Developing HPC Stencil Code using the YASK Framework

This tutorial applies to YASK version 4.05.00 and later*

Chuck Yount
Intel Americas, Inc.

April 8, 2024

*see README.md for any exceptions

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.

Outline

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
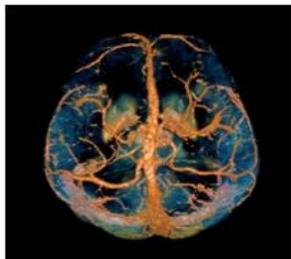
Image Processing
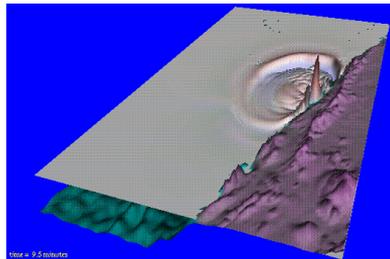
Seismic Modeling
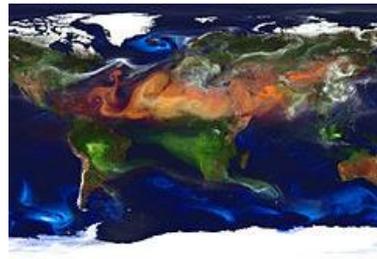
Weather Simulation

Images from https://commons.wikimedia.org

About the name:
- "YASK" is named in the tradition of Linux utilities such as YACC, YAML and YAKL: https://en.wikipedia.org/wiki/Yet_another
- Prior to version 3, "YASK" stood for "Yet Another Stencil Kernel"

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 stencils for different Intel® CPU and GPU 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

2sec SA, 2% in 50 yrs
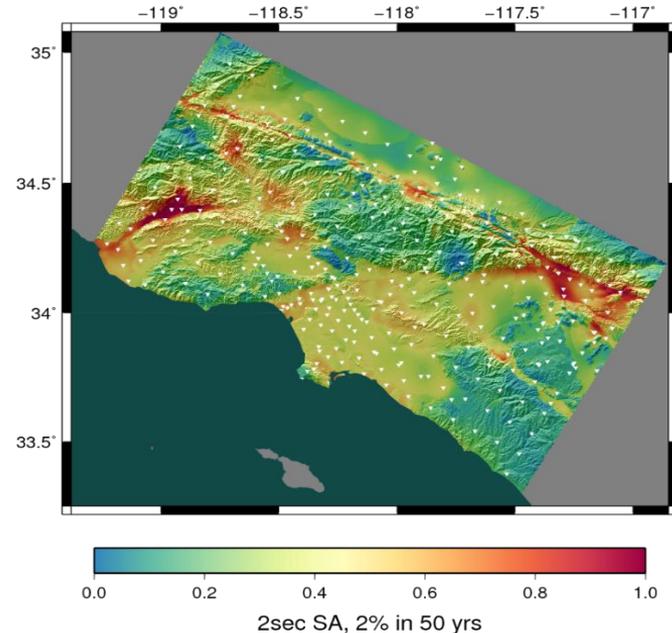
- CyberShake Study 15.4 hazard map for 336 sites around Southern California
- Warm colors represent areas of high hazard

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, CPU/GPU 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® oneAPI HPC Toolkit"
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html
- Common Linux utilities (gmake, perl, awk, python, 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 on CPU

- Makefile and run script will automatically determine architecture
  - `make clean; make -j 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

Image from https://commons.wikimedia.org/wiki/File:PlatformHolly.jpg. Public domain--U.S. DoE.

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
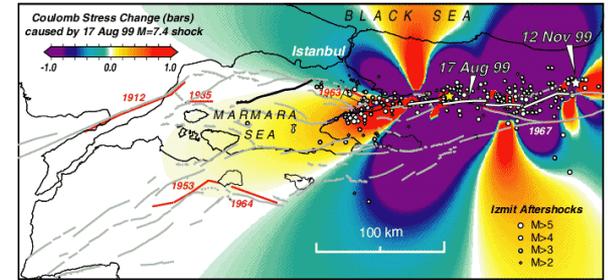
Recipes for building and running on CPU

- Makefile and run script will automatically determine architecture
  - `make clean; make -j 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
  - Compare this to the simpler `-g` option in the iso3dfd example that set the global-domain size in all three spatial dimensions
  - This is a common technique used for all spatial-dimension command-line options

Image from https://commons.wikimedia.org/wiki/File:Izmit_11-12-99.gif. Public domain--U.S.G.S.

High-level tool-chain flow
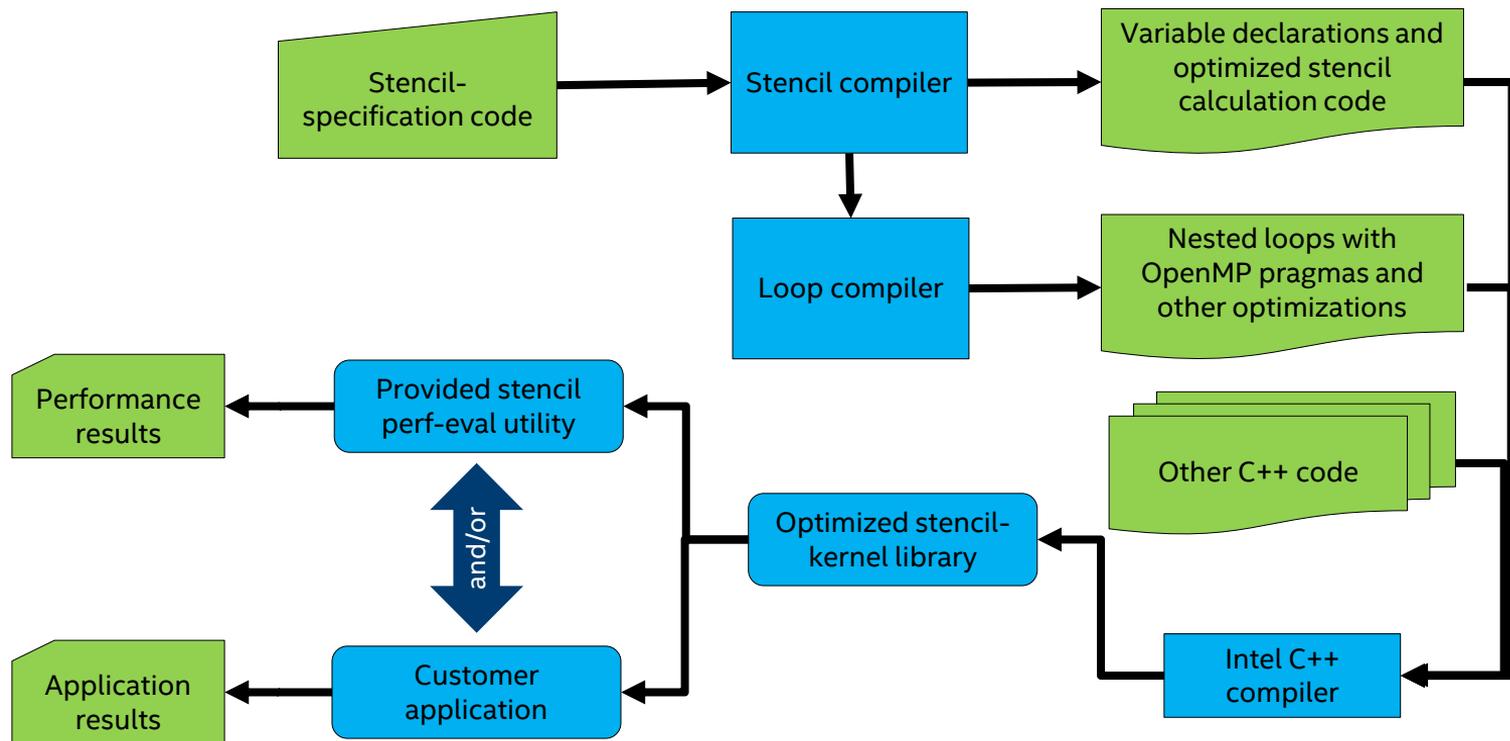
Stencil specification
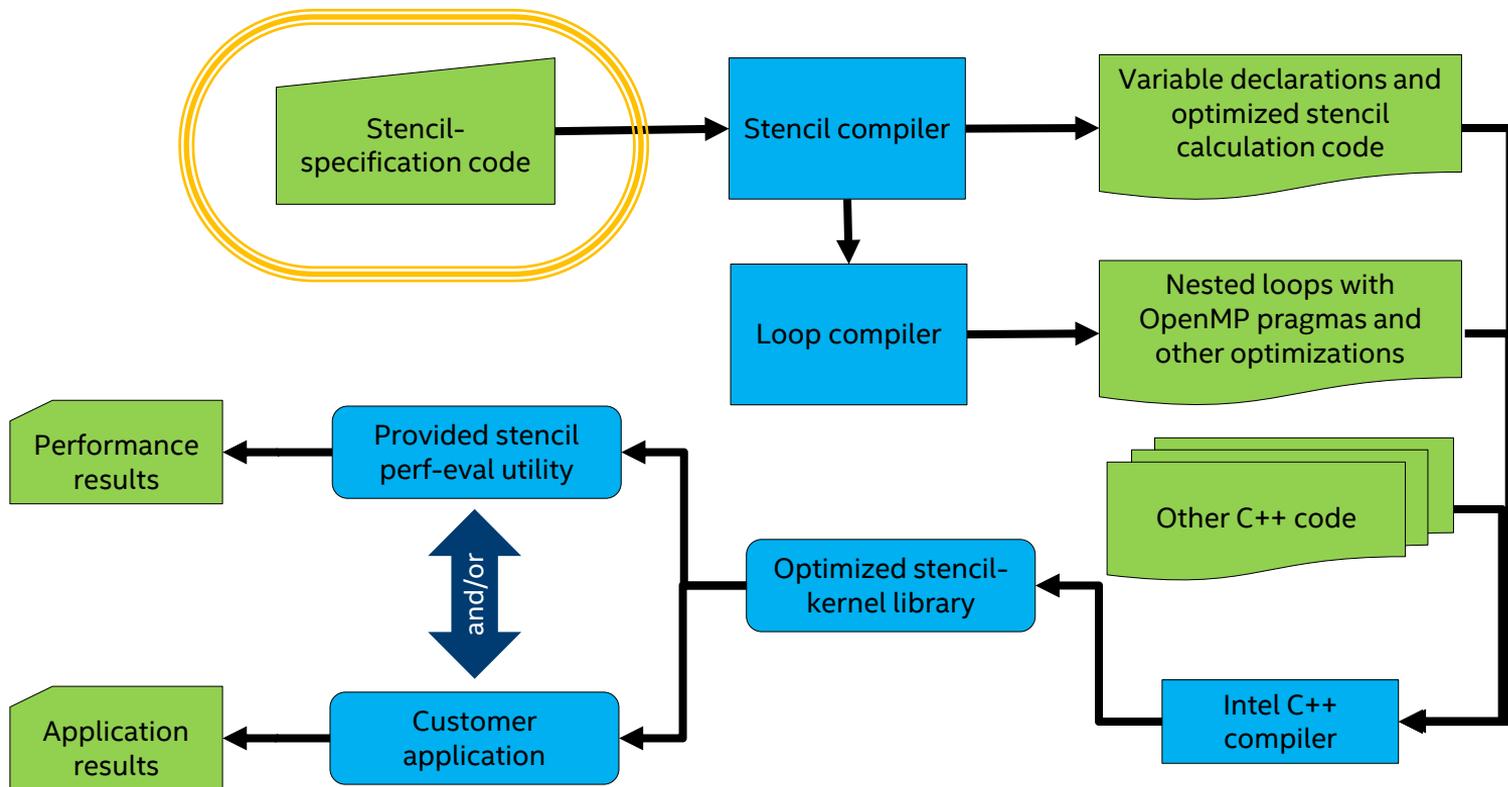
Example simple 25-point 3-D stencil

25 points in 3 spatial dimensions in u(t)

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

$u(t)$ → $u(t+1)$

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

$$u(t+1, i, j, k) = c_0 u(t, i, j, k)$$
$$+ \sum_{r=1}^{4} c_r [u(t, i-r, j, k) + u(t, i+r, j, k) + u(t, i, j-r, k)$$
$$+ u(t, i, j+r, k) + u(t, i, j, k-r) + u(t, i, j, k+r)]$$
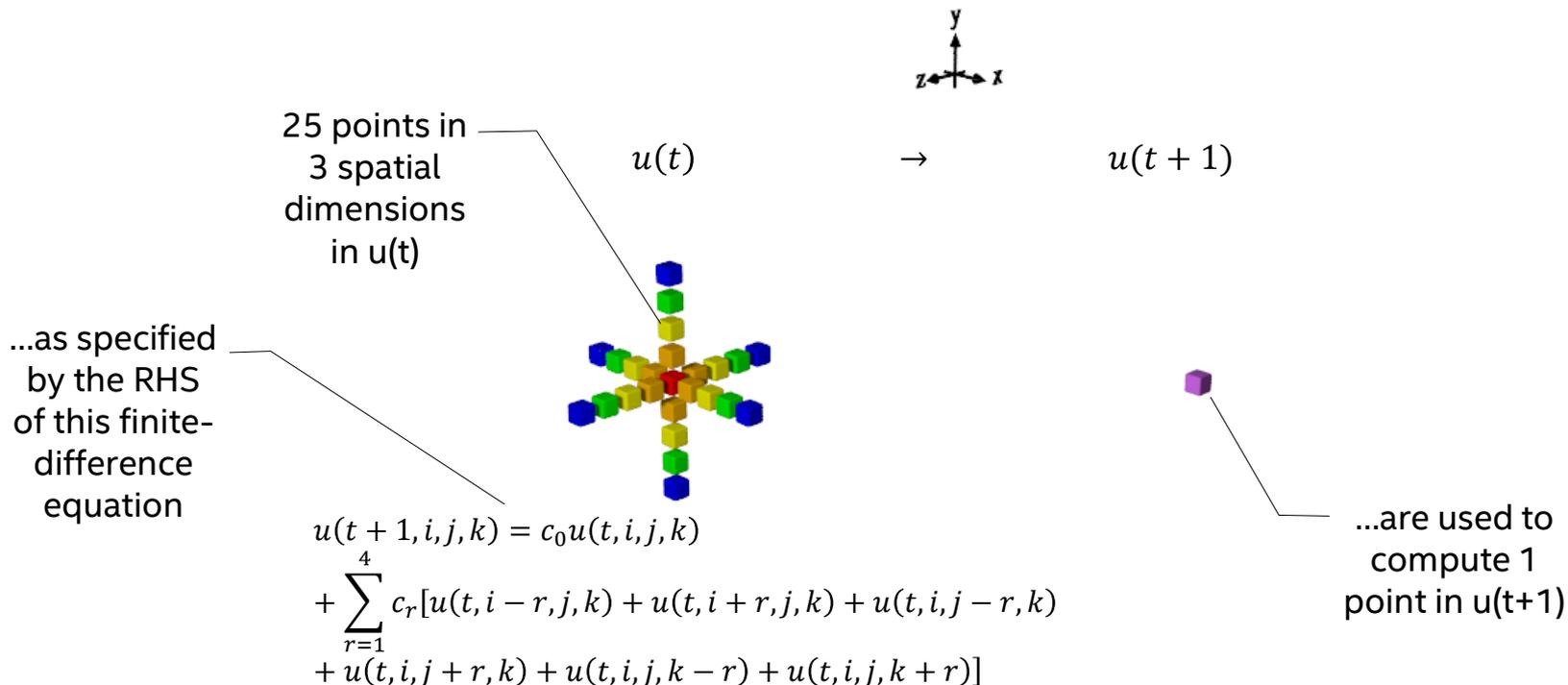
Stencil will be applied over entire problem domain

Entire problem domain—typically millions of points

"Halo" data regions

$u(t)$ → $u(t+1)$

Repeat for u(t+2), u(t+3), …
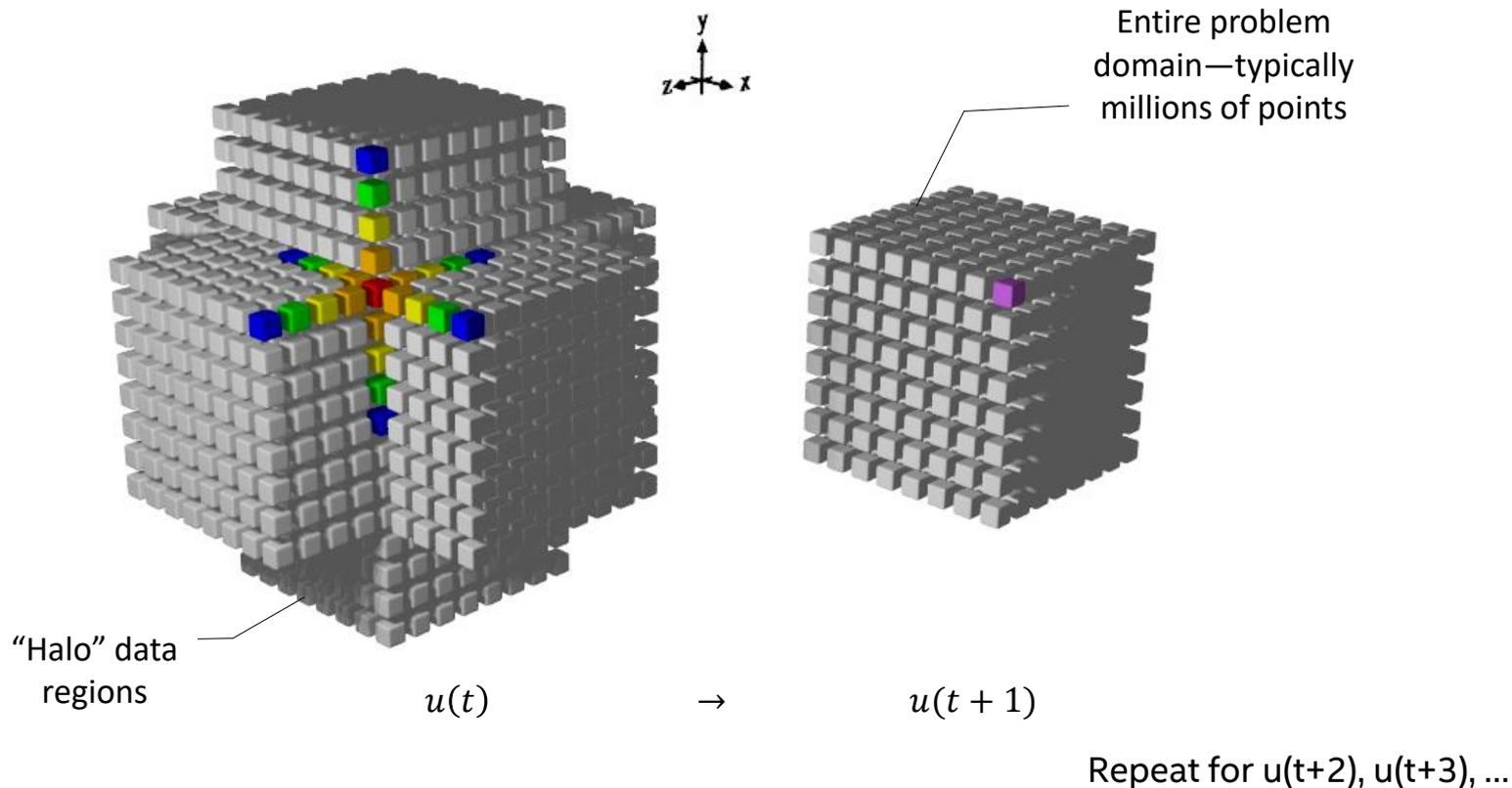
Example stencil DSL code

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

Mathematical description of finite-difference approximation implemented in DSL code

$$u(t+1, i, j, k) = c_0 u(t, i, j, k)$$

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

$$+ u(t, i+r, j, k)$$
$$+ u(t, i, j-r, k)$$
$$+ u(t, i, j+r, k)$$
$$+ u(t, i, j, k-r)$$
$$+ u(t, i, j, k+r)]$$

```cpp
#include "yask_compiler_api.hpp"
using namespace yask;
namespace {
class MyStencil : public yc_solution_with_radius_base {

public:
  MyStencil(int radius=4) :
      yc_solution_with_radius_base("my_stencil", radius) { }

  MAKE_STEP_INDEX(t);
  MAKE_DOMAIN_INDEX(x);
  MAKE_DOMAIN_INDEX(y);
  MAKE_DOMAIN_INDEX(z);
  MAKE_VAR(u, t, x, y, z);
  MAKE_MISC_INDEX(i);
  MAKE_VAR(c, i);

  virtual void define() {
    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;
  }
};

REGISTER_SOLUTION(MyStencil);
}
```

Declare 1 time and 3 space indices and 4D "u" var

Declare misc index and 1D "c" array for coefficients

Overload `define()` method

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

Building a stencil
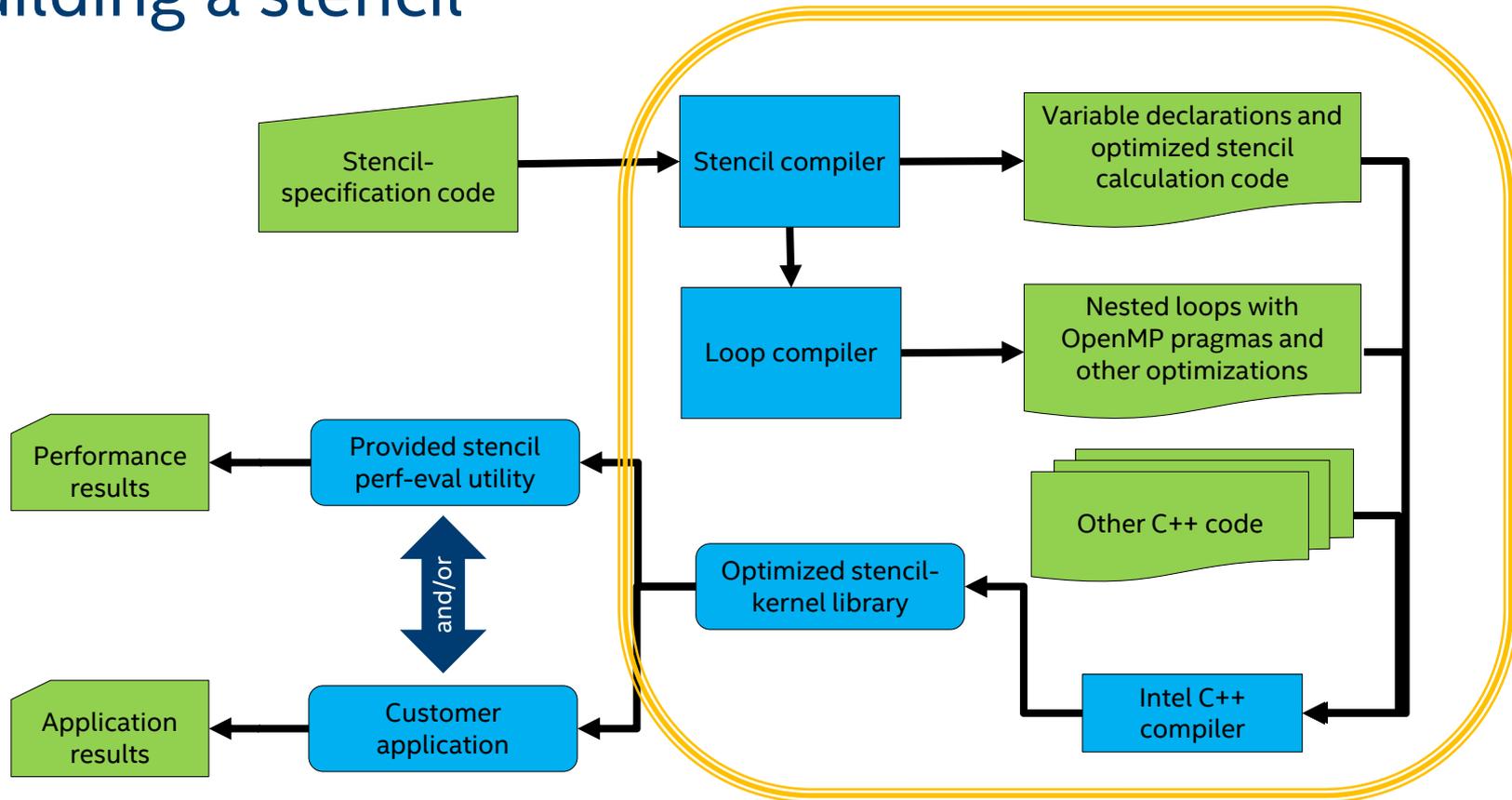
Building your example stencil

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 for a CPU target

- 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.*, `avx2`
    - Targeting a GPU device will be covered later
  - 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
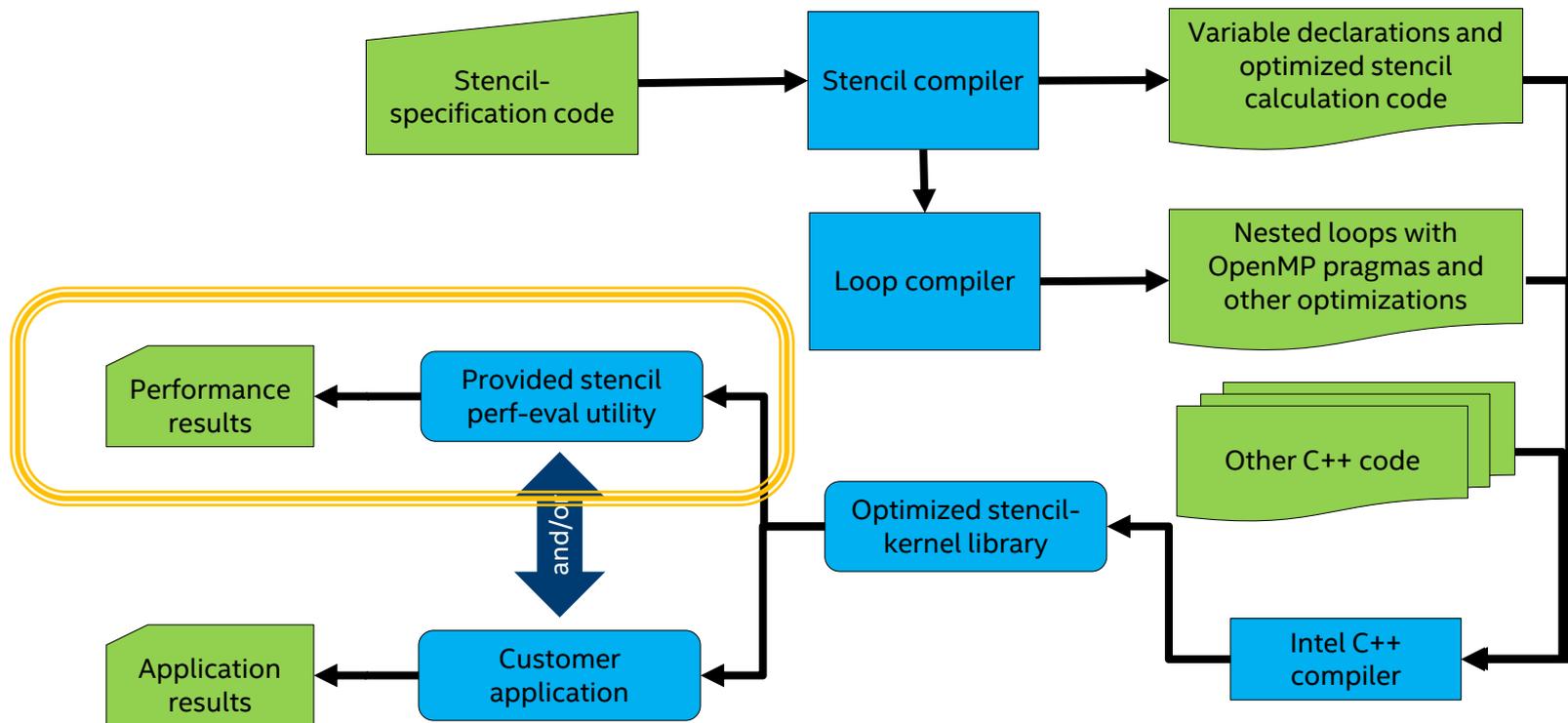
Running a stencil on a CPU platform

Run the provided benchmark 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)
  - Auto-tuner is disabled ; other parameters are set to their defaults
  - The script will use MPI to start a separate process on each NUMA node
    - The global-domain size is automatically decomposed across the processes
    - By default, uses shared-memory (shm) buffers for inter-process communication
      - Override communication type with `-no-use_shm` option, which forces MPI calls between all processes
      - May also have to use this if your platform limits the amount of shm that can be allocated
  - 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
    - The most important performance statistic is usually "`mid-throughput (num-points/sec)`", which is printed near the end of the output
  - A log file is kept in the "`logs`" directory, which includes some compilation and platform information for posterity

Automatically tuning the size parameters

Use the auto-tuner

- `bin/yask.sh –stencil my_stencil -g 512`
  - As on previous slide, but with the block-size auto-tuner enabled (by default)
    - The block size determines the amount of work done by each OpenMP thread at any given time
    - The utility prints each block size that the auto-tuner tries and its measured performance, e.g.,
      ```
      auto-tuner: searching block sizes...
       auto-tuner: search-dist=16: 3.59 steps/sec (2 steps(s) in 556.7m secs) with size x=96 * y=28 * z=96 -- best so far
       auto-tuner: search-dist=16: 3.74 steps/sec (2 steps(s) in 533.7m secs) with size x=32 * y=4 * z=80 -- best so far
       …
      ```
    - When the tuner is done, it prints the final sizes, e.g.,
      ```
       …
       auto-tuner: applying size t=0 * x=96 * y=4 * z=80
       auto-tuner: done
      Waiting for auto-tuner to converge on all ranks...
      Auto-tuner done after 73.62 secs
      Final settings:
       auto-tuner: mega-block-size:        t=0 * x=512 * y=512 * z=512
       auto-tuner: block-size:             t=0 * x=96 * y=4 * z=80
       auto-tuner: micro-block-size:       t=0 * x=96 * y=4 * z=80
       auto-tuner: nano-block-size:        x=96 * y=4 * z=80
       auto-tuner: pico-block-size:        x=96 * y=4 * z=80
      ```
    - Note that the tuned block size printed here is different than at the beginning of the run
    - Using other auto-tuner options, the other types of blocks (explained later) can be tuned
  - Compare output and performance of the two runs with and without the auto-tuner

Hand-tuning the size parameters

Block size

- From the log file of the last run, find the best block size found by the auto-tuner as highlighted on the previous slide
- 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 4 –bz 80
    ```
  - 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 domain sizes, e.g., `–b`
- 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
  - The long list of options might be overwhelming; we'll point out the most important ones in this tutorial

Scaling out to multiple nodes in a cluster

Domain decomposition

- Many HPC problem sizes won't fit in the DRAM memory of one platform
- Most HPC problems can be divided spatially across multiple nodes in a cluster via *domain decomposition*

YASK implementation

- Data exchanges use 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 MPI ranks
  - 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 ranks

Conceptual view of 2D, 2×2 rank domains

Rank 0 Rank 1

Rank 2 Rank 3

Read/write "domain" regions

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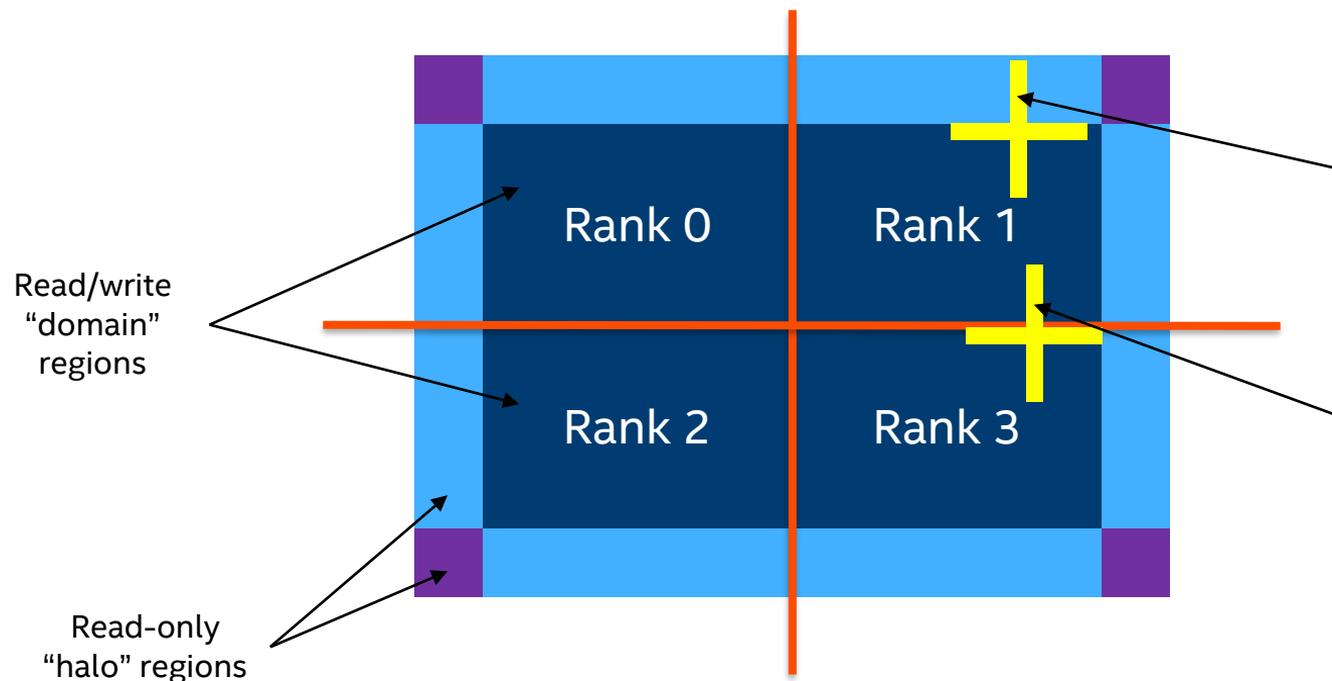
Due to the read-radius of stencils, halo data must be read to calculate some points in the domain.

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.

Rank 0

Rank 1

Rank 2
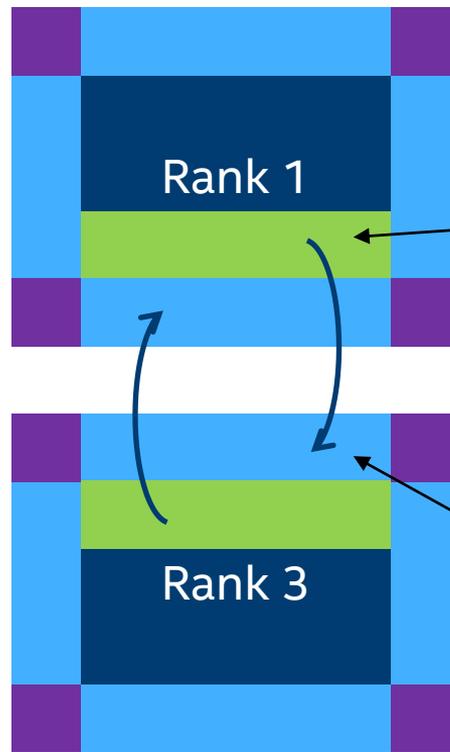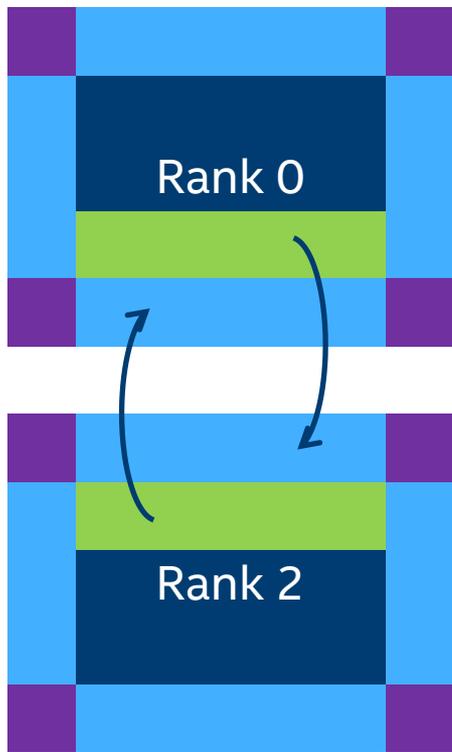
Rank 3

Data calculated in rank 1 at time-step *n*…

…is needed by rank 3 at time-step *n+1*

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

Width of halos is based on radius of stencil in each dim.
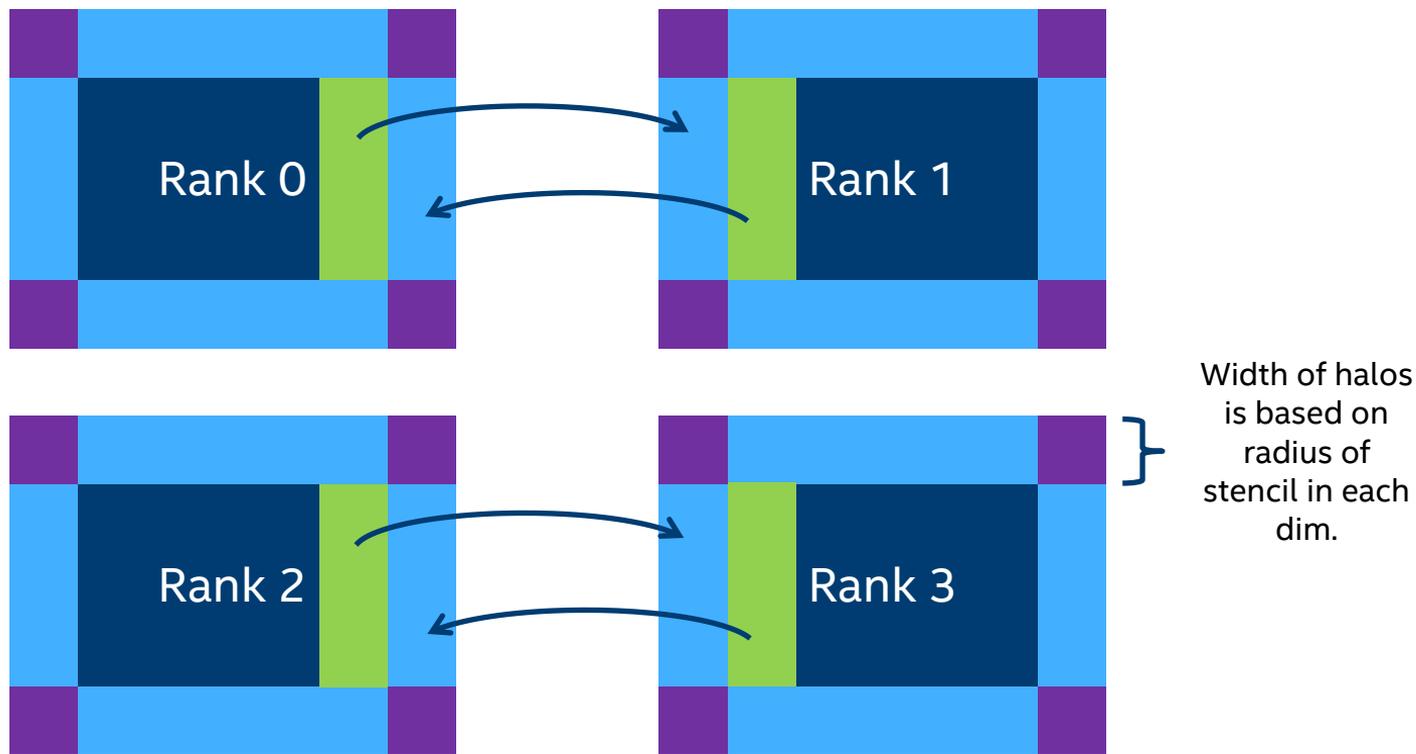
Schematic of 2D, 2×2 rank corner exchanges

Corner exchanges are only needed for halos that have diagonal points.
This is determined automatically by the stencil compiler

Rank 0

Rank 1

Rank 2

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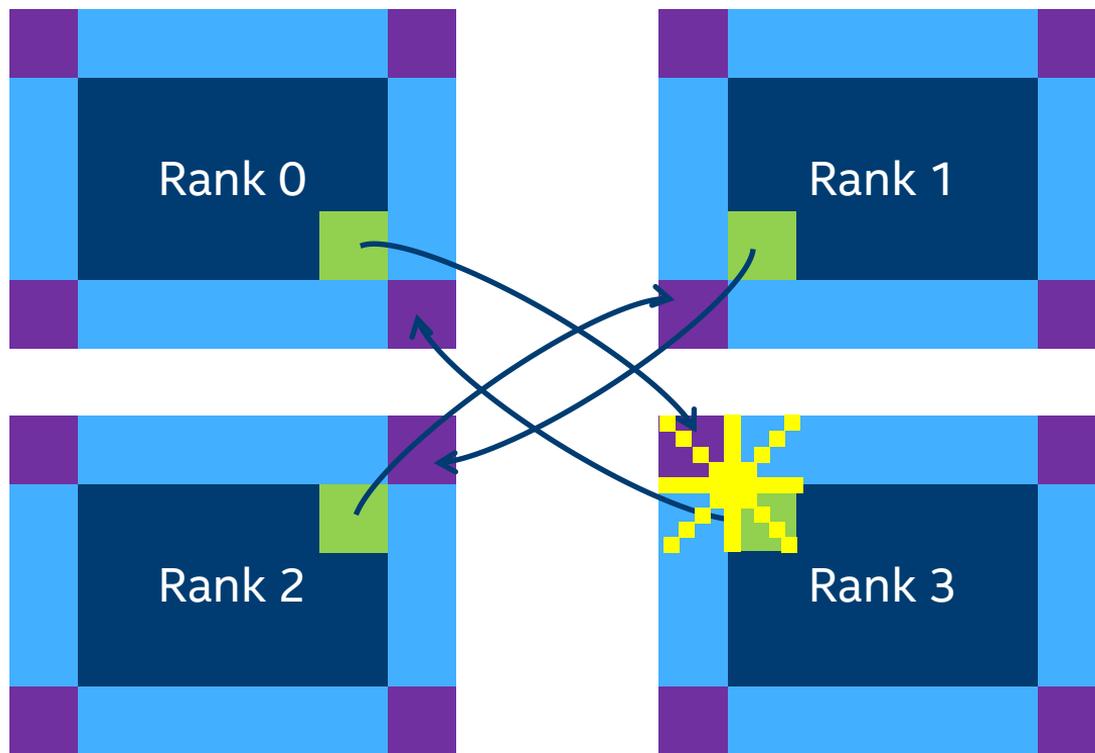
For 3D stencils, exchanges may be needed along faces, edges, and corners of the rectangular solids.

This concept can be extended to any number of dimensions.

Running on multiple nodes in a cluster

Running via Slurm

▪ If a Slurm job is requested with the proper number of nodes and MPI tasks, the `bin/yask.sh` utility should issue the proper `mpirun` command

Running the test utility with an explicit MPI command

▪ `bin/yask.sh` **-mpi_cmd 'mpirun <…>'** `-stencil my_stencil -g 1024`

   – Replace <…> with the specification of host-names, tasks per node, and other needed parameters
      – Use the technique for your scheduler, e.g., `-f` or `-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 in each cluster node
   – YASK will automatically partition the global domain and distribute it among the MPI ranks
      – Use the `-nr`* YASK 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

Implementation options

▪ For typical configurations, YASK overlaps communication with some computation using this sequence [simplified]:

   1. Only the data needed by another rank is calculated on each rank for a given time-step and packed for delivery

   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 and the received data is unpacked before the next time-step is begun

   – You can disable this feature via `-no-overlap_comms` to measure the impact

▪ Shared-memory (shm) communication will automatically be applied between ranks that share a virtual address space, but MPI can be forced between all ranks with `-no-use_shm` as explained earlier

USING THE APIS

Making a stencil application
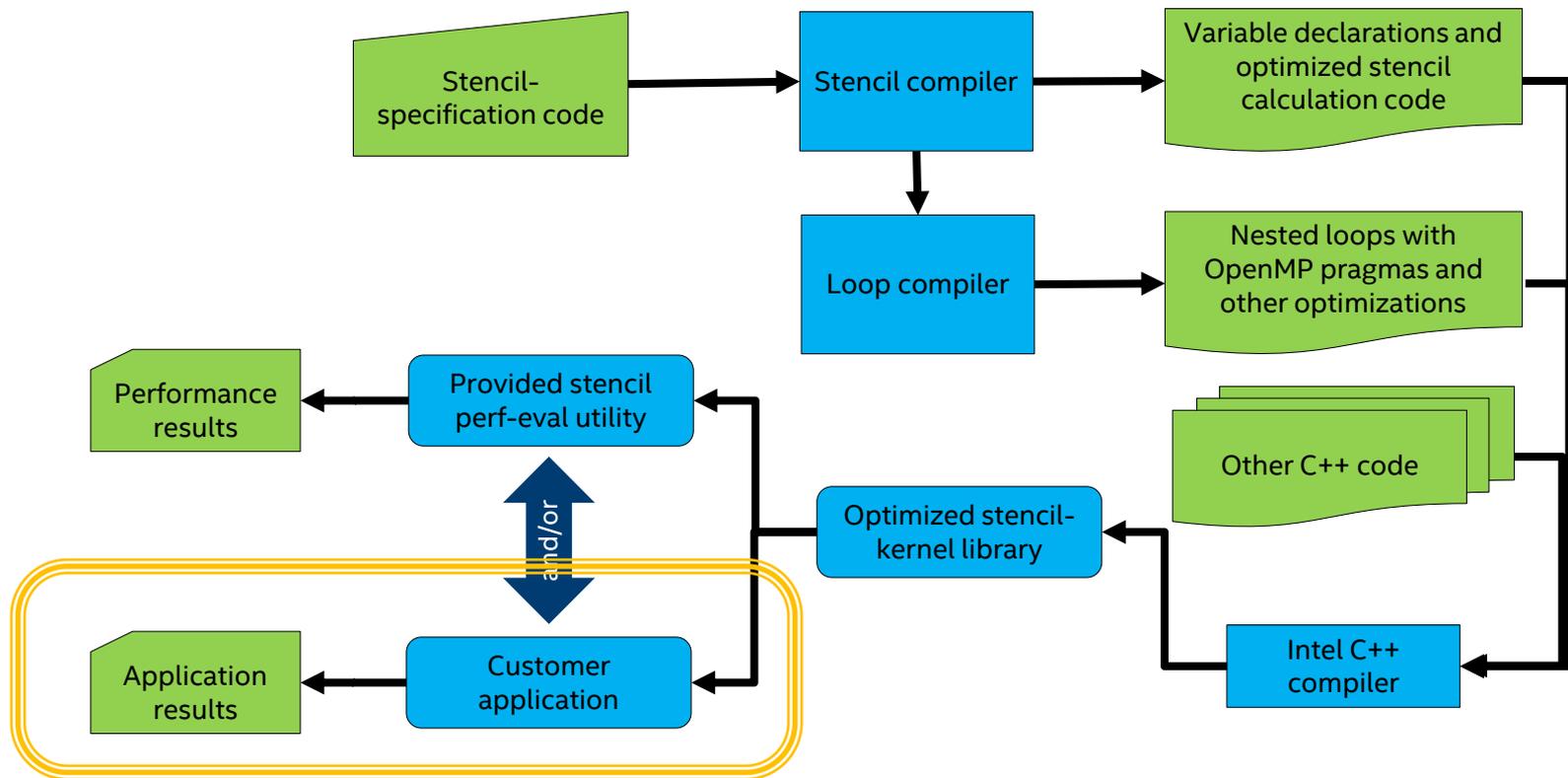
API overview

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
  - <u>This is the best place to start looking for the API you want</u>
  - **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

API combinations

There are two main steps in using a YASK stencil:

1. Create a library from the YASK DSL via the "compiler" APIs

2. Use the library in an application via the "kernel" APIs

For each of these steps, there are three options (making 9 combos):

- Use the binary provided in the YASK package

- Write your own app in C++

- Write your own app in Python

The most common combos:

1. Compiler: Use the provided YASK compiler binary to create a library for your stencil via the `Makefile`

2. Kernel:
   - Use the provided YASK binary to test and tune your stencil's performance by running `yask.sh`
   - Create your own binary using the C++ kernel APIs to use the stencil library in production

```
Start
  │
Generate stencil library
  │
  or
 ╱  ╲
Write DSL code and use          Create your own YASK
bin/yask_compiler               compiler via APIs in
via make                        C++ or Python (unusual)
 ╲  ╱
Use stencil library
  │
and/or
 ╱  ╲
Run bin/yask.sh          Create your own app
                         via APIs in C++ or Python
 ╲  ╱
Done
```
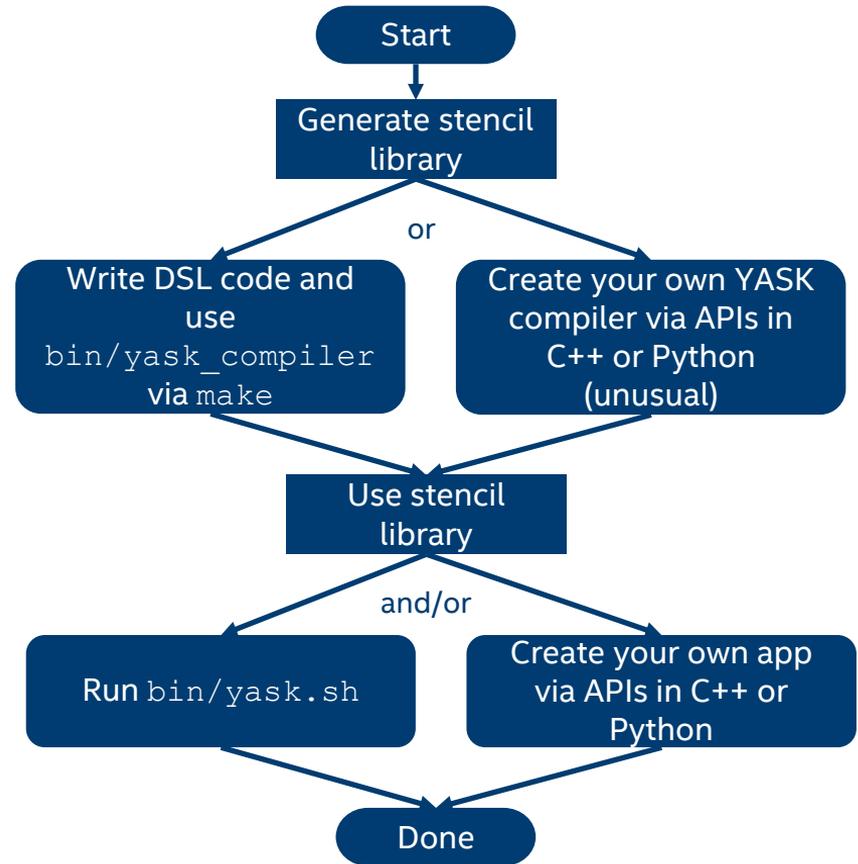
Access to the APIs

C++

- The standard method of writing YASK stencils uses the "DSL" part of the C++ compiler API library
  - Many examples in the `src/stencils` directory
  - The provided YASK compiler generates C++ stencil code from the DSL code; this is compiled into a kernel library
  - You only need to make your own YASK compiler binary in special situations, such as creating a custom stencil generator; otherwise, just build your stencil from the `Makefile` and use the generated library
    - The rest of this section in this tutorial covers using the kernel APIs, not the compiler APIs
- When you run `make` as in the previous examples, you have implicitly made a kernel library containing the kernel APIs
  - A shared object is created in the `lib` directory, one for each stencil and target architecture
  - The performance-test binaries we've been running via `yask.sh` are just wrappers 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

Python

- Build the APIs for Python* only if you want to make a Python app instead of a C++ app
- 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—use the C++ docs with appropriate type mapping
- Performance-wise, the stencil calculations will be the same speed as in C++, but data access is slower

Main YASK kernel process steps

Create a YASK "solution" object → Set problem size and other parameters → "Prepare" the solution: allocate memory and sync meta data

Initialize data in YASK variables → Run solution for n time-steps

Inject data representing external stimulus if needed, etc.

Done? — No → Run solution for n time-steps

Done? — Yes → Extract results from YASK variables
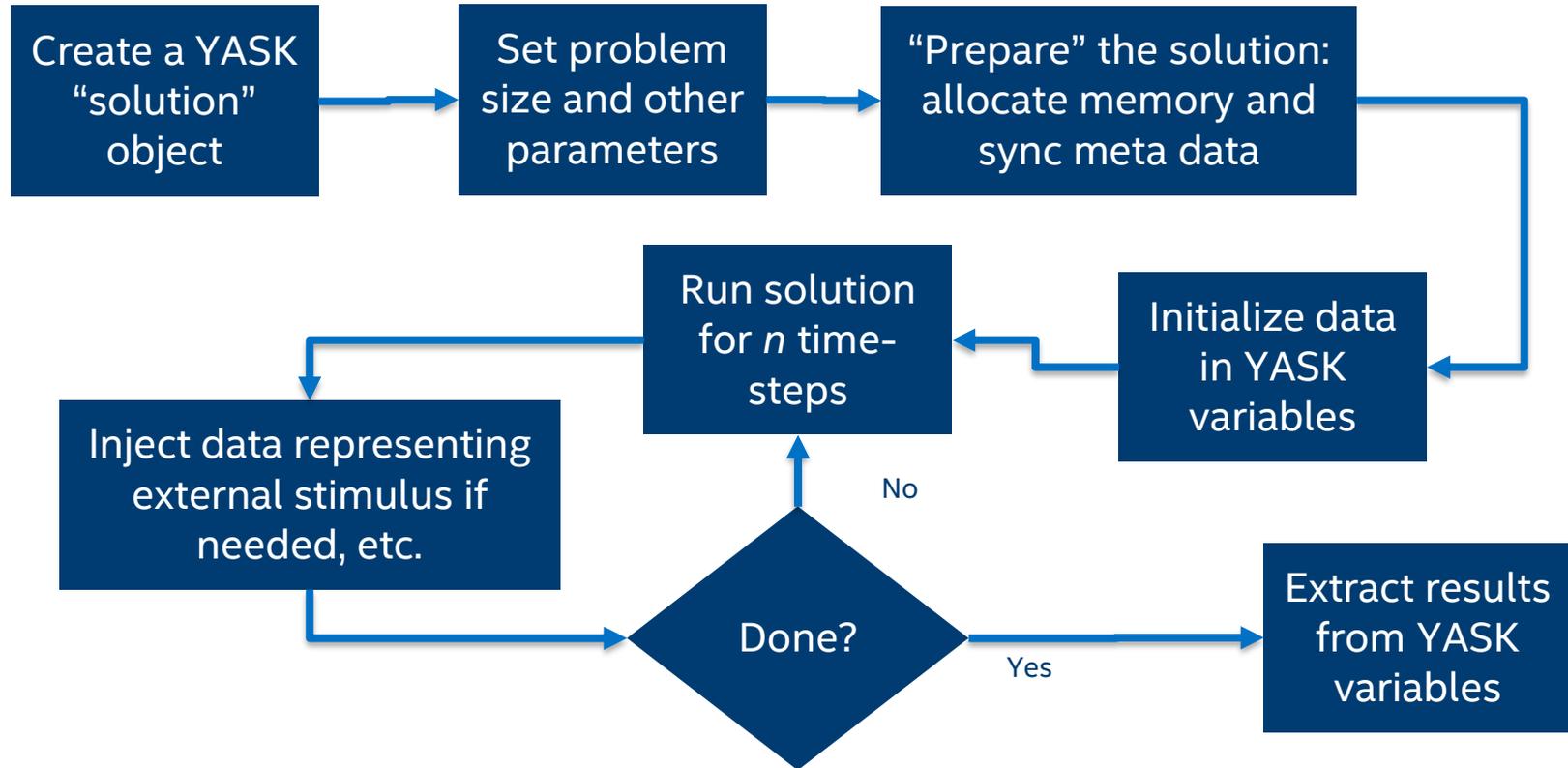
Creating a YASK kernel via the APIs

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

FAQ: 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 `yk_solution::disable_debug_output()` to simply discard it
  - Use a `yask_output_factory` to create new output streams
  - NB: this mechanism was used instead of raw C++ streams to enable Python usage, but you can get access to the C++ stream via `yk_env::get_debug_output()->get_ostream()`

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:** often "t" for "time"
  - **Domain dim:** often 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 and fixed 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

Important: Halo regions are *not* considered part of the domain size

Rank 0

Rank 1

Rank 2

Rank 3

Global-domain size in y dim

Local-domain size in x dim in ranks 1 and 3
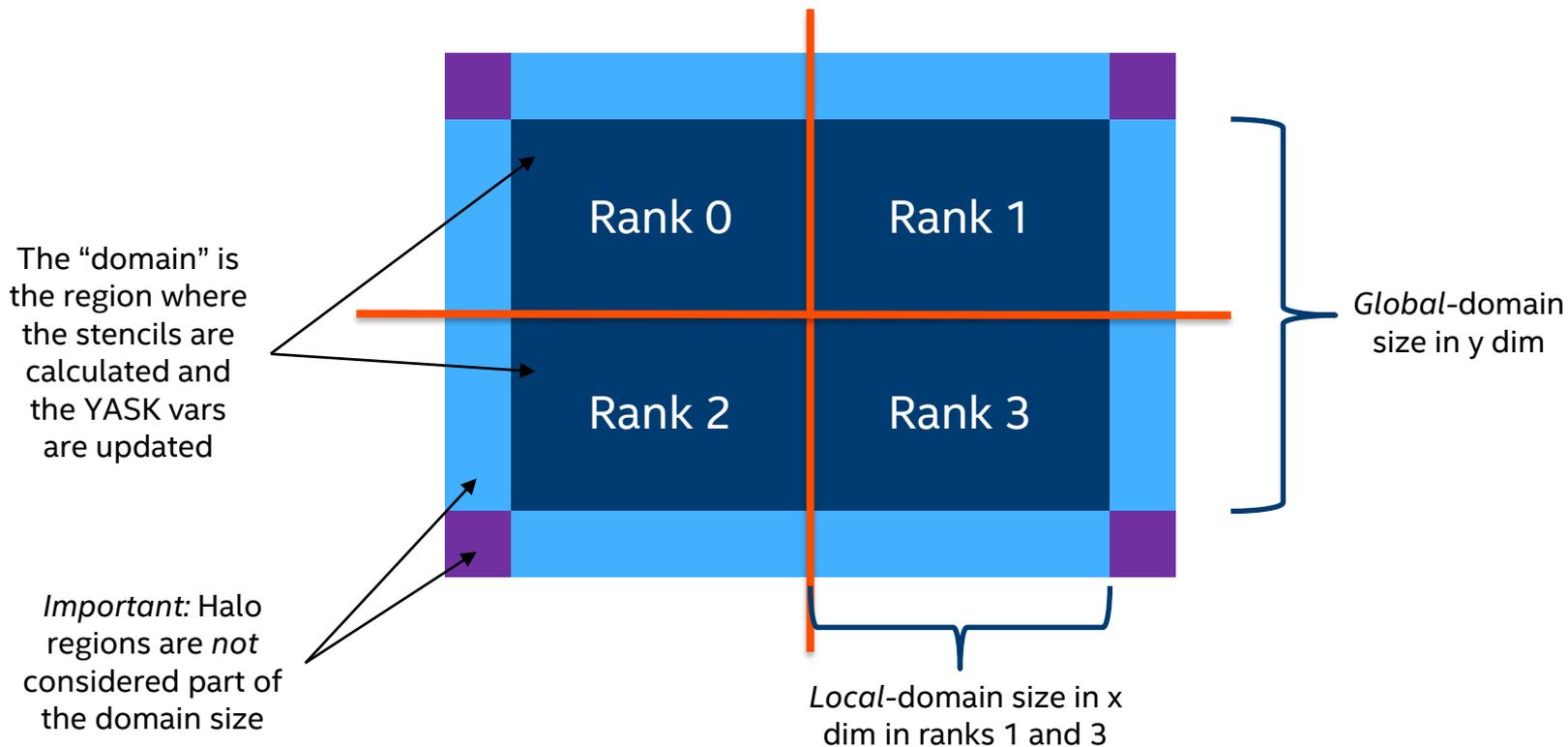
Per-rank view of 2D problem sizes

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

Rank 0

Rank 1

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

Halo size—can be different for each variable, dimension, and side of domain; determined automatically by stencil compiler

Rank 2

Rank 3

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
- Other solution parameters, including other tile sizes, can be set via `yk_solution::apply_command_line_options()`

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
  - NB, 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

get_rank_domain_size("x") → 100

get_left_halo_size("x") → 5

get_right_extra_\
padding_size("x") → 3
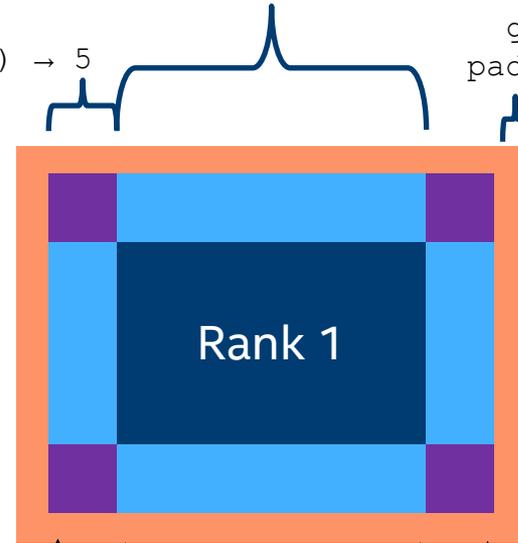
Important: <u>var indices are always global</u>!

Note: some may be negative.

Rank 0

Rank 1

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

get_first_rank_\
halo_index("x") → -5

get_first_rank_domain_index("x") → 0

get_last_rank_domain_index("x") → 99

get_last_rank_halo_index("x") → 104

get_last_local_index("x") → 107

get_first_local_index("x") → 92

get_first_rank_halo_index("x") → 95

get_first_rank_domain_index("x") → 100

get_last_rank_domain_index("x") → 199

get_last_rank_\
halo_index("x") → 204

Notes on initializing data

Of course, load data into the domain areas

Rank 0

Rank 1

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
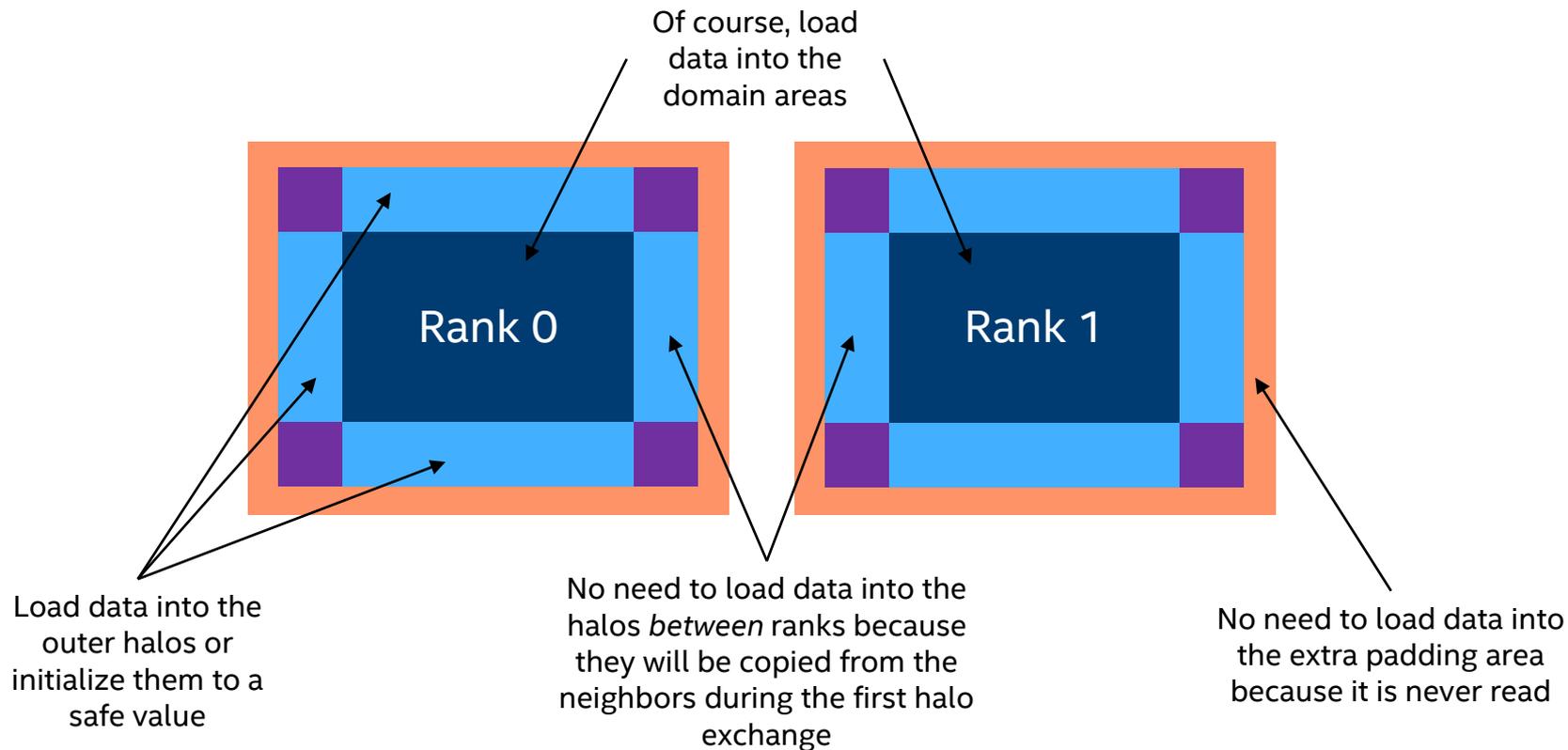
Make the stencil calculations and get results

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
  - Examples: `get_element()`, `get_elements_in_slice()`, `reduce_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.

Example applications

2D Wave equation

- Stencil DSL: src/stencils/Wave2DStencil.cpp
- Application code: src/examples/wave_eq_main.cpp
- Illustrates
  - Sub-domains and scratch vars (see next section on advanced stencils)
  - Using both YASK-library and application-specific options
  - Determining global and local domain sizes
  - Initializing YASK var data efficiently by slices
  - Running the simulation loop, and calculating reductions and reporting simulation stats within the simulation loop
  - Retrieving final simulation results and reporting performance

2D Shallow-water equation

- Stencil DSL: src/stencils/SWE2DStencil.cpp
- Application code: src/examples/swe_main.cpp
- Builds upon 2D wave-equation example to additionally illustrate
  - Different sub-domains for various vars
  - Restricting certain calculations to specific simulation steps controlled by the simulation loop

Run "make examples" to build and "make example-tests" to run validation tests

- Use "-bench" and "-g" knobs to run benchmarks of various sizes
- Use "mpirun" and "-outer_threads" knobs to tune for number of NUMA nodes and cores per NUMA node

Advanced APIs

Misc interesting APIs (not 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; may be useful for checkpoint/restore of vars, etc.

- 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 or for setting options that do not have corresponding APIs

- 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 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 multiple passes over the domain, called "stages"
- YASK automatically determines the dependencies between equations in the DSL and schedules their stencils

Boundary regions

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
  - This stencil illustrates the use of combining 3D and 2D variables to save memory
  - *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 special stencils in boundary regions 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 < first_domain_index(x) + 5)` defines a sub-domain on the left side of the domain, 5 elements wide
  - `(x > last_domain_index(x) - 5)` would be the same on the right side
  - `(x >= first_domain_index(x) + 5) && (x <= last_domain_index(x) - 5)` would be used for the interior sub-domain between the left and right boundary sub-domains
- These conditions are placed after the stencil equation using the `IF_DOMAIN` operator in the DSL
  - Example: `A(t+1,x) EQUALS A(t,x) + A(t,x+1) IF_DOMAIN x > 10`
  - Search for `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

Sub-domain described by
`(x < first_index(x) + 5)`
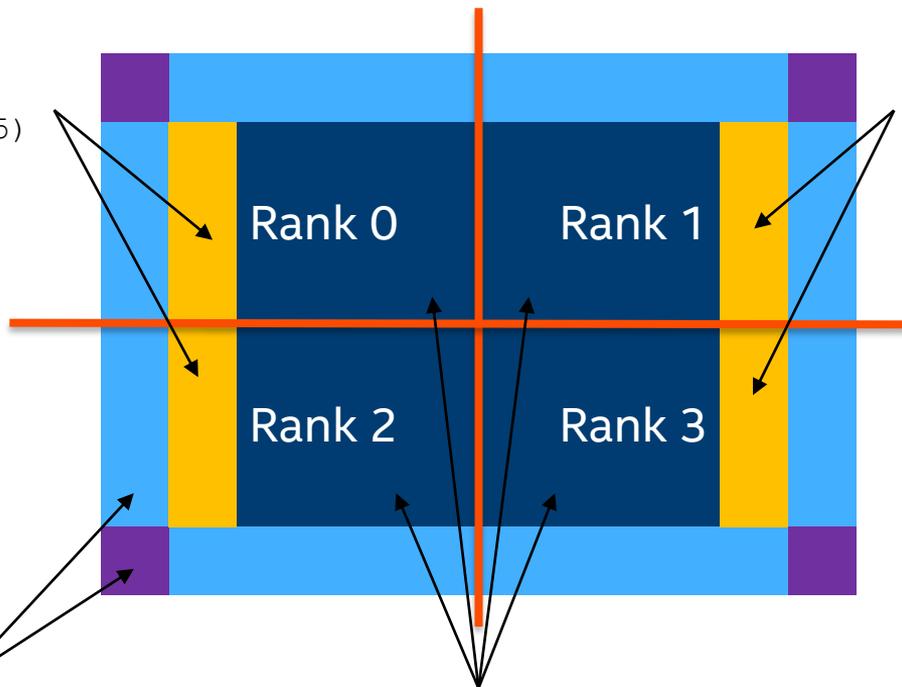
Sub-domain described by
`(x > last_index(x) - 5)`

Rank 0   Rank 1

Rank 2   Rank 3

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

Sub-domain described by
`(x >= first_index(x) + 5) &&`
`(x <= 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 % 8 == 0)` is true every 8th time-step
- These conditions are written after the equation using the `IF_STEP` operator in the DSL
  - Example: `A(t+1,x) EQUALS A(t,x) + A(t,x+1) IF_STEP t > 5;`
  - Look for TestStepCondStencil in `src/stencils/SimpleTestStencils.cpp` for a simple example
  - To apply both sub-domain and step conditions, put parentheses around the value definition and the first condition, e.g.,
    - `( A(t+1,x) EQUALS A(t,x) + A(t,x+1) IF_DOMAIN x > 10 ) IF_STEP t > 5;`

Scratch variables

Purpose

- Provide a mechanism to calculate intermediate values efficiently

Usage

- Often, stencils create intermediate values that do not need to be accessed across simulation steps
  - Rather, these intermediate values are then used in subsequent stencils to calculate the final desired values
- These intermediate values can be stored in "scratch" variables instead of regular YASK variables
- Scratch variables cannot have the step dimension since they do not exist across steps
- Unlike regular variables, the values of all possible points must be defined, *including those outside of the YASK domain*
  - When defining scratch values without conditions, this happens normally, but be careful when using conditions on scratch vars
  - As a development step or debug aid, the option `-init_scratch_vars` may be used to set all the values to zero in each scratch var before calculations are done, but this usually results in poor performance because most values in each scratch var will be written twice
- See `src/stencils/Wave2dStencil.cpp` for example usage, e.g., `u_sub1`

Implementation

- Unlike regular vars, scratch-var memory is not allocated to cover the whole domain
  - Memory is automatically allocated by the YASK runtime to cover only a block (more accurately, a micro-block as described later)
  - The memory is reused as each micro-block is evaluated sequentially by each thread (actually, each outer thread as described later)
  - There is a separate scratch-var memory allocation for each outer thread
- YASK can also reuse the same memory allocation for more than one scratch variable when possible
- All these implementation details contribute to increasing cache locality and decreasing memory usage when using scratch variables

Scratch-variable examples

Without sub-domain or step conditions

```
MAKE_VAR(A, t, x, y);
MAKE_SCRATCH_VAR(S1, x, y);
…
S1(x,y) EQUALS (A(t,x,y) + A(t,x+1,y)) / 2;
A(t+1,x,y) EQUALS (S1(x,y) + S1(x,y+1)) / 2;
```

With sub-domain conditions

```
MAKE_VAR(A, t, x, y);
MAKE_SCRATCH_VAR(S1, x, y);
auto d1 = x>=first_domain_index(x) && x<=last_domain_index(x) &&
  y>=first_domain_index(y) && y<=last_domain_index(y);
…
S1(x,y) EQUALS (A(t,x,y) + A(t,x+1,y)) / 2 IF_DOMAIN d1;
S1(x,y) EQUALS 0.0 IF_DOMAIN !d1; // Define all possible points
A(t+1,x,y) EQUALS (S1(x,y) + S1(x,y+1)) / 2;
```

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 many systems, esp. those containing high-bandwidth memory (HBM)
- 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 up to 1.5× speedup from vector-folding

Traditional 1D Vectorization

Inner 3D loop iterates in x direction, i.e., *same dimension* as vectorization
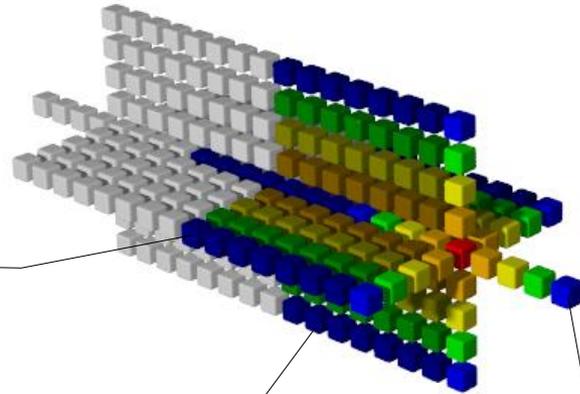
8 new vectors must be read for k±r points
(4 for k+r and 4 for k-r for r=1..4)

8 new vectors must be read for j±r points

Only 1 new vector must be read for i±r points due to overlap along x axis

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

4 new 4x2x1 vectors must be read for j±r points
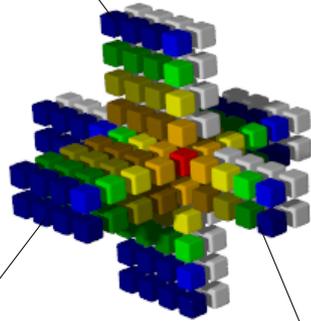
Only 1 new vector must be read for k±r points

2 new vectors must be read for i±r points
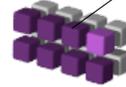
Inner 3D loop iterates in z direction, i.e., *perpendicular* to 2D vector

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

5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	…
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	…
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	…
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	…
1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	…

y

x

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

Load and permute instructions generated automatically by YASK stencil compiler

Layout in memory (1D)

| 1,1 | 1,2 | 1,3 | 1,4 | 2,1 | 2,2 | 2,3 | 2,4 | 1,5 | 1,6 | 1,7 | 1,8 | 2,5 | 2,6 | 2,7 | 2,8 | 1,9 | … |

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

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 1×2×8 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 (*i.e.*, the fold you specify is a *hint*)
  - The vector length in any dimension not specified defaults to one (1)
  - The default fold varies depending on dimensionality, architecture, and FP precision
    - Is 4×4×1 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, `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, `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 `make clean`) ☞

Misc settings

- Any preprocessor macro can be set by `make … EXTRA_MACROS='name=value …'`
- Type `make help` for examples of settings the C++ optimization level and more

More grouping terms: parts and stages

You may notice the terms "parts" and "stages" in the YASK debug output

Parts

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

Stages

- A *stage* is the term used for a group of parts that are not inter-dependent
- This can occur when there are equations in different sub-domains that are not dependent on one another
- Multiple stages are scheduled in the correct order based on dependencies (e.g., in a staggered-grid problem)
- The amount of work and performance of each stage is listed in the debug output

Temporal wave-front tiling

Goal

- Increase temporal locality in caches at a package level
- Especially useful for large unified cache (such as HBM used for caching DDR memory)

Technique

- In each MPI rank
  - Evaluate n time-steps within a subset of the local rank domain
    - We refer to this subset as a *Mega-block* in YASK
  - Evaluate each Mega-block *sequentially* until entire rank domain is evaluated for n time-steps
- Some redundant work is done between ranks to allow halo exchange after every n time-steps
- YASK handles all the temporal tiling complexity automatically

Temporal wave-front dependencies

Key

○	value already known
⊠	value not yet known
○ (orange)	known value used as input
■ (green)	value being calculated
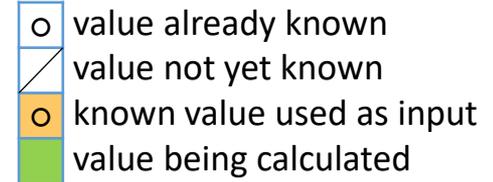
Highest x-value that may be calculated for t=4 due to the dependencies on t=3 values

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

Tile ends here for t=1
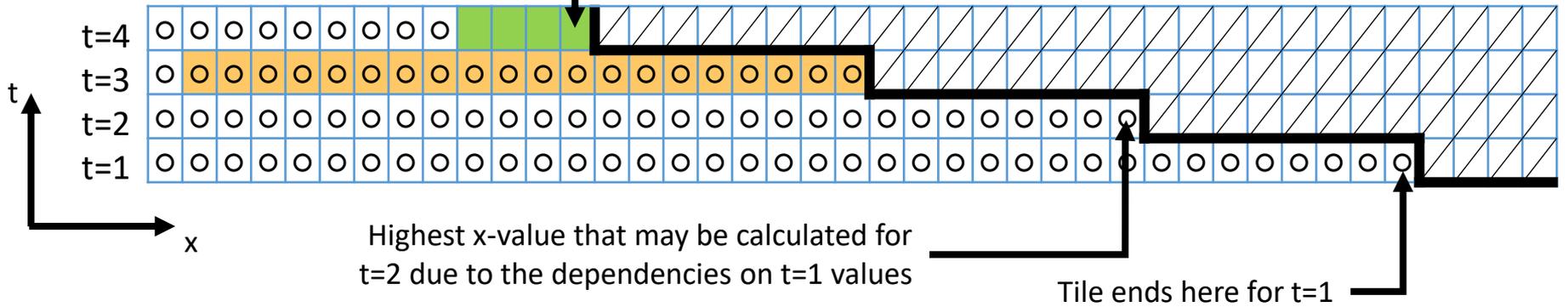
- In the first temporal wave-front tile shown here, the number of values that can be calculated is reduced for each time-step (and/or each stage within a time-step)
- The amount of shift is called the *wave-front or temporal angle* and is based on the radius of the largest stencil (shift of 8 shown)

Covering a temporal range via multiple tiles
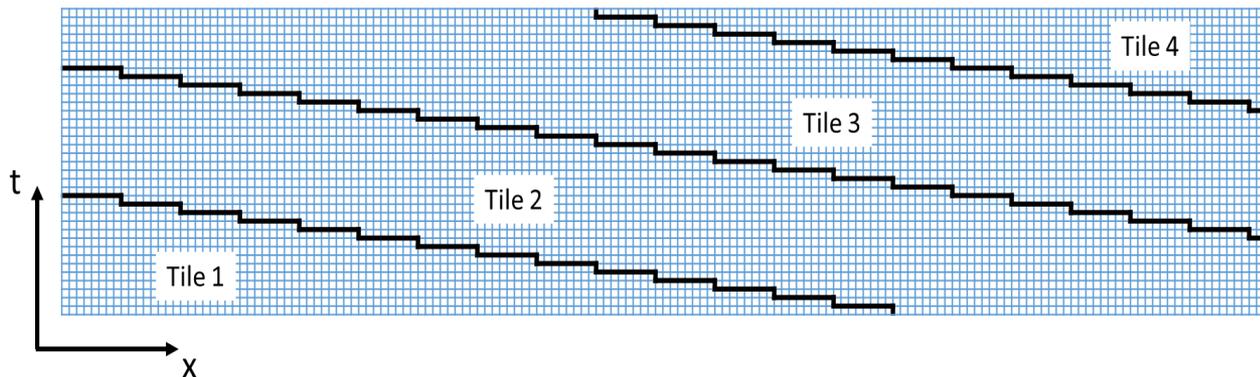
- Wave-front tiles (Mega-blocks) are computed *sequentially,* but multiple values within one time-step of a given tile may still be evaluated *concurrently*
- After last tile is complete, YASK variables contain the same data as they would have without temporal tiling, so no change is needed when inputting data or reading results
- This concept is directly extended to 2D or 3D stencils by shifting in each spatial dimension

Using temporal wave-front 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 *Mega-blocks*, where each Mega-block is a wave-front tile
- Control the spatial size of a Mega-block with `-Mb, -Mbx, -Mby,` etc.
  - Be sure to use a capital "M"; a lower-case "m" is for micro-blocks, explained later
- Control the temporal size of a Mega-block with `-Mbt` (assuming your time dim is "t")
- By default, the spatial size of a Mega-block is the size of the rank

Selecting values

- When using a very large unified cache (e.g., HBM used to cache DDR memory), set the spatial size to fit within this cache
  - See the paper in the upcoming reading list showing ~2× 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-level 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 Mega-blocks, and a Mega-block is composed of blocks
- Evaluate blocks *concurrently* until entire Mega-block is evaluated for n time-steps

Temporal block dependencies

Key
- o value already known
- / value not yet known
- o known value used as input (orange)
- value being calculated (green)
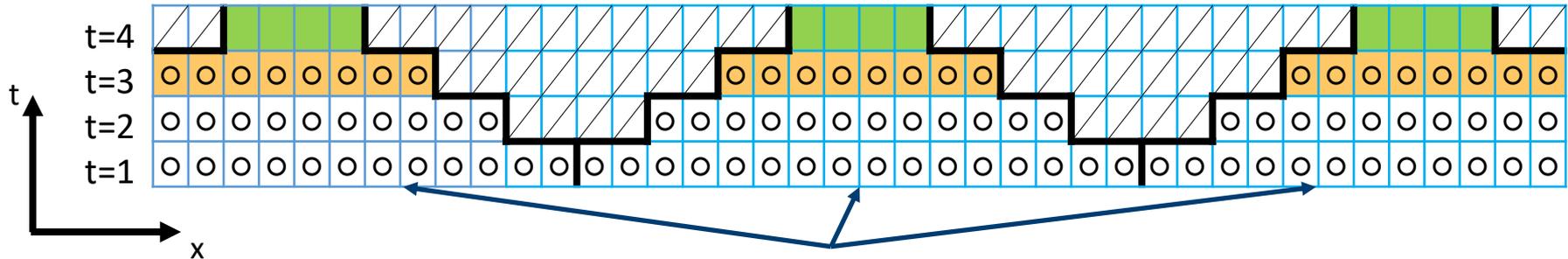
These three "upward" triangle blocks may be evaluated concurrently.
When they are complete, the "downward" triangle blocks are evaluated.

- In one spatial dimension (as shown), this is called triangle or trapezoid tiling (half of diamond tiling)
- Extension into multiple spatial dimensions uses 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)

Using temporal block tiling in YASK

Not another level of tiling hierarchy

- 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, there is no temporal block tiling

- Temporal block tiling occurs *within* wave-front rank tiling, so the *temporal* size of a Mega-block defaults to the temporal size of a block

  - But the default *spatial* size of a Mega-block is still the spatial size of a rank domain

  - You can use both wave-front rank tiling and temporal block tiling, and the temporal size of the Mega-blocks can be larger than the temporal size of the blocks

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 micro-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 there are many variables in a stencil, which requires small blocks to fit in cache
  - This can lead to many short OpenMP tasks, which may not be optimal
- Also, it may be beneficial in some cases to combine the concepts of trapezoid 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 *micro-block* in YASK
- Evaluate each micro-block *sequentially* until entire block is evaluated

Temporal micro-block dependencies

Key

- value already known
- value not yet known
- known value used as input
- value being calculated

Block boundaries as shown previously

Mini-block boundaries
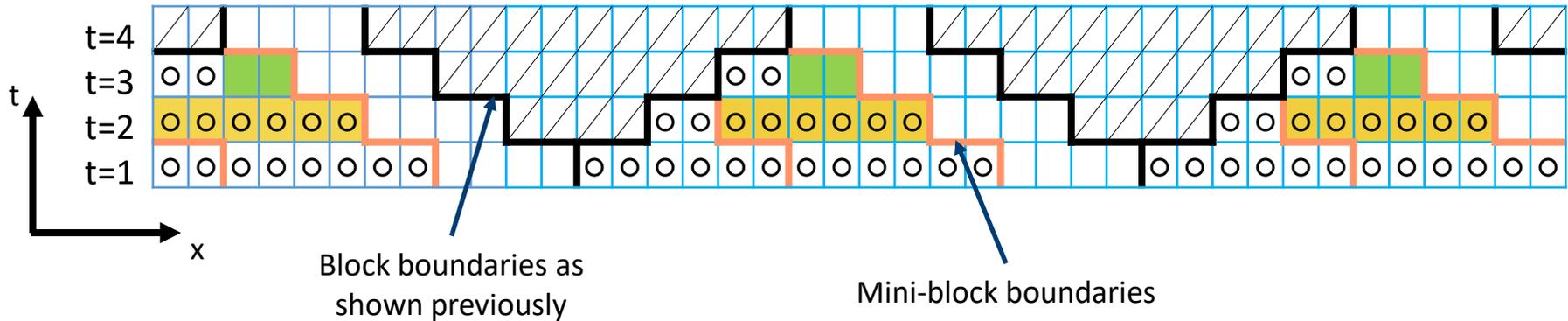
- Blocks are evaluated *concurrently* as before using OpenMP threads
- Micro-blocks are evaluated *sequentially* within each block
- Micro-blocks use wave-front tiling, similar to Mega-blocks, but inside *blocks* instead of *ranks*

Using micro-block tiling in YASK

Another level of tiling hierarchy

- Blocks may be divided into *micro-blocks*, where each micro-block is a wave-front tile
- Control the spatial size of a micro-block with `-mb, -mbx, -mby,` etc.
- The *temporal* size of a micro-block is always the same as a block, so it is set implicitly via `-bt`
- By default, the *spatial* size of a micro-block is the size of a block

Selecting values

- The size of a micro-block should normally correspond to the size of a level-2 cache, considering the number of variables accessed and the FP-element size
  - Any scratch-var values are evaluated at the micro-block level, so consider their sizes
- The block sizes can now be larger than what would fit in a level-2 cache, considering the number of cores with thread balancing
- By sizing micro-blocks as wide as blocks in one or two dimensions and/or sizing blocks as wide as Mega-blocks 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 micro-block by threads that share caches
  - We refer to this subset as a *nano-block* in YASK
- Evaluate nano-blocks *concurrently* until a <u>single time-step</u> in a mini-block is evaluated (thus, there is no *temporal* nano-block tiling)
- Since blocking already uses OpenMP threads, nano-blocking is implemented with a nested level of OpenMP threading

Using nano-block tiling in YASK

Another level of tiling hierarchy

- Micro-blocks may be divided into *nano-blocks*

- Control the spatial size of a nano-block with `-nb, -nbx, -nby,` etc.

- There is no temporal nano-blocking, so there is not an `-nt` option

Selecting values

- By default, the spatial size of a nano-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 nano-block is the size of a micro-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 nano-block is a narrow slab the width of one vector (usually in the first dim) and the size of a micro-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

  - By default, the `yask.sh` script runs one thread per core by passing `-outer_threads` to the binary, and the default number of inner threads is one (1).

  - If you want to use hyper-threads, use the `-inner_threads` option, e.g., `-inner_threads 2` will use both hyper-threads on most Intel® Xeon™ processors, assuming hyper-threading is enabled in the BIOS settings

  - You can also override the -outer_threads value to, for example, experiment with multiple cores sharing data across their L2 caches

- Try adjusting threads-per-block and/or nano-block sizes on the test stencil and check performance

Using pico-block tiling in YASK

Lowest level of tiling hierarchy

- Nano-blocks may be divided into *pico-blocks*
- Control the spatial size of a pico-block with `-pb, -pbx, -pby,` etc.
- There is no temporal pico-blocking, so there is not an `-pt` option
- Pico-blocking can be used for level-one (L1) cache-blocking on CPUs, but its use is more critical on GPUs, explained later in this tutorial

Selecting values

- By default, there is only one pico-block per nano-block
- Pico-blocking allows each nano-block, which is evaluated by an inner OpenMP thread, to have even more locality
- In some stencils, pico-blocking may be able to take advantage of L1 cache locality
- Try adjusting pico-block sizes on the test stencil and check performance

Review of the CPU tiling hierarchy

Hierarchy level	Spatial size options	Concurrent evaluation?	Temporal tiling?	Temporal size options (assuming time dim is "t")
Global domain	-g*	N/A (only one)	N/A	-trial_steps
Local domains	-l*	Yes, via MPI	No	
Mega-blocks	-Mb*	No	Yes, via sequential wave-front tiling	-Mbt
Blocks	-b*	Yes, via outer OpenMP threads	Yes, via concurrent hyper-trapezoid tiling	-bt
Micro-blocks	-mb*	No	Yes, via sequential wave-front tiling	-bt (implicitly)
Nano-blocks	-nb*	Yes, via inner OpenMP threads	No	
Pico-blocks	-pb*	No (unless on GPU)	No	
Vectors	fold=...	Yes, via HW ILP (instruction-level parallelism)	No	
Elements	real_bytes=...	Yes, via HW SIMD	No	

Example tile settings for temporal tiling

The following options were hand-tuned for an Intel® Xeon® Platinum 8352Y CPU

```
bin/yask.sh -stencil my_stencil -l 1024 \
-mbx 32 -mby 32 -mbz 128 \
-b 256 -bt 12 \
-no-pre_auto_tune
```

- Micro-block settings selected targeting 1MiB
  - The Intel® 8352Y has a 1280KiB L2 cache per core, and this gives some extra space
  - 32 × 32 × 128 × 4B × 2 = 1MiB (128 ⇒ longer in unit stride; "4B" ⇒ 4 bytes per FP element; "× 2" to keep two time-steps in cache)
- Spatial block settings selected for a multiple of number of cores per socket
  - The Intel® 8352Y has 32 cores per socket
  - Trying 2 blocks per core: 64 blocks (modify for CPU with different core count)
    - A small multiple (like 2 or 4) often works better than the exact count because several blocks per core gives more opportunity for dynamic load balancing
  - Trying 4 blocks across each dimension: 4 × 4 × 4 = 64
  - 1024 ÷ 4 = 256, so using `-b 256`
- Temporal block setting (`-bt 12`) was chosen experimentally by iterating manually through several values
- Mega-block size not set because L3 blocking is not being targeted in this experiment
- Nano-block size not set because only one thread per block is used in this experiment
- Pico-block size not set because L1 blocking is not being targeted in this experiment
- These settings resulted in about a 40% speedup over default non-temporal tiling; your results may vary

Controlling when the automatic tuner runs

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

Controlling what the automatic tuner does

Tiling levels

- By default, the auto-tuner is applied to the "block" tiles only
- More generally, you can supply a list of tiling levels via `-auto_tune_targets`
  - The argument is a comma-separated list of level abbreviations: `Mb` for mega-blocks, `b` for blocks, `mb` for micro-blocks, `nb` for nano-blocks, and `pb` for pico-blocks
  - The order specified is the order in which the tile sizes are tuned, and targets can be repeated
  - Example: `-auto_tune_targets b,Mb,b` will tune blocks, then mega-blocks, then blocks again

Other controls

- `-auto_tune_radius` controls how far the tuner will start exploring from the initial settings
- `-auto_tune_trial_secs` controls how long a new trial must be run to consider it better than the current best trial; higher values avoid measurement noise but require more time to converge

GPU OFFLOADING

GPU offload overview

Mechanism

- GPU offloading is via OpenMP *Device Directives* (see [OpenMP 5.1 spec](#))
  - Run `make offload=1` to build a kernel with GPU offloading
    - Add `offload_arch=device_name` to `make` command to change the OpenMP target
      - Default device name is `spir64` to target Intel GPUs when using the Intel compiler
      - When targeting `spir64`, specific device code is generated at run-time with JIT offload compiler
  - Run `yask.sh -offload` to select the GPU offload kernel
- The nano- and pico-block loops are offloaded by default
  - Nano loops executed via `omp target`…
    - Thread limit set via `-device_thread_limit` command-line option
  - Pico loops are nested within nano loops
    - These are the default auto-tuner targets for an offload binary (instead of blocks as for CPU binaries)
    - For a solution with N domain dimensions, N-1 dim loops are concurrent; one (1) is sequential
  - All higher-level loops are executed on the CPU
    - The default number of CPU threads for an offload binary is one (1)
    - Using more than one CPU thread allows launching more than one GPU kernel concurrently
    - The default sizes of all blocks larger than pico-blocks are the full rank size

Example offload pseudo-code for a 3D stencil when using Intel oneAPI

```
// The "nano" loop distributes the device teams across the 3D space to the "pico"
blocks.
#pragma omp target teams distribute thread_limit(thread_limit)
for (3D-index in manually-collapsed 3D loop over current nano block,
     stepping by size of a pico block) {

  // The "pico" loops evaluate points in the pico block within each device team.
  #pragma omp parallel for collapse(2)
  for (y-index over current pico block) {
    for (z-index over current pico block) {
      Set up pointers for inner loop;
      for (x-index over current pico block) {
        Evaluate stencil equation at x, y, z indices;
  } } }
}
```

Example offload using Intel oneAPI

Build for offload device

- `make -j stencil=iso3dfd offload=1`
  - Note the CPU-target.offload-target *arch* string that is printed at the end of the build
  - By default, the Intel compiler is used for a `spir64` target

Run via yask.sh

- `bin/yask.sh -stencil iso3dfd -offload -g 1024 \`
  `-no-pre_auto_tune -pb 32 -pbx 256`
- `bin/yask.sh -stencil iso3dfd -offload -g 1024`
- Use the `-device_thread_limit` option to set the number of OpenMP device threads per team
- The script will attempt to find the number of Intel GPUs on the current system and use that to determine how many MPI ranks to start; use `-ranks` to override

Example offload pseudo-code for a 3D stencil when using a non-Intel compiler

```
// The "nano" loop assigns workers across the 3D space to the "pico" blocks.
#pragma omp target teams distribute parallel for collapse(3) \
 thread_limit(thread_limit)
for (x-index over current nano block) {
  for (y-index over current nano block) {
    for (z-index over current nano block) {

      // The "pico" loops evaluate points in the pico block in each device thread.
      for (y-index over current pico block) {
        for (z-index over current pico block) {
          Set up pointers for inner loop;
          for (x-index over current pico block) {
            Evaluate stencil equation at x, y, z indices;
      } } }
} } }
```

Example offload using the Nvidia* HPC SDK

Build for offload device

- `make –j stencil=iso3dfd offload=1 mpi=0 YK_CXX=nvc++`
  - This command disables MPI and sets the C++ compiler
  - For MPI support (for more than 1 GPU), build with `mpi=1 YK_CXX=mpic++`

Run via yask.sh

- `bin/yask.sh -stencil iso3dfd –offload -g 1024 \`
  `-no-pre_auto_tune -pb 1 -pbx 128`
- `bin/yask.sh -stencil iso3dfd –offload -g 1024`
- The script will attempt to find the number of Nvidia GPUs on the current system and use that to determine how many MPI ranks to start; use `–ranks` to override

Note: it is not suggested that these build configurations and run options are optimum for any particular stencil or device

WRAP-UP

Read more about YASK features and applications

- "Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation." C Yount. *17th International Conference on High Performance Computing and Communications (HPCC),* 2015

- "YASK—Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning." C Yount, J Tobin, A Breuer, A Duran. *Domain-Specific Languages and High-Level Frameworks for High Performance,* 2016

- "Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling." C Yount, A Duran. *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance,* 2016

- "Accelerating seismic simulations using the Intel Xeon Phi knights landing processor." J Tobin, A Breuer, A Heinecke, C Yount, Y Cui. *International Supercomputing Conference,* 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*

- "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,* March, 2019

Please use one or more of the above citations in any publication submissions

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!

https://github.com/intel/yask