in MCDRAM or DDR on Xeon Phi CPUs
- Call `yk_solution::new_var()` to make variables beyond those used in the solution
- Call `yk_solution::apply_command_line_options()` to parse a command-line string—useful for quickly applying options from the test utility in another application
- Call `yk_var::add_to_element()` to atomically update a var element—useful for threaded source injection
Exceptions

YASK APIs that can trigger errors throw exceptions

- Allows applications to catch and process errors
- Find exception documentation in the YASK Common Utilities module tab

C++

- Throws object of type `yask_exception`
- Call `yask_exception::get_message()` for a human-readable explanation
- See `src/kernel/tests/yask_kernel_api_exception_test.cpp`

Python

- Throws `RuntimeError` as defined by SWIG*
- Call `format()` for a human-readable explanation
- See `src/kernel/tests/yask_kernel_api_exception_test.py`
ADVANCED STENCILS AND TUNING
Solutions with multiple stencils

Purpose

- Many modern seismic simulations require multiple variables to be updated, each with a different stencil

Independent stencils

- **Definition:** for any two stencils, the input of one stencil does not depend on the output of the other within the same time-step, and vice-versa

- **Example:** $x$-stress$(t+1)$ depends on $x$-stress$(t)$, and $y$-stress$(t+1)$ depends on $y$-stress$(t)$, but $x$-stress$(t+1)$ does not depend on $y$-stress$(t+1)$ or vice-versa

  - See `define_str_TL()` in `src/stencils/SSGElasticStencil.cpp`
  - Thanks to contributor Albert Farres from the Barcelona Supercomputing Center!

- For independent stencils, YASK applies the stencils simultaneously in one pass over the domain

Dependent stencils

- **Definition:** for any two stencils, the output of one stencil is required for the input of the other stencil in the same time-step (the opposite cannot also be true)

- **Example:** $x$-velocity$(t+1)$ depends on $x$-stress$(t)$, and $x$-stress$(t+1)$ depends on $x$-velocity$(t+1)$ – see same DSL file

- For dependent stencils, YASK cannot apply the stencils simultaneously and makes subsequent passes over the domain

- YASK automatically determines the dependencies between equations in the DSL and schedules their stencils
Absorbing boundaries

Used in wave-propagation simulation
- Simulation domain covers only a finite block of earth, water, etc.
- Simulated waves will erroneously reflect from the arbitrary boundaries
- Artificial absorbing boundary conditions (ABC) or boundary layers are used to reduce these reflections
- There are many techniques and papers covering this field of study

Two high-level approaches to use in YASK
- Modify the stencil(s) to include simple attenuation factors
  - *Example:* Cerjan sponge layers
  - *See Iso3dfdSpongeStencil class in* src/stencils/Iso3dfdStencil.cpp
    - Try modifying your test stencil to use sponge layers and check the impact on performance
  - *Tip:* this stencil and its parent class also show the recommended practice of creating functions to return expressions, allowing more flexible composition
- Use *different* stencils in the boundary layers by creating *sub-domains*
  - *Examples:* Higdon BCs, perfectly-matched-layers (PML), and the convolutional form (CPML)
Sub-domain (spatial) conditions

A sub-domain is a subset of the domain in which a stencil is applied

- Defining stencils in boundaries is a common use-case for sub-domains

A sub-domain is defined in the DSL by a Boolean expression on the domain indices

- The DSL includes terms for the left-most and right-most indices in the domain

Example: \((x < \text{first}\_\text{index}(x) + 5)\) defines a sub-domain on the left side of the domain, 5 elements wide

- \((x > \text{last}\_\text{index}(x) - 5)\) would be the same on the right side
- \((x \geq \text{first}\_\text{index}(x) + 5) && (x \leq \text{last}\_\text{index}(x) - 5)\) would be used for the interior sub-domain between the left and right boundary sub-domains

- These conditions are written after the equation using the \texttt{IF\_DOMAIN} operator in the DSL

  - Search for \texttt{TestBoundaryStencil} in src/stencils/TestStencils.cpp for simple synthetic examples
  - See src/stencils/FSGElasticStencil.cpp for a more complex real-world example
Sub-domains on multiple ranks

Halo regions are *not* considered part of any domain or sub-domain.

Sub-domain described by
\( x < \text{first_index}(x) + 5 \)

Sub-domain described by
\( x > \text{last_index}(x) - 5 \)

Sub-domain described by
\( x \geq \text{first_index}(x) + 5 \) \& \( x \leq \text{last_index}(x) - 5 \)

Sub-domains always apply across the overall problem domain.

(This is one reason that domain indices are always global.)
Step (temporal) conditions

A step-condition expresses which time-steps are valid for a stencil
- Some stencil applications require special processing on regular intervals
- Wave-field sub-sampling can be implemented with step conditions

Like sub-domain conditions, step conditions restrict stencil application
- Sub-domain expressions can use only the domain indices
- Step-condition expressions can use the step index (usually time) or values in other YASK variables (but not via domain indices)

**Example:** \((t \mod 8 == 0)\) is true every 8\(^{th}\) time-step

These conditions are written after the equation using the `IF_STEP` operator in the DSL
- Look for TestStepCondStencil in `src/stencils/SimpleTestStencils.cpp` for a simple example
Vector folding (multi-dimensional vectorization)

Concept
- Store small 2D or 3D block of data into each SIMD vector
- **Pros**: reduces memory loads and memory streams compared to traditional 1D in-line vectorization
- **Cons**: requires non-traditional tiled data layout and additional shift and/or permute operations preceding SIMD arithmetic operations

Results
- Significant speedup shown on large-radius stencils, especially on Intel® Xeon Phi™ processors
- Works well paired with tiling and other performance techniques

Implementation
- The YASK compiler automatically generates the proper shift and permute instructions
- The YASK kernel code automatically generates code to store the tiled data and look up elements by index when needed
- Intel® AVX-512 instruction set is needed for efficient permutes, so only enabled when available
- See the paper in the upcoming reading list showing ~1.5× speedup on some stencils
Traditional 1D Vectorization

8 new vectors must be read for \(k \pm r\) points
(4 for \(k+r\) and 4 for \(k-r\) for \(r=1..4\))

Only 1 new vector must be read for \(i \pm r\) points due to overlap along x axis

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

Total BW cost for traditional “in-line” vectors
= 17 new vector inputs for each vector of output
(some loads will come from cache with blocking)
2D Vector-Folding

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

4 new 4x2x1 vectors must be read for \( j \pm r \) points

Only 1 new vector must be read for \( k \pm r \) points

2 new vectors must be read for \( i \pm r \) points

Total BW cost for 4x2x1 vector with z-axis loop = 7 new vector inputs for each vector of output (2.4x lower than in-line)
Vector-Folding Memory Layout and Code Gen

2D “4x2” vector folding

Logical indices in 2D with 8-element SIMD in x and y dimensions

| X | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |...
|---|----|----|----|----|----|----|----|----|----|
| Y | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |...

- 2D vector folding layout (8×1)
- Two aligned vectors are colored
- Unaligned read shown with bold borders done by loading aligned vectors and then shuffling the requisite elements via an AVX-512 permute instruction

Layout in memory (1D)

Load and permute instructions generated automatically by YASK stencil compiler

Access to elements in custom memory layout encapsulated behind C++ & Python APIs

YASK tutorial
Vector-folding customization

Specify the vectorization length in each dimension

- Use the `fold='x=n,y=n,z=n'` argument to the make command-line
  - The values are passed to the YASK stencil compiler and used to generate code
  - The fold settings are also included in non-generated code during compilation
  - **Example:** `make fold='x=1,y=2,z=8'` generates code using a $1\times2\times8$ fold
  - Try different fold settings on the test stencil and check the impact on performance
    - **Important:** Be sure to run `make clean` before re-compiling when changing compile-time options like vectorization

- The product of the fold lengths should equal the number of SIMD vector elements in the target architecture and FP precision (single or double)
  - **Example:** single-precision FP using 512-bit SIMD contains 16 elements per vector
  - If the fold is not the right size, the compiler will adjust the requested fold using a heuristic algorithm
  - The vector length in any dimension not specified defaults to one (1)
  - The default fold varies depending on dimensionality, architecture, and FP precision
    - Is $4\times4\times1$ for a 3D problem using SP FP on Xeon Scalable CPUs, for example
More compile-time settings

Floating-point precision
- The default FP size is 4 bytes, which is the norm in seismic modeling
- Build with double-precision by adding `real_bytes=8` to the `make` command

Prefetching
- Software prefetch instructions can be added by the YASK compiler
  - Use `make ... pfd_l1=n` to set the L1 prefetch distance to `n` iterations ahead
  - Use `make ... pfd_l2=n` to set the L2 prefetch distance to `n` iterations ahead
  - Set `n` to zero (0) to disable generation of any prefetch instructions for a level
- The default prefetch depends on the architecture

Time allocation
- By default, the YASK compiler tries to determine the minimum number of time-steps that need to be saved in each variable, but you may want to override it
- Override this with `make ... time_alloc=n` (to affect all vars at compile-time)
- Override at run-time with the `yk_var::set_alloc_size()` API
More compile-time settings

Order of domain dimensions

- The order of domain dimensions determines
  - The inner-most or “unit-stride” dimension in the memory layout
  - The nesting order of loops for the stencil-calculation code
  - Default vector-folding and default rank layout

- By default, the order of domain dimensions is determined from the variable declarations
  - For example, \texttt{new\_var("A", \{t, x, y, z\})} implies domain dimensions are in 'x, y, z' order
  - If vars have different domain-dimension index orders, dimensions are sorted in the order they are seen by the compiler

- You can explicitly set the order of domain dimensions
  - For example, \texttt{make ... domain\_dims='z, y, x'} creates domain dimensions in 'z, y, x' order
  - This setting overrides the order of the dimensions specified in the variable declarations and affects memory layout, looping order, vector-folding, and rank layout
  - Try different orders on the test stencil and check the performance impact (remember to \texttt{make clean})

Misc settings

- Any preprocessor macro can be set by \texttt{make ... EXTRA\_MACROS='name=value ...'}
- Type \texttt{make help} for examples of settings the C++ optimization level and more
More grouping terms: clusters, bundles, and packs

Clusters
- The YASK compiler can also unroll the innermost loops a specified number of times, effectively calculating a *cluster* of SIMD vectors in each iteration.
- Control via the `cluster='x=n,y=n,z=n'` argument to the make command-line similar to the `fold` argument for vector-folding:
  - Try different cluster settings on the test stencil and check the performance impact (remember to `make clean`).
- The default clustering is to not unroll (cluster size of one in each dimension).
- Clustering increases register pressure, making it useful only for small stencils.

Bundles
- A *bundle* is the term used for independent equations that are grouped into one function by the YASK compiler.
- A bundle cannot contain equations from different sub-domains.
- The compiler eliminates common sub-expressions between stencils in a bundle.

Packs
- A *pack* is the term used for a group of equation bundles that are no inter-dependent.
- This can occur when there are equations in different sub-domains that are not dependent on one another.
- Two or more packs may be evaluated within one cache block that crosses sub-domain boundaries.
- Multiple packs are then scheduled in the correct order based on dependencies.
Temporal wave-front rank tiling

**Goal**
- Increase temporal locality in caches at a package level
- Especially useful for large unified cache (such as MCDRAM cache on Xeon Phi)

**Technique**
- Evaluate $n$ time-steps in a subset of local rank domain
  - We refer to this subset as a region in YASK
- Evaluate each region *sequentially* until entire rank domain is evaluated for $n$ time-steps

**Increased software complexity**
- Must respect mathematical dependencies between time-steps
- Halo exchanges between ranks become more complex
- Any inter-time-step data processing (like source injection) becomes more complex
Temporal wave-front dependencies

- In the first temporal wave-front tile shown here, the number of values that can be calculated is reduced for each time-step (and each pack).
- The amount of shift is called the wave-front or temporal angle and is based on the radius of the largest stencil (shift of 8 shown).

Key:
- ○: value already known
- □: value not yet known
- ◼: known value used as input
- ◼: value being calculated

Highest x-value that may be calculated for t=4 due to the dependencies on t=3 values

Highest x-value that may be calculated for t=2 due to the dependencies on t=1 values

Tile ends here for t=1
Covering a temporal range via multiple tiles

- Wave-front tiles are computed *sequentially*, but multiple values within one time-step of a given tile may still be evaluated *concurrently*.
- After last tile is complete, YASK variables contain the same data as they would have without temporal tiling.
- This concept is directly extended to 2D or 3D stencils by shifting in each spatial dimension.
- To handle MPI halo exchanges, some redundant calculations are made between ranks.
Using temporal wave-front rank tiling in YASK

Another level of tiling hierarchy
- Earlier, we explained how each rank domain is divided into blocks, set via \( -b \)
- Ranks may also be divided into regions, where each region is a wave-front tile
- Control the spatial size of a region with \( -r, -rx, -ry \), etc.
- Control the temporal size of a region with \( -rt \) (assuming your time dim is “t”)
- By default, the spatial size of a region is the size of the rank, and the temporal size is one (1)

Selecting values
- When using a very large unified cache (e.g., a 16GiB MCDRAM cache on Xeon Phi), set the spatial size to fit within this cache
  - See the paper in the upcoming reading list showing \(~2 \times\) speedup on two stencils
- For processors that have much smaller third-level caches (e.g., a Xeon Scalable processor), values are much more critical, and additional performance is more difficult to obtain
Temporal block tiling

Goal
- Increase temporal locality in caches at a core level, e.g., level-2 caches

Technique
- Evaluate $n$ time-steps in each block
  - Recall that a rank is composed of regions, and a region is composed of blocks
- Evaluate blocks *concurrently* until entire region is evaluated for $n$ time-steps

Increased software complexity (similar to wave-front tiling)
- Must respect mathematical dependencies between time-steps
  - Due to concurrency, shapes of blocks are more complex than with sequential wave-front tiling
- Any inter-time-step data processing (like source injection) becomes more complex
Temporal block dependencies

- In one spatial dimension (as shown), this is called triangle or trapezoid tiling (half of diamond tiling)
- Extension into multiple spatial dimensions requires a more complex series of multi-dimensional shapes (polytopes) to tessellate the space (not shown)
- The amount of shift or temporal angle is based on the radius of the largest stencil (shift of 2 shown)

These three “upward” triangle blocks may be evaluated concurrently. When they are complete, the “downward” triangle blocks may be evaluated.

Key
- o value already known
- / value not yet known
- □ known value used as input
- ◼ value being calculated

YASK tutorial
Using temporal block tiling in YASK

Not another level of tiling hierarchy (yet)

- As before, control the spatial size of a block with \(-b, -bx, -by, \) etc.
- Control the temporal size of a block with \(-bt\) (assuming your time dim is “t”) 
- By default, the temporal size of a block is one (1)
- Temporal block tiling occurs \textit{within} wave-front rank tiling, so the \textit{temporal} size of a region defaults to the temporal size of a block
  - But the default \textit{spatial} size of a region is still the spatial size of a rank domain

Selecting values

- When selecting spatial sizes, consider level-2 cache size, number of YASK variables accessed, the FP-element size, and the number of cores
Temporal wave-front mini-block tiling

Goal
- Observation: block sizes are used as a unit-of-work for OpenMP* threads as well as level-2 cache targeting
  - These objectives can conflict when small cache sizes relative to the number of variables calls for small blocks, which can lead to small OpenMP tasks
- Also, it may be beneficial in some cases to combine the concepts of triangle tiling and wave-front tiling across different dimensions
- Goal is to separate the thread and cache block concepts and provide more tiling flexibility

Technique
- Evaluate $n$ time-steps in a subset of each block
  - We refer to this subset as a mini-block in YASK
- Evaluate each mini-block sequentially until entire block is evaluated

Increased software complexity (similar to previous tiling)
- Must respect mathematical dependencies between time-steps
  - Similar to sequential wave-front tiling, but within [hyper] triangles of temporal blocks
Temporal mini-block dependencies

- Blocks are evaluated concurrently as before using OpenMP threads
- Mini-blocks are evaluated sequentially within each block
- Mini-blocks use wave-front tiling, similar to wave-front rank tiling, but inside blocks instead of ranks
Using mini-block tiling in YASK

Yet another level of tiling hierarchy

- Blocks may be divided into *mini-blocks*, where each mini-block is a wave-front tile
- Control the spatial size of a mini-block with \(-\text{mb}, \ -\text{mbx}, \ -\text{mb}y, \ etc.\)
- The temporal size of a mini-block is always the same as a block, so it is set implicitly via \(-\text{bt}\)
- By default, the spatial size of a mini-block is the size of a block

Selecting values

- The size of a mini-block should normally correspond to the size of a level-2 cache, considering the number of variables accessed and the FP-element size
- The block sizes can now be larger, considering the number of cores with thread balancing
- By sizing mini-blocks as wide as blocks in one or two dimensions and/or sizing blocks as wide as regions in some dimensions, interesting special-case scenarios may be created
Nested OpenMP threads

Goal

- **Observation:** using hyper-threads (SMT) across blocks effectively reduces the usable size of the level-2 cache available to each thread
- This is particularly impactful on Xeon Phi processors with not only 4 hyper-threads per core but also 2 cores sharing a level-2 cache
- **Goal:** allow multiple threads to use shared caches constructively rather than destructively

Technique

- Evaluate subsets of each mini-block by threads that share caches
  - We refer to this subset as a *sub-block* in YASK
- Evaluate sub-blocks *concurrently* until a *single time-step* in a mini-block is evaluated (thus, no temporal sub-block tiling)
- Since blocking already uses OpenMP threads, sub-blocking is implemented with a nested level of OpenMP threading
Using sub-block tiling in YASK

The lowest level of tiling hierarchy

- Mini-blocks may be divided into sub-blocks
- Control the spatial size of a sub-block with -sb, -sbx, -sby, etc. (the temporal size is always one)

Selecting values

- By default, the spatial size of a sub-block depends on the number of threads are used per block
  - If there is one thread per block (no nested OpenMP), the default size of a sub-block is the size of a mini-block
    - In this case, it is recommended to use only one thread per level-2 cache
  - If there are >1 threads per block (nested OpenMP active), the default size of a sub-block is a narrow slab the width of one vector (in the first dim) and the size of a mini-block in the other dims
    - This setting is intended to increase reuse between hyper-threads while keeping threading overhead as low as possible
- Control the number of threads per blocks with -block_threads
  - If you do not want to use hyper-threads, also use -thread_divisor, which divides the default number of threads
    - On a Xeon CPU, use -thread_divisor 2 -block_threads 1 to run one thread from each core per block
  - For Xeon Phi, choosing 8 threads per block often works well and is the default
  - For Xeon processors, using both threads is better for some stencils, and using only one is better for others
- Try adjusting threads-per-block and/or sub-block sizes on the test stencil and check performance
### Review of the hierarchy

<table>
<thead>
<tr>
<th>Hierarchy level</th>
<th>Target structure</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall (global) problem domain</td>
<td>Cluster</td>
<td></td>
</tr>
<tr>
<td>Rank 0</td>
<td>Rank 1</td>
<td>...</td>
</tr>
<tr>
<td>Region 0</td>
<td>Region 1</td>
<td>...</td>
</tr>
<tr>
<td>Block 0</td>
<td>Block 1</td>
<td>...</td>
</tr>
<tr>
<td>Mini-block 0</td>
<td>Mini-block 1</td>
<td>...</td>
</tr>
<tr>
<td>Sub-block 0</td>
<td>Sub-block 1</td>
<td>...</td>
</tr>
<tr>
<td>NUMA node</td>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>Pkg cache</td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>OMP thread</td>
<td>OMP team</td>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>OMP thread</td>
<td>Nested OMP</td>
<td></td>
</tr>
</tbody>
</table>
Combining tile settings

Try the following on a two-socket Intel® Xeon® Scalable CPU

- `bin/yask.sh -stencil my_stencil -l 1024 \
  -mbx 32 -mby 32 -mbz 128 \
  -bx 220 -by 256 -bz 256 -bt 12 -trial_steps 36 \
  -block_threads 1 -thread_divisor 2 -no-pre_auto_tune`

- Mini-block settings selected for a 1MiB L2 cache
  - 32 × 32 × 128 × 4B × 2 = 1MiB (128 in unit stride; “× 2” for two time-steps in memory)

- Spatial block settings selected for a multiple of number of cores per socket
  - Ceiling(1024 ÷ 220) = 5 in x dimension, 1024 ÷ 256 = 4 in y and z
  - 5 × 4 × 4 = 80 = 4 × 20 cores (modify block size if your CPU has a different core count)
  - A small multiple (like 4) often works better than the exact count because several blocks per core gives more opportunity for dynamic load balancing

- Temporal block setting (-bt 12) chosen experimentally

- Region size not used because there is no very large cache (like MCDRAM on Xeon Phi CPUs)

- Block-threads and thread-divisor chosen to use only one hyper-thread, so sub-blocks are automatically scaled up to mini-block size
More about the block-size automatic tuner

Pre-calculation mode
- Runs before the desired stencil calculations are done
- Does not ensure that calculations are done in the proper order
  - Thus, important to [re]initialize data after running in this mode for use in deployed applications
- Intended for benchmarking to tune block-size before running actual time-steps
- Use [-no]-pre_auto_tune option to control in provided test utility (default is on)
- Call API run_auto_tuner_now() to activate outside in your application

Intra-calculation mode
- Runs during desired stencil calculations
- Maintains proper calculations
- Intended for deployment, esp. on multiple platforms or when all final platforms are not known a priori
- Use [-no]-auto_tune option to control in provided test utility (default is off)
- Call reset_auto_tuner() to turn off or on explicitly
- Call API is_auto_tuner_enabled() to determine whether it is [still] running
Multi-parameter auto-tuner

Goal
- Use a genetic algorithm to explore the design space of many parameters
- Can target compile-time and/or run-time parameters

Usage
- Run `utils/bin/yask_tuner.pl` to see options
- By default, explores almost every compile-time and run-time parameter
  - Practically, you probably want to limit some of them
  - To fix the problem size, use `-g*` or `-l*` options, e.g., `-g=1024 -gz=128`
  - To search only run-time parameters, use `-noBuild` option
- Tuner will stop when there has been no improvement over 5 generations
- Tip: alternate between manual and automated tuning for final optimization
- Currently only works for 3D (spatial) problems with dims named `t`, `x`, `y`, and `z`
YASK tutorial

YASK workflow driven by auto-tuner

- **Stencil-specification code**
- **Stencil compiler**
- **Optimized stencil calculation and prefetch code**
- **Loop compiler**
- **Nested loops with OpenMP, prefetch code, etc.**
- **Executable stencil kernel binary**
- **Intel C++ compiler**
- **Other C++ code**
- **Automated Tuner**
- **Performance results**
WRAP-UP
Current work item: Devito* integration

- Devito: a symbolic finite-difference software framework
- Generates stencils automatically from differential equations
- Under development at Imperial College London

\[ \text{Devito (Python)} \]

\[ \text{PDE(s) and data} \] \rightarrow \text{Devito (Python)} \rightarrow \text{Stencil-compiler Python module} \rightarrow \text{Loop compiler} \rightarrow \text{Optimized stencil calculation and prefetch code} \rightarrow \text{Nested loops with OpenMP, prefetch code, etc.} \rightarrow \text{Other C++ code} \rightarrow \text{Intel C++ compiler} \rightarrow \text{Results} \]

- \text{PDE(s) translated to stencil AST} 
- \text{Data to stored in vector-folding format} 
- \text{Stencil-kernel Python module} 
- \text{Generated stencils automatically from differential equations} 
- Under development at Imperial College London
Read more about YASK features and applications

- “Multi-level spatial and temporal tiling for efficient HPC stencil computation on many-core processors with large shared caches.” C Yount, A Duran, J Tobin. *Future Generation Computer Systems*, 2017
- “Performance Optimization of Fully Anisotropic Elastic Wave Propagation on 2nd Generation Intel® Xeon Phi (TM) Processors.” A Farres, C Rosas, M Hanzich, A Duran, C Yount. *2018 IEEE International Parallel and Distributed Processing Symposium*
Call to action

Work through this tutorial!

Code your own stencil

- Use it in a real application
- Please feel free to contact the developers with questions
- Contribute your stencil for others to use: create a fork on github* and submit a pull request
- Please tell the developers about your experience, maybe even co-publish results

Contribute to the project

- See the “issues” database on github* for the to-do list
- Talk to the developers about something you’d like to work on
  - Probably some good academic projects in there!