YET ANOTHER STENCIL KERNEL (YASK):
A FRAMEWORK FOR FINITE-DIFFERENCE OPTIMIZATION

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

with contributed material from

- Alex Duran, Intel Corporation Iberia, Spain
- Alex Breuer & Josh Tobin, Univ. of CA, San Diego
- Alex Heinecke, Intel Labs

Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
October 4, 2017

Legal Disclaimers

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

No computer system can be absolutely secure.

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

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.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Statements in this document that refer to Intel's plans and expectations for the quarter, the year, and the future, are forward-looking statements that involve a number of risks and uncertainties. A detailed discussion of the factors that could affect Intel's results and plans is included in Intel's SEC filings, including the annual report on Form 10-K.

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

Optimization Notice

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

Notice revision #20110804

Agenda

Background

- Introduction to HPC Stencil Algorithms via earthquake simulation
- Overview of the Intel® Xeon Phi™ and Intel® Xeon® Scalable Processors

YASK Framework for creating and tuning Stencil Code

- Vector-folding feature
- Automatic-tuning feature
- Current and future work

Summary

HPC STENCIL CODE INTRODUCTION

Application Domain: HPC Stencil Computation

- Iterative kernels that update elements in one or more N-dimensional grids using a fixed pattern of computation on neighboring elements
- Fundamental algorithm in many scientific simulations, commonly used for solving differential equations using finite-difference methods (FDM)

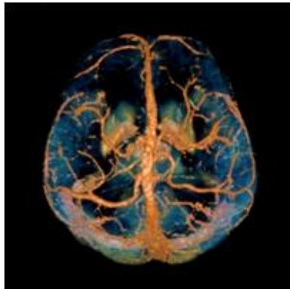
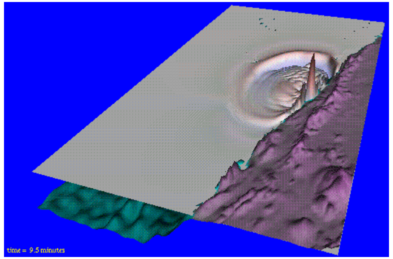
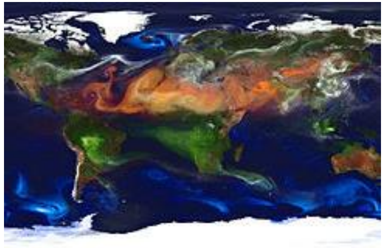


Image Processing



Seismic Modeling



Weather Simulation

Images from https://commons.wikimedia.org

Content on this slide courtesy of UCSD

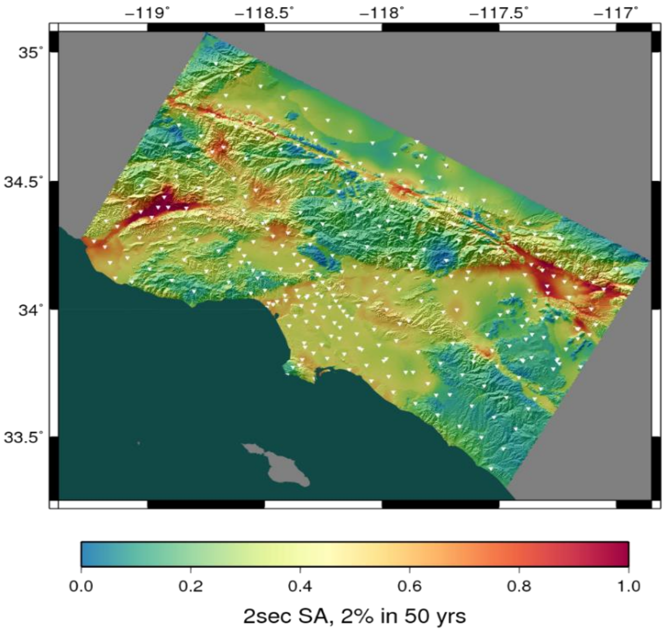
Example: AWP-ODC-OS

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
- In recent years has primarily run on GPU accelerated supercomputers

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)



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

AWP-ODC-OS Development Process

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \qquad \sigma = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{pmatrix}$$

$$\partial_t v = \frac{1}{\rho} \nabla \cdot \sigma$$
$$\partial_t \sigma = \lambda(\nabla \cdot v) I + \mu(\nabla v + \nabla v^T)$$

1. Geophysicists use differential equations to represent velocity and stress of rock and soil during an earthquake

$$\partial_x \Phi_{i,j,k} \approx \frac{c_1 \left(\Phi_{i+\frac{1}{2},\, j,k} - \Phi_{i-\frac{1}{2},\, j,k}\right) + c_2 \left(\Phi_{i+\frac{3}{2},\, j,k} - \Phi_{i-\frac{3}{2},\, j,k}\right)}{h}$$

$$\partial_t v(t) \approx \frac{v(t + \frac{\Delta t}{2}) - v(t - \frac{\Delta t}{2})}{\Delta t}$$
$$\partial_t \sigma \left(t + \frac{\Delta t}{2}\right) \approx \frac{\sigma(t + \Delta t) - \sigma(t)}{\Delta t}$$

2. Derivatives are approximated in both time and space (only *x* dimension shown)

$$\underset{(t+1)}{\sigma_{xx}[i,j,k]} = \sigma_{xx}[i,j,k] + \frac{\Delta t}{h}\left[(2\mu + \lambda)D_x + \lambda D_y + \lambda D_z\right]$$

$$\lambda = 8(\lambda[i,j,k] + \lambda[i+1,j,k] + \\ \lambda[i,j-1,k] + \lambda[i+1,j-1,k] + \ldots$$

$$\mu = 8(\mu[i,j,k] + \mu[i+1,j,k] + \\ \mu[i,j-1,k] + \mu[i+1,j-1,k] + \\ \mu[i,j,k-1] + \mu[i+1,j,k-1] + \\ \mu[i,j-1,k-1] + \mu[i+1,j-1,k-1])^{-1}$$

$$D_x = c_1(v_x[i+1,j,k] - v_x[i,j,k]) + \\ c_2(v_x[i+2,j,k] - v_x[i-1,j,k])$$
$$D_y = c_1(v_y[i,j,k] - v_y[i,j-1,k]) + \\ c_2(v_y[i,j+1,k] - v_y[i,j-2,k])$$
$$D_z = c_1(v_z[i,j,k] - v_z[i,j,k-1]) + \\ c_2(v_z[i,j,k+1] - v_z[i,j,k-2])$$

3. Equations are expanded to finite-difference stencils (this is one of 15 stencils for AWP-ODC staggered-grid formulation)

4. Stencils are coded and tuned for HPC clusters (our focus)

LATEST INTEL® PROCESSORS FOR HPC

Intel® Xeon Phi™ Processors and Intel® Xeon® Scalable Processors

Intel® Xeon Phi™ Product Family x200

(previously code-named Knights Landing, “KNL”)

Host Processor in Groveport Platform

Self-boot Intel® Xeon Phi™ processor

KNL Architecture Overview

Up to 72 cores with 4 hyper-threads each

ISA

Intel® Xeon® Processor Binary-Compatible (w/Broadwell)

On-package memory

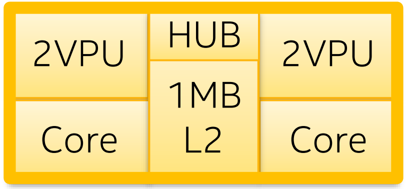
16GiB MCDRAM, ~490 GB/s Stream Triad

Platform Memory

Up to 384GiB DDR4-2400, ~90 GB/s Stream Triad

- ✓ 2D Mesh Architecture
- ✓ Out-of-Order Cores
- ✓ 3X single-thread vs. KNC

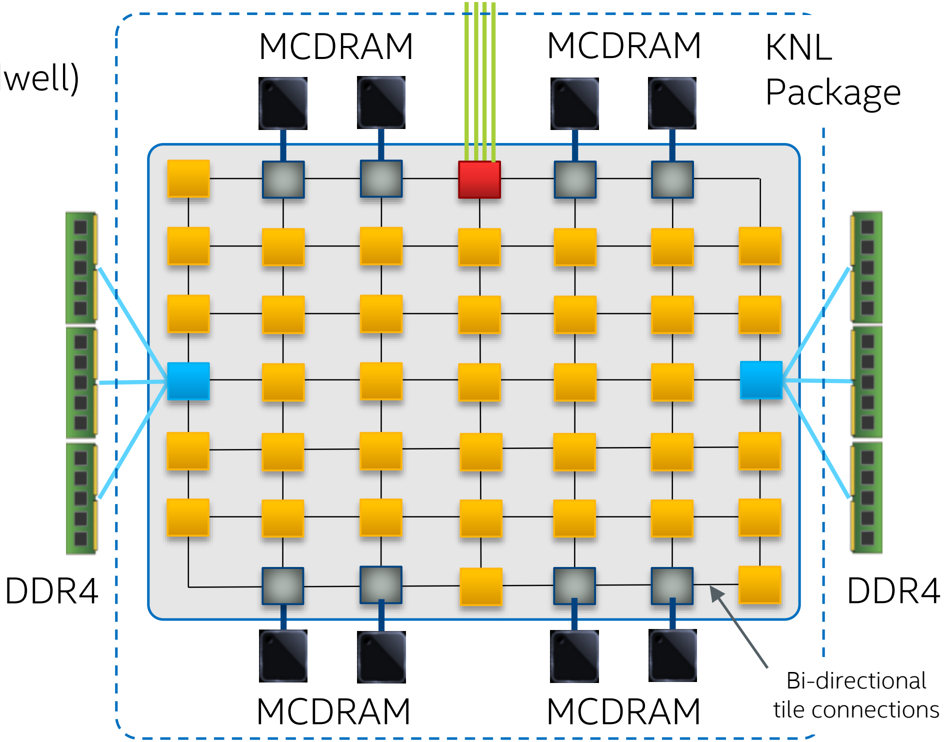
TILE: (up to 36)



2VPU | HUB | 2VPU

Core | 1MB L2 | Core

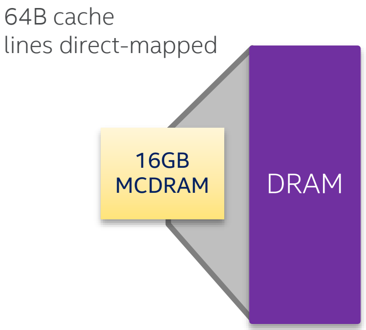
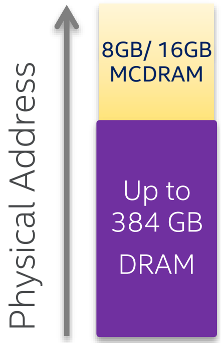
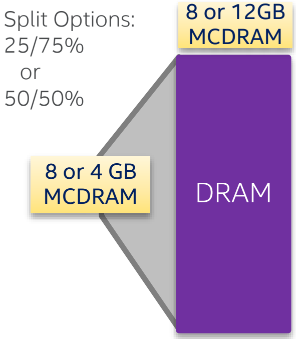
Enhanced Intel® Atom™ cores based on Silvermont Microarchitecture



🟧 Tile ▨ EDC (embedded DRAM controller) 🟦 IMC (integrated memory controller) 🟥 IIO (integrated I/O controller)

Integrated On-Package Memory Usage Models

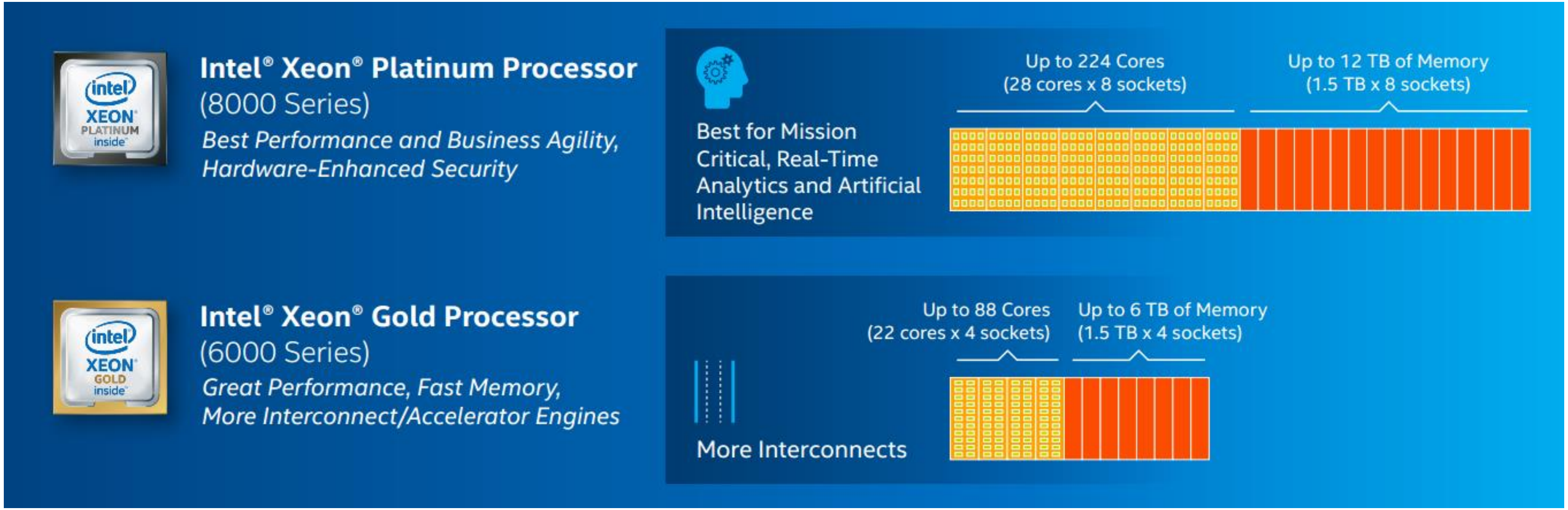
Model configurable at boot time and software exposed through NUMA[1]

	Cache Model Ideal for large data size (>16GB) cache blocking apps	**Flat Model** Maximum bandwidth for data reuse aware apps	**Hybrid Model** Maximum flexibility for varied workloads
			 Split Options: 25/75% or 50/50% 8 or 12GB MCDRAM 8 or 4 GB MCDRAM DRAM
Description	Hardware automatically manages the MCDRAM as a "L3 cache" between CPU and DDR memory	Manually manage how the app uses the integrated on-package memory and DDR for peak perf	Harness the benefits of both Cache and Flat models by segmenting the integrated on-package memory
Usage Model	▪ App and/or data set is very large and will not fit into MCDRAM ▪ Unknown or unstructured memory access behavior	▪ App or portion of an app or data set that can be "locked" into MCDRAM so it doesn't get flushed out	▪ Need to "lock" in a relatively small portion of an app or data set via the Flat model ▪ Remaining MCDRAM is configured as Cache

[1] NUMA = non-uniform memory access

Intel® Xeon® Scalable Processors

(previously code-named Skylake Xeon, “SKX”)



Intel® Xeon® Platinum Processor
(8000 Series)
Best Performance and Business Agility, Hardware-Enhanced Security

Best for Mission Critical, Real-Time Analytics and Artificial Intelligence

Up to 224 Cores (28 cores x 8 sockets)

Up to 12 TB of Memory (1.5 TB x 8 sockets)

Intel® Xeon® Gold Processor
(6000 Series)
Great Performance, Fast Memory, More Interconnect/Accelerator Engines

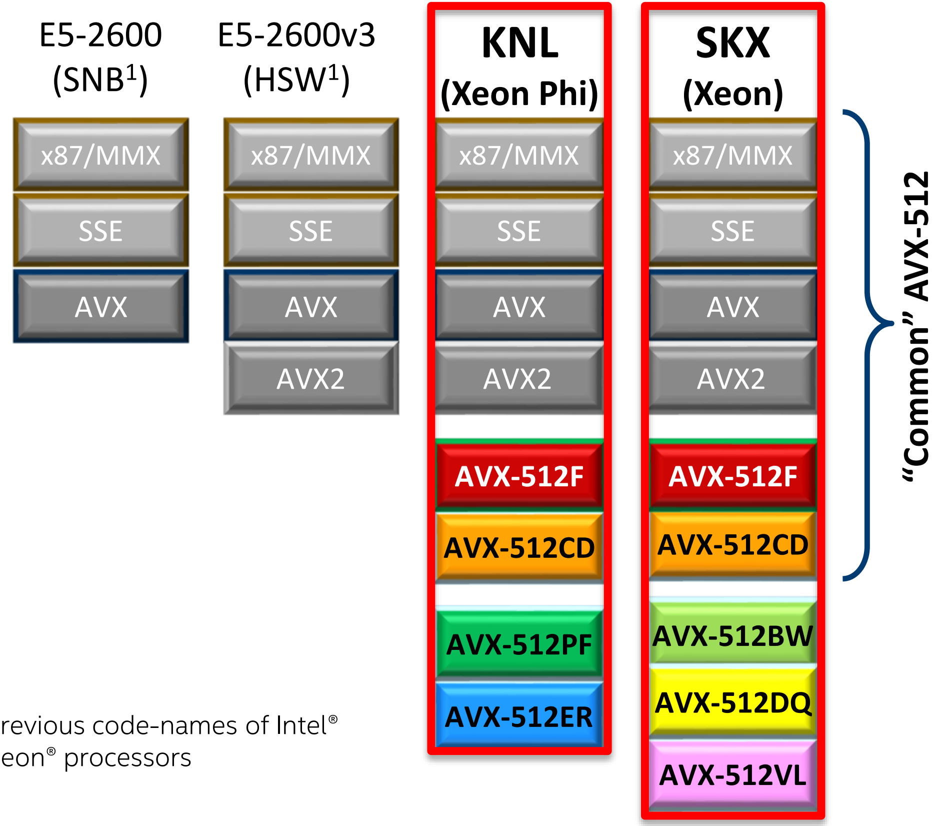
More Interconnects

Up to 88 Cores (22 cores x 4 sockets)

Up to 6 TB of Memory (1.5 TB x 4 sockets)

- Above graphic shows maximum sockets. Two-socket platforms are common in HPC installations.
- Also available: Gold (5000 Series), Silver (4000 Series), and Bronze (3000 Series)

SIMD Instruction Sets



E5-2600 (SNB[1])	E5-2600v3 (HSW[1])	KNL (Xeon Phi)	SKX (Xeon)
x87/MMX	x87/MMX	x87/MMX	x87/MMX
SSE	SSE	SSE	SSE
AVX	AVX	AVX	AVX
	AVX2	AVX2	AVX2
		AVX-512F	AVX-512F
		AVX-512CD	AVX-512CD
		AVX-512PF	AVX-512BW
		AVX-512ER	AVX-512DQ
			AVX-512VL

AVX-512:

Foundation

- 512-bit FP/Integer Vectors
- 32 SIMD registers
- 8 mask registers
- Vector gather/scatter

Conflict Detection for vectorizing histogram-type algorithms

PreFetch gather/scatter

Exponential and Reciprocal instructions

Byte and Word integer SIMD elements

Double- and Quad-word int SIMD

Vector-Length orthogonality (128 and 256-bit operations)

1. Previous code-names of Intel® Xeon® processors

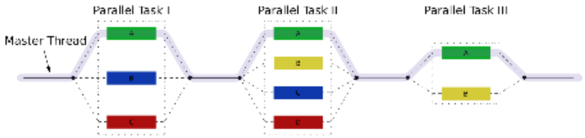
Intel® Xeon Phi™ & Xeon® Scalable on Top500

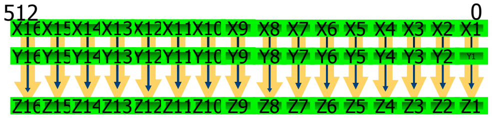
(Those in top 20 from June 2017 list, https://www.top500.org)

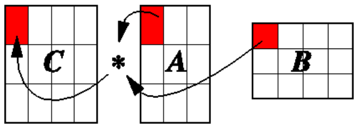
Rank	System	Cores	Rmax (TFlop/s)
6	Cori, Intel Xeon Phi 7250 68C 1.4GHz, DOE/SC/LBNL/NERSC, United States	622,336	14,014.7
7	Oakforest-PACS, Xeon Phi 7250, Joint Center for Advanced High Performance Computing, Japan	556,104	13,554.6
12	Stampede2, Xeon Phi 7250, Texas Advanced Computing Center/Univ. of Texas, United States	285,600	6,807.1
13	MareNostrum, Xeon Platinum 8160, Barcelona Supercomputing Center, Spain	148,176	6,227.2
14	Marconi - CINECA Cluster, Xeon Phi 7250, CINECA, Italy	241,808	6,223.0
16	Theta, Xeon Phi 7230, DOE/SC/Argonne National Laboratory, United States	231,424	5,884.6

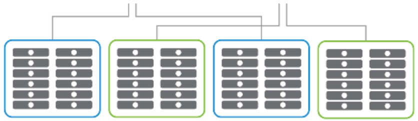
STENCIL-CODE MODERNIZATION FOR THE INTEL® XEON PHI™ PROCESSOR

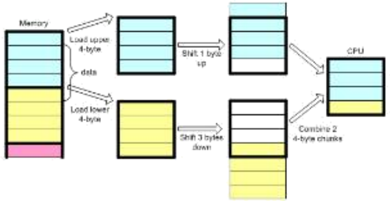
What is “Modernized” Code? (generic HPC advice)




512
0
X16 X15 X14 X13 X12 X11 X10 X9 X8 X7 X6 X5 X4 X3 X2 X1
Y16 Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1
Z16 Z15 Z14 Z13 Z12 Z11 Z10 Z9 Z8 Z7 Z6 Z5 Z4 Z3 Z2 Z1







	What	Defined	Tools of the trade
1	**Thread Scaling**	Increase concurrent thread use across coherent shared memory	OpenMP, TBB, Cilk Plus
2	**Vector Scaling**	Use wide-vector (512-bit) instructions	Vector loops, vector functions, array notations
3	**Cache Blocking**	Use algorithms to reduce memory bandwidth pressure and improve cache hit rate	Blocking algorithms
4	**Fabric Scaling**	Tune workload to increased node count	MPI
5	**Data Layout**	Optimize data layout for unconstrained performance	AoS→SoA, directives for alignment

Modernizing Stencil Code

Category	Example techniques for stencil code
1 **Thread Scaling**	• Typical: Evaluate multiple blocks in parallel using hyper-threading and multi-core • Advanced: Use nested parallelism to increase cooperation between hyper-threads and/or KNL cores sharing a tile
2 **Vector Scaling**	• Typical: Use wide-vector (512-bit) instructions • Advanced: Use KNL reciprocal instructions to improve division performance when allowable
3 **Cache Blocking**	• Typical: Use one level of blocking within a time-step to increase L2 reuse • Advanced: Use additional level of blocking with temporal wave-fronts to utilize KNL's MCDRAM cache
4 **Fabric Scaling**	• Typical: Use MPI to exchange halos between time-steps • Advanced: Schedule MPI communication to occur simultaneously with calculations of internal points
5 **Data Layout**	• Typical: Align accesses on cache-line boundaries and use KNL's MCDRAM when possible • Advanced: Use custom layout to enable vector-folding, which reduces memory bandwidth demand (details following)

Challenges

- Implementing the optimizations can be complex and error-prone
- Optimal tuning requires trading off multiple (sometimes conflicting) optimizations, each with multiple parameters
- Domain experts may reject code that obfuscates the underlying math!

Y.A.S.K. – Yet Another Stencil Kernel

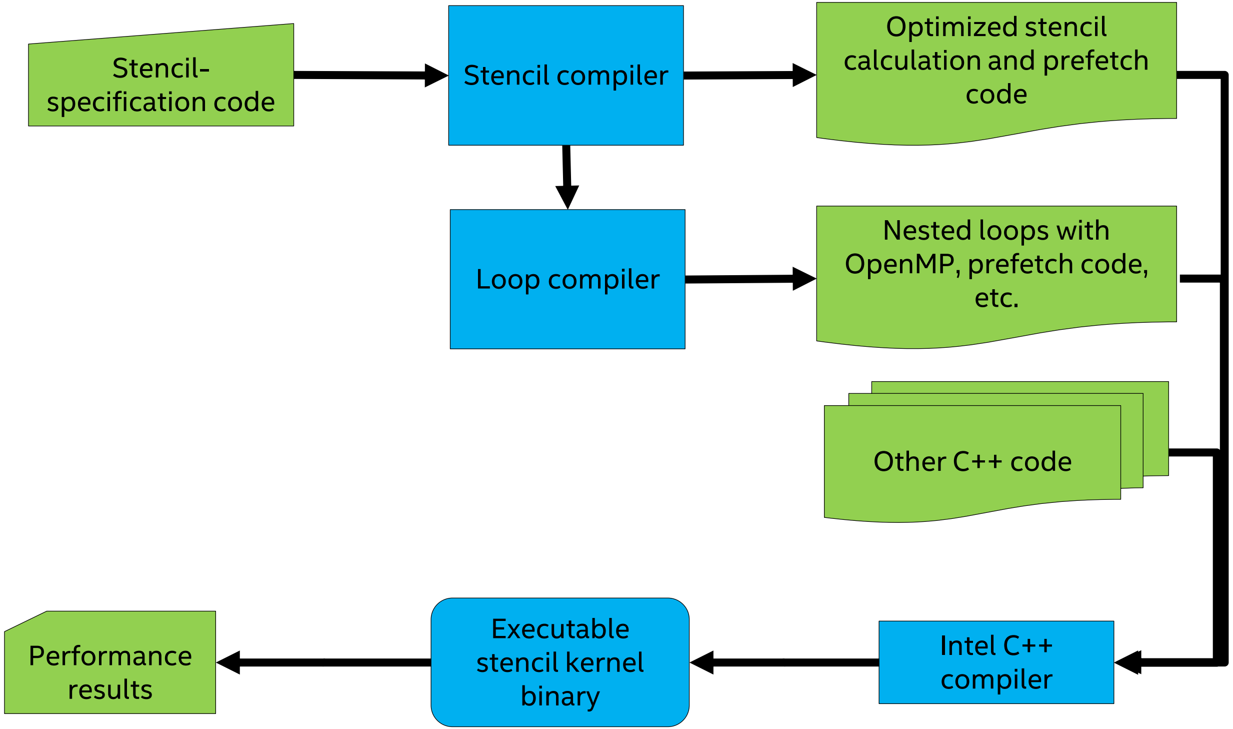
What it is [and isn't]

- A *software framework* to implement and tune stencil code for Intel® Xeon® processors and Intel® Xeon Phi™ processors and coprocessors
- Not [just] a library because stencil formulation isn't known *a priori* for all problems

Goals

- Create high-performing kernel code from a straightforward specification of stencil equations in a domain-specific language (DSL)
- Provide a simple kernel-driver to test and tune stencil performance
  - Expose optimization trade-off choices without requiring code changes
  - Automate searching through the optimization design space
- Provide ability to integrate generated code into larger applications (work in progress)

YASK High-Level Flow

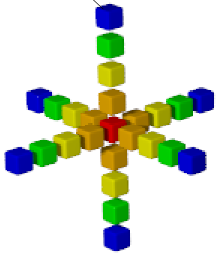

Stencil-specification code
Stencil compiler
Optimized stencil calculation and prefetch code
Loop compiler
Nested loops with OpenMP, prefetch code, etc.
Other C++ code
Intel C++ compiler
Executable stencil kernel binary
Performance results

Example YASK Feature: *Vector-Folding*

Example 3D stencil:

25 points from 3D grid u(t)

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



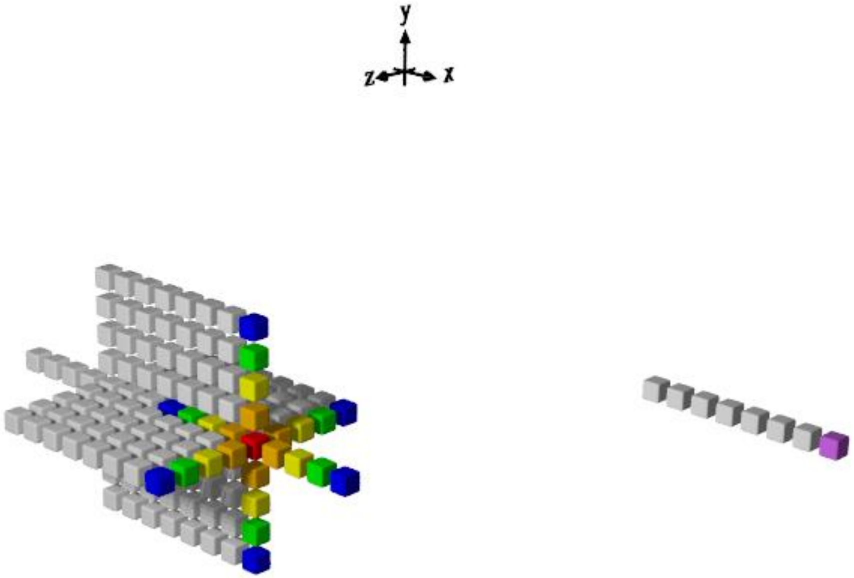
...are used to compute 1 point in u(t+1)

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

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

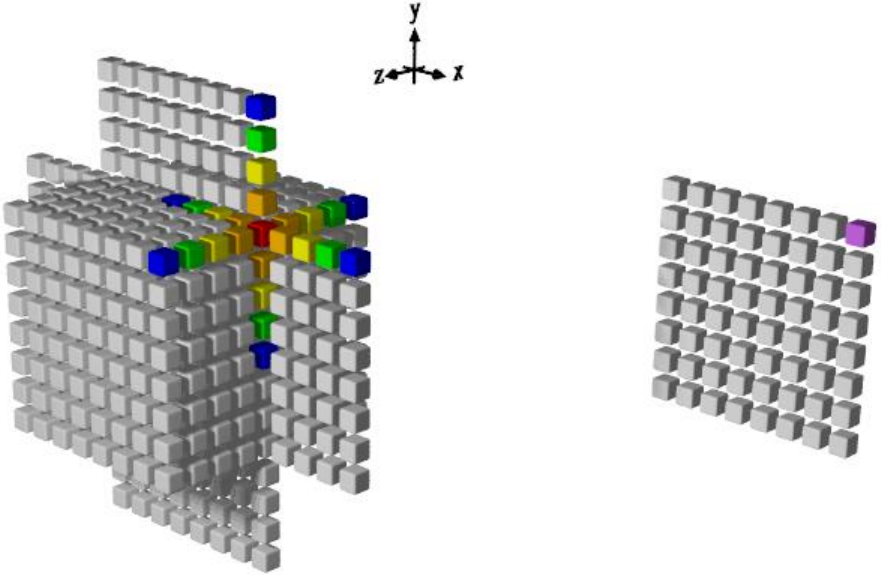
$u(t)$ → $u(t+1)$

Example 3D stencil:



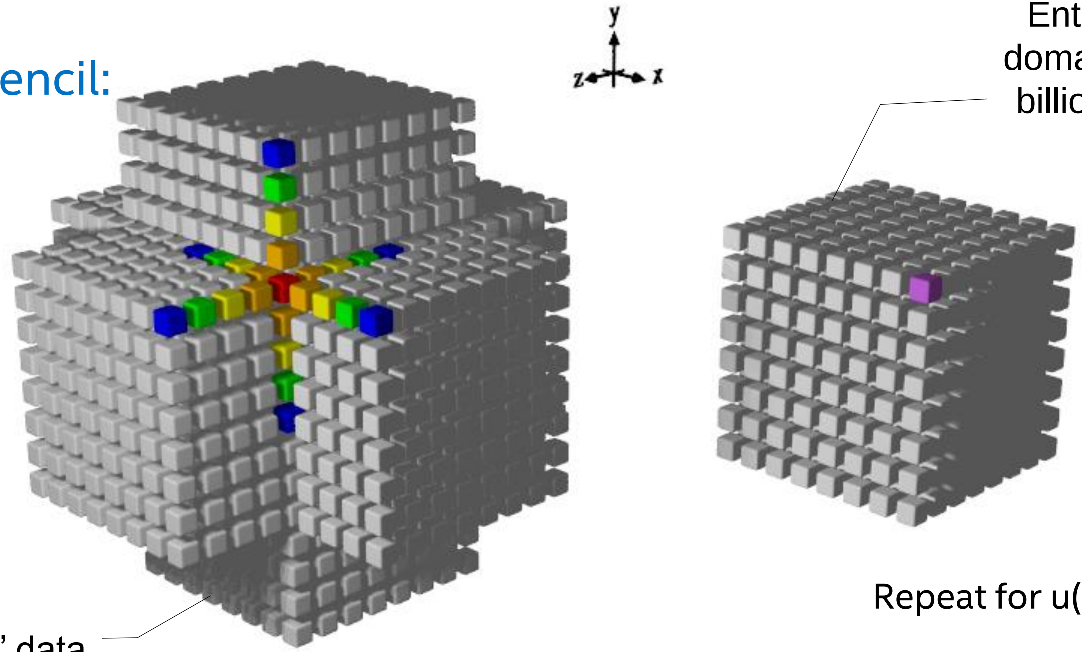
$u(t)$ → $u(t+1)$

Example 3D stencil:



$u(t)$ → $u(t+1)$

Example 3D stencil:



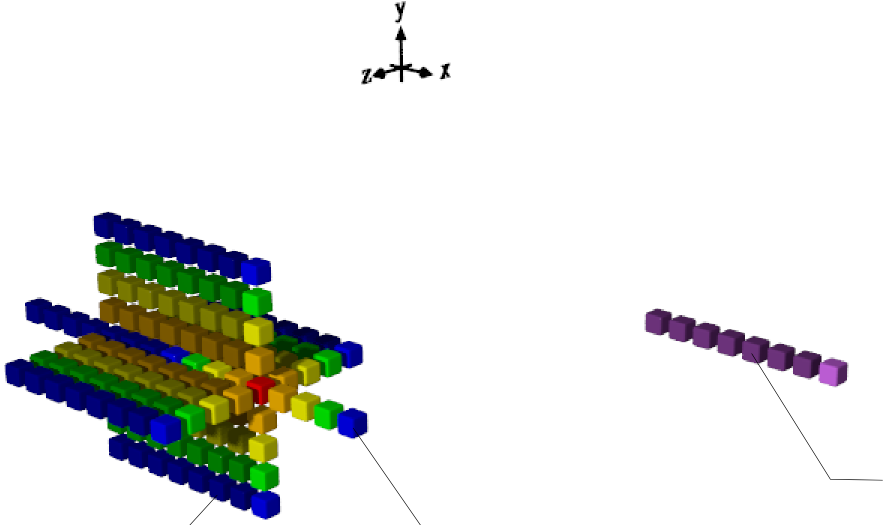
Entire problem domain—typically billions of points

Repeat for u(t+2)…

“Halo” data regions

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

Traditional 1D Vectorization



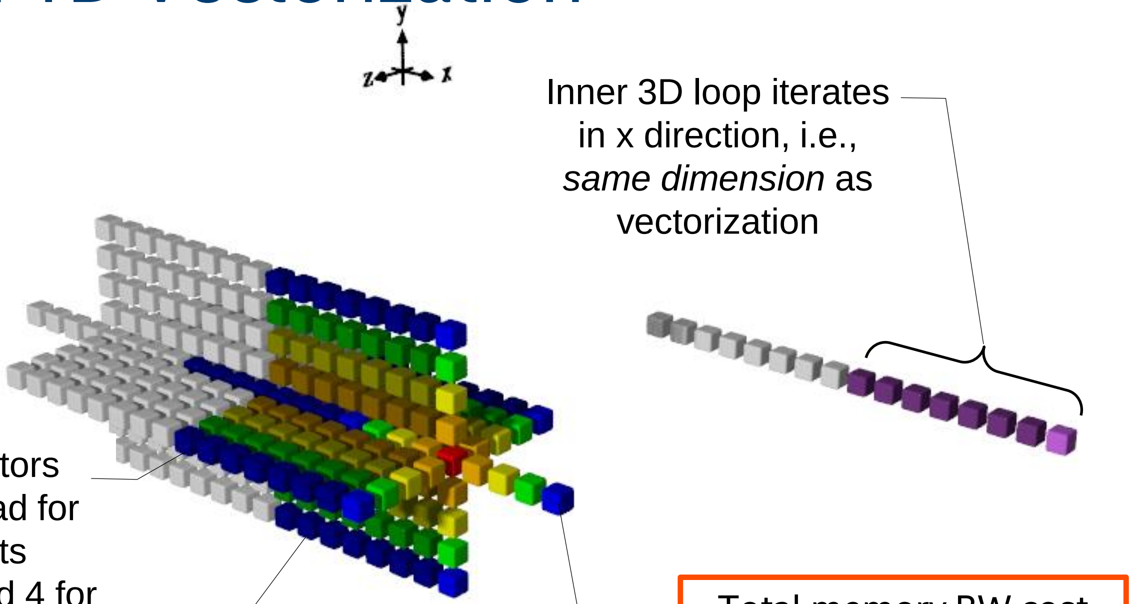
25 8-element input *vectors* (200 points) from u(t)

Notice overlap of vectors along x axis

…to compute 8 points in u(t+1)

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

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

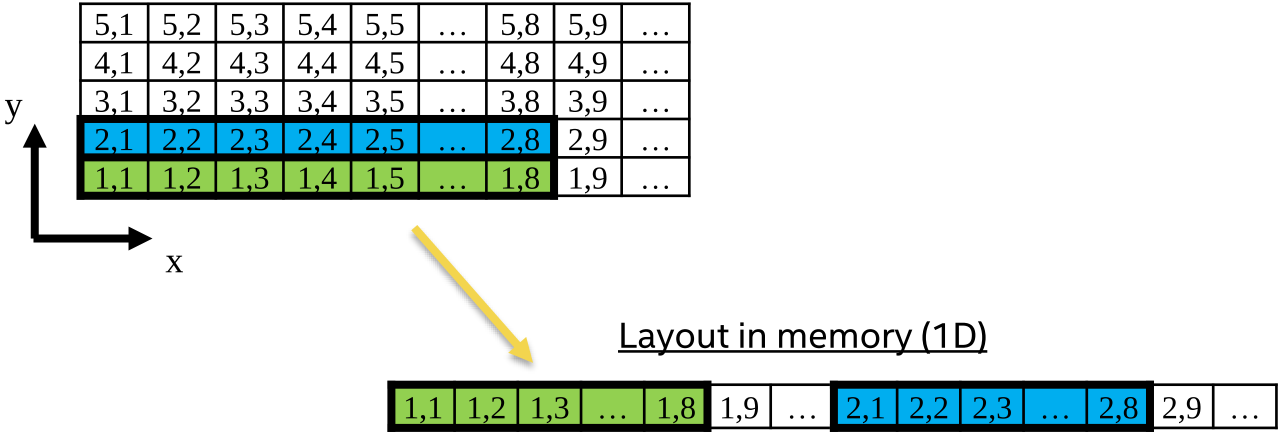
Only 1 new vector must be read for i±r points due to overlap along x axis

Total memory BW cost for traditional "in-line" vectors = **17** new vector inputs for each vector of output

Traditional Memory Layout and Code Gen

1D vectorization

Logical indices in 2D with 8-element SIMD in x-dimension



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

Layout in memory (1D)

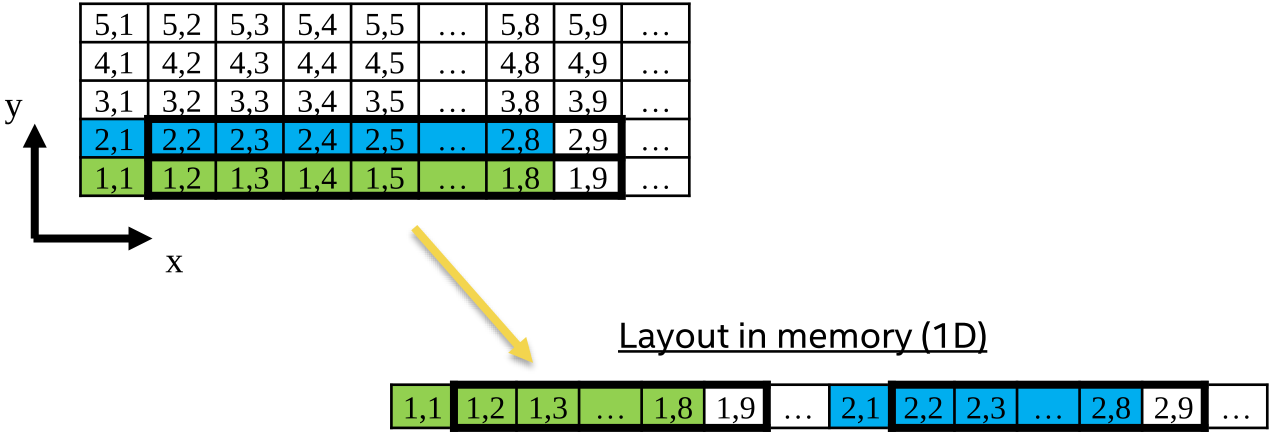
1,1	1,2	1,3	…	1,8	1,9	…	2,1	2,2	2,3	…	2,8	2,9	…

- Traditional 1D vectorization layout (8×1)
- Two aligned vectors are colored
- Aligned reads shown with bold borders done with simple and efficient aligned vector loads

Traditional Memory Layout and Code Gen

1D vectorization

Logical indices in 2D with 8-element SIMD in x-dimension



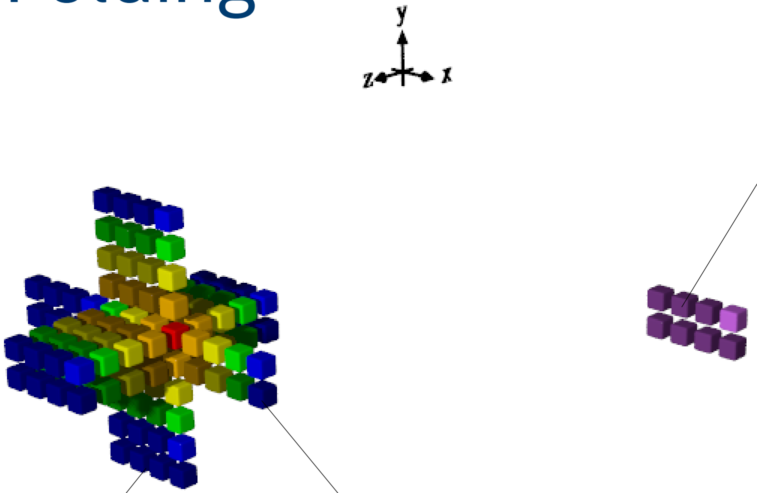
5,1	5,2	5,3	5,4	5,5	…	5,8	5,9	…
4,1	4,2	4,3	4,4	4,5	…	4,8	4,9	…
3,1	3,2	3,3	3,4	3,5	…	3,8	3,9	…
2,1	2,2	2,3	2,4	2,5	…	2,8	2,9	…
1,1	1,2	1,3	1,4	1,5	…	1,8	1,9	…

Layout in memory (1D)

1,1	1,2	1,3	…	1,8	1,9	…	2,1	2,2	2,3	…	2,8	2,9	…

- Traditional 1D vectorization layout (8×1)
- Two aligned vectors are colored
- Unaligned reads shown with bold borders done with unaligned load or two aligned loads plus a simple shift instruction

2D Vector-Folding



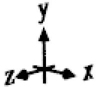
4x2x1 micro-block in
x and y dimensions

Twenty-five
4x2x1 input
vectors

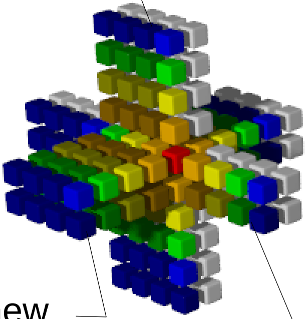
Notice overlap
of vectors
along x *and* y
axes

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

2D Vector-Folding

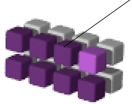


4 new 4x2x1 vectors must be read for j±r points



Only 1 new vector must be read for k±r points

2 new vectors must be read for i±r points



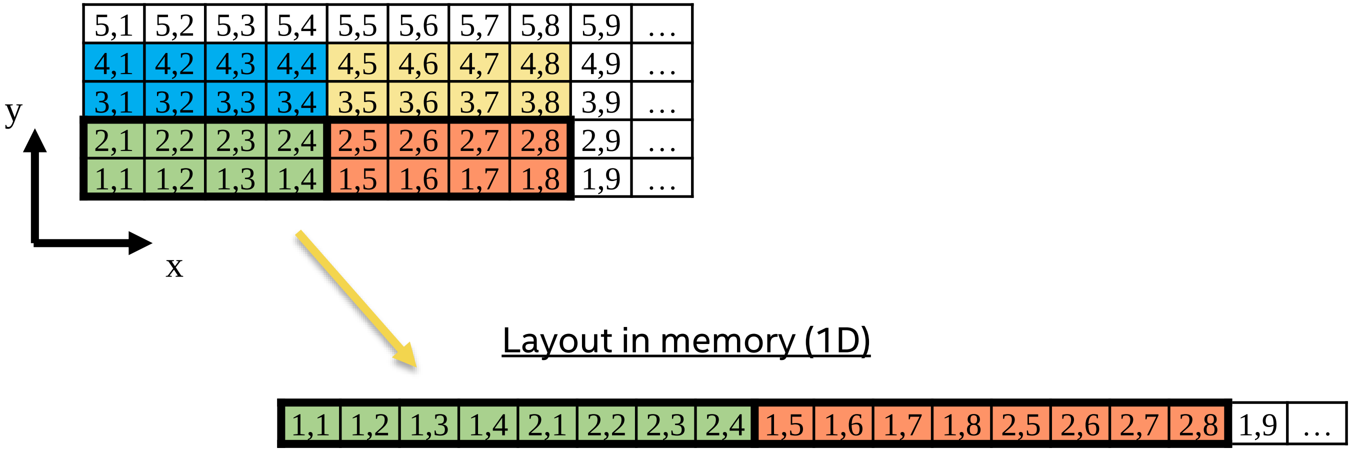
Inner 3D loop iterates in z direction, i.e., *perpendicular* to 2D vector

Total memory 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	…

Layout in memory (1D)

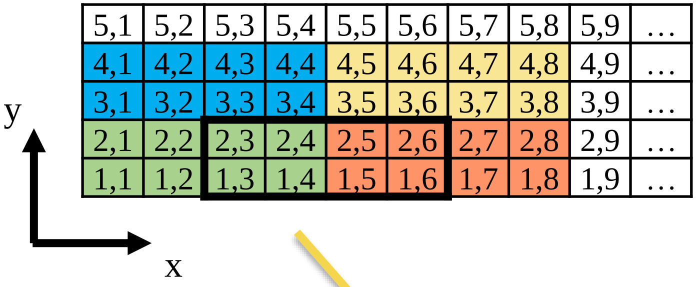
1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4	1,5	1,6	1,7	1,8	2,5	2,6	2,7	2,8	1,9	…

- 2D vector folding layout (8×1)
- Two aligned vectors are colored
- Aligned reads shown with bold borders done with simple and efficient aligned vector loads

Vector-Folding Memory Layout and Code Gen

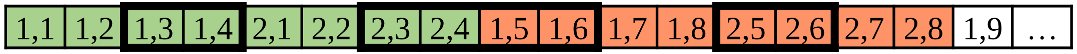
2D “4x2” vector folding

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	…

Layout in memory (1D)



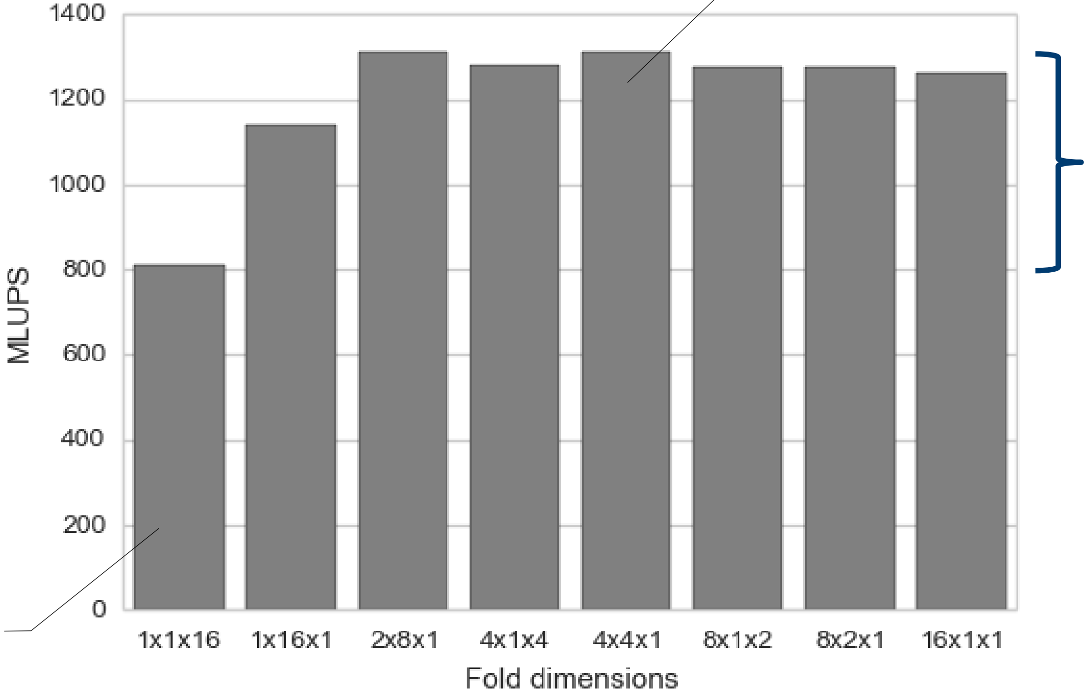
1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4	1,5	1,6	1,7	1,8	2,5	2,6	2,7	2,8	1,9	…

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

Results of Vector-Folding on AWP-ODC-OS

Content on this slide courtesy of UCSD

- Performance measured in number of Lattice Update Points per Second, updating 15 grids
- Kernel performance only (does not include surface boundary updates)
- See ISC'17 paper for more details



2D layout selected

Traditional 1D layout

Up to 1.6x performance gain

Example YASK Feature: Automatic Tuner

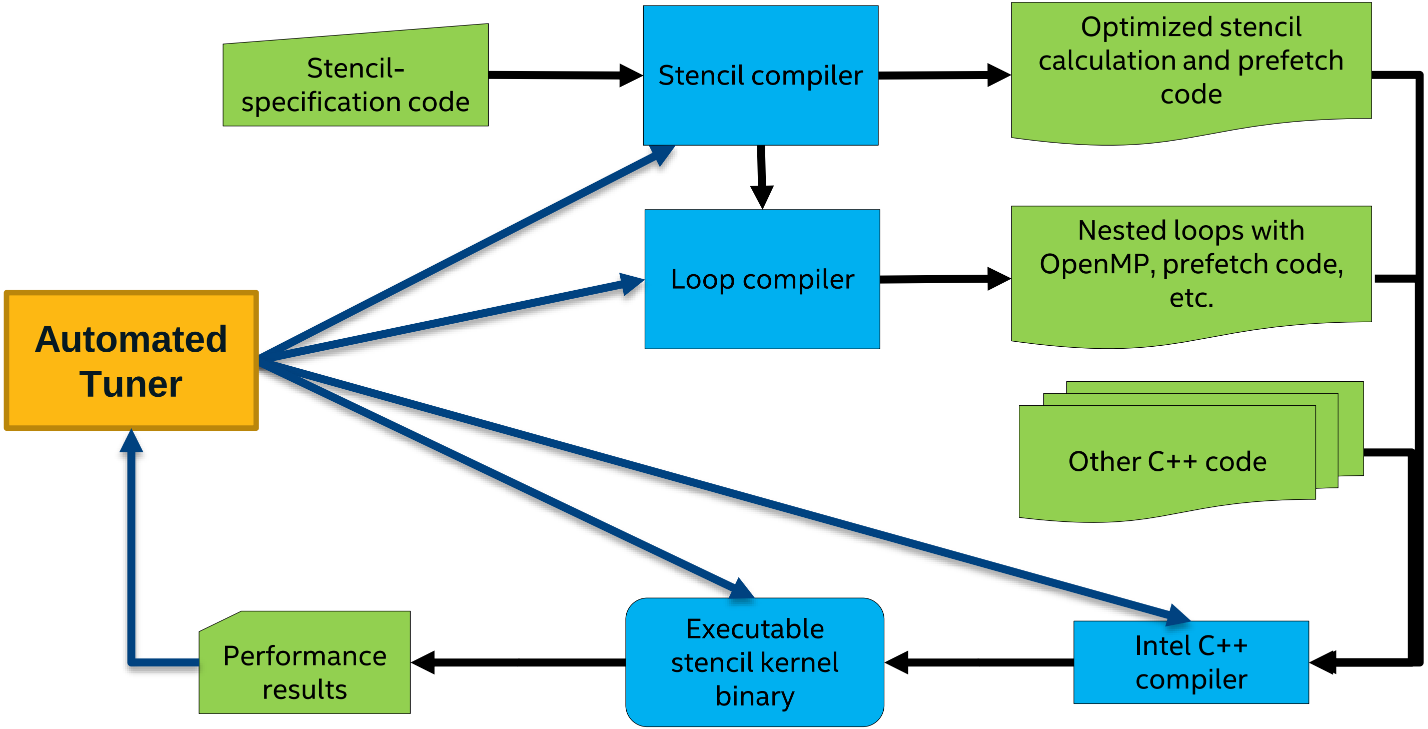
Challenge

- Dozens of possible optimization strategies
- Some of these can take hundreds of values (e.g., cache-block dimensions)
- Leads to combinatorial explosion in size of possible design space

Solution

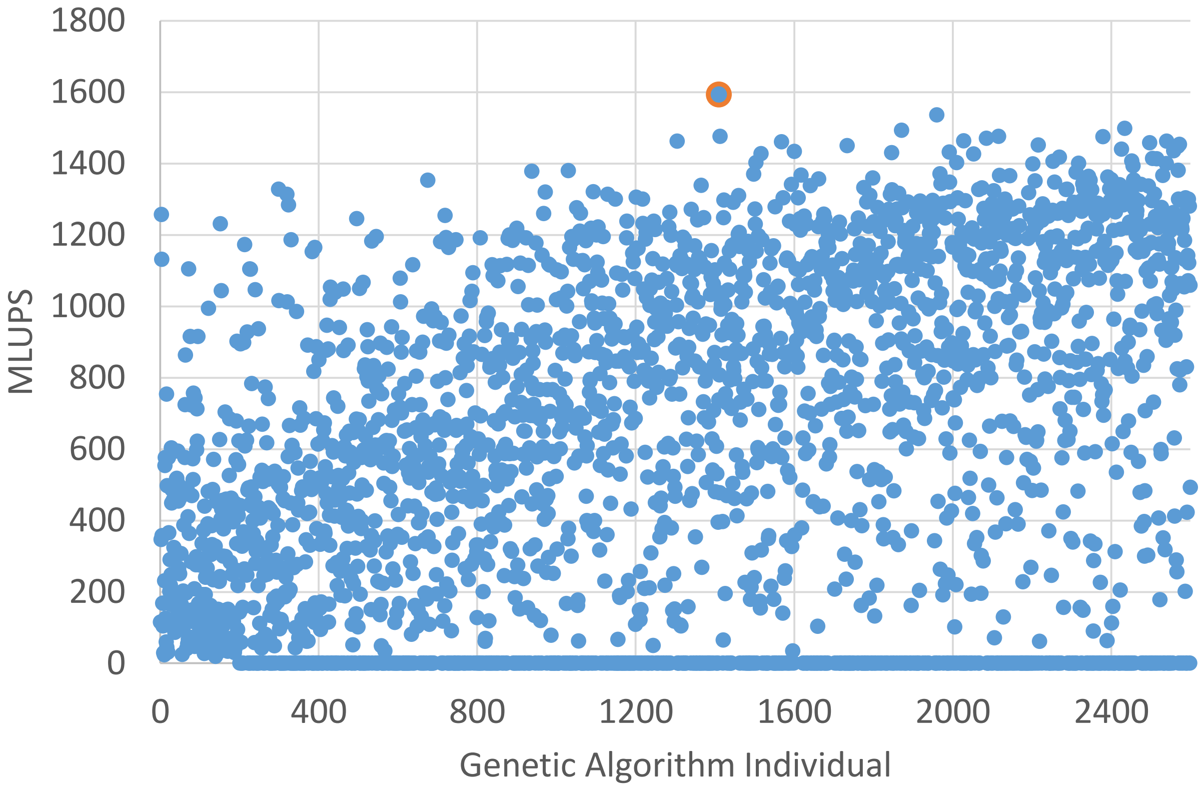
- Use genetic algorithm to select optimizations and tune parameters
- Tuner repeats the following sequence until convergence
  - Chooses optimization strategies and parameters based on random values (first generation) or mutation and crossover (subsequent generations)
  - Runs stencil compiler, loop compiler, C++ compiler, and kernel itself
  - Inputs resulting performance as fitness

YASK High-Level Flow with Tuner


Stencil-specification code
Stencil compiler
Optimized stencil calculation and prefetch code
Loop compiler
Nested loops with OpenMP, prefetch code, etc.
Other C++ code
Automated Tuner
Performance results
Executable stencil kernel binary
Intel C++ compiler

Results of Auto-Tuning on AWP-ODC-OS

- 13 generations of 200 individuals each
- Best individual (highlighted) found in 8th generation
- Performance measured in number of Lattice Update Points per Second, updating 15 grids
- Kernel performance only (does not include surface boundary updates)



Additional YASK Features

Cache-blocking with nested OpenMP

- Two levels of OpenMP parallelism applied to “block” and “sub-block” sub-domains
- Allows efficient use of level 1 and 2 local caches shared between threads and/or cores

Temporal wave-front blocking

- Allows efficient use of large shared cache (e.g., Xeon Phi MCDRAM in cache mode)

MPI halo exchange

- Allows domain decomposition across compute nodes in a cluster

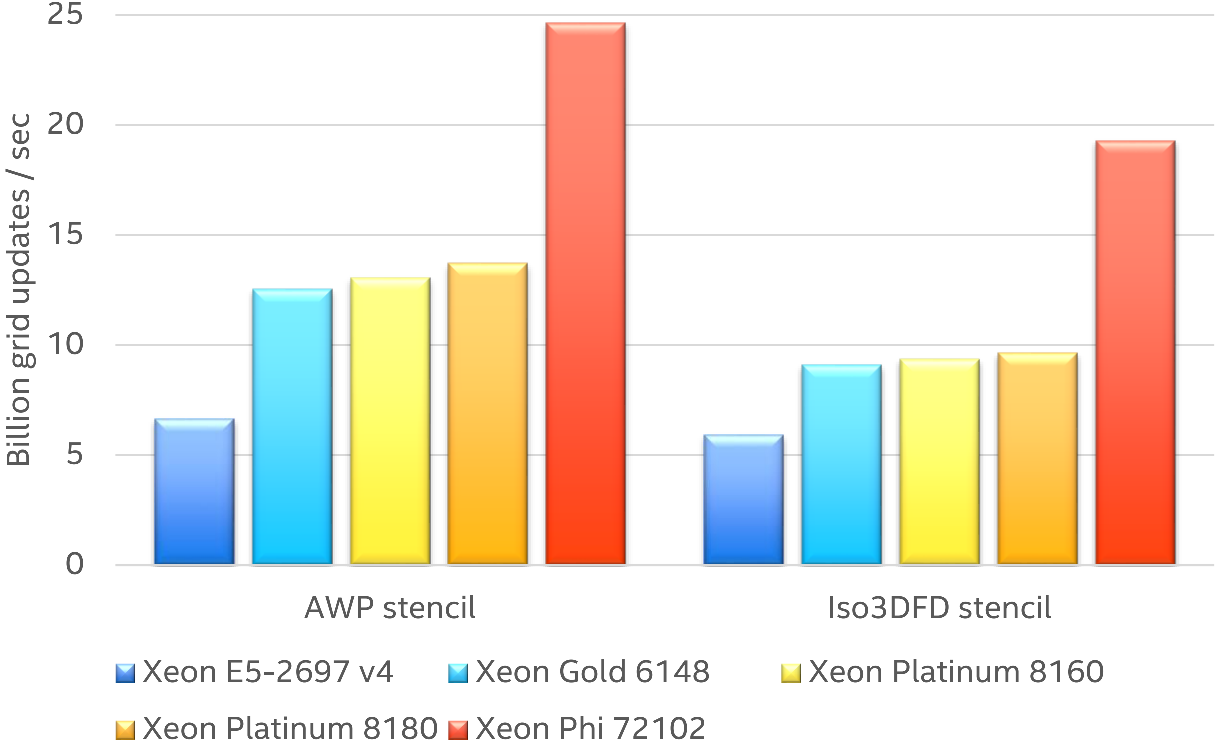
Experimental Platforms

Products	KNL (Intel® Xeon Phi™ Processor 7250)	SkyLake/Wolf Pass (Intel® Xeon® Gold 6148/ Platinum 8160/ Platinum 8180)	Broadwell (Intel® Xeon® E5-2697 v4)
Sockets -TDP	1S - 215W	2-150W/ 2-150W/ 2-205W	2-145W
Frequency – Cores/socket – Threads – L3 cache	1.4 GHz - 68 – 272 - NA	2.4 GHz - 20 – 80 – 27.5MiB/ 2.1 GHz – 24 – 96 – 33 MiB/ 2.5 GHz – 28 – 112 – 38.5 MiB	2.3 GHx – 18 – 72 – 45 MiB
DDR4	96 GiB 2400 MHz	192 GiB 2666 MHz	128 GiB 2400 MHz
MCDRAM	16 GiB (Quadrant Flat)	N/A	N/A
Turbo	On	On	On
Compiler version	Intel C++ compiler 17.0.4	Intel C++ compiler 17.0.4	Intel C++ compiler 17.0.4
OS	RHEL7.2	RHEL 7.3	RHEL7.2

YASK Performance

AWP:

- $1024^2 \times 128$ problem size
- N.B.: Results are in grid updates per sec, 15× the lattice updates in previous slide



Iso3DFD:

- 1024^3 problem size
- 16th order-accurate in space, 2nd order-accurate in time acoustic wave

Xeon Phi:

- Both problems fit within 16 GiB MCDRAM in flat mode

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 **www.intel.com/benchmarks**. Measurements created by **UC San Diego San Diego Supercomputer Center (SDSC)** as of Oct., 2016. See complete configuration details on "Configuration" slide. *Other names and brands may be claimed as the property of others

Current Work Items

Project engagements

- Imperial College London (ICL): Integrating YASK as a backend to Devito, a symbolic finite-difference solution software
- Barcelona Supercomputing Center (BSC): Applying YASK to full staggered-grid formulation for energy exploration
- Univ. of CA San Diego (UCSD): Continued support of AWP

Software development

- Creating a documented API (application-programmer interface) for both the stencil compiler and kernel [almost done]
- Generalizing the 3D grids to nD arrays to allow 1D and 2D stencils as well as mixed array sizes in stencils (e.g., for absorbing boundary-condition arrays) [almost done]
- Allowing domain subsets via conditional equations [in progress]

Development Process with Devito

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \qquad \sigma = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{pmatrix}$$

$$\partial_t v = \frac{1}{\rho}\nabla \cdot \sigma$$

$$\partial_t \sigma = \lambda(\nabla \cdot v)I + \mu(\nabla v + \nabla v^T)$$

1. Geophysicists use differential equations to represent velocity and stress of rock and soil during an earthquake

$$\frac{c_1\left(\Phi_{i+\frac{1}{2},\,j,k} - \Phi_{i-\frac{1}{2},\,j,k}\right) + c_2\left(\Phi_{i+\frac{3}{2},\,j,k} - \Phi_{i-\frac{3}{2},\,j,k}\right)}{h}$$

$$\partial_t v(t) \approx \frac{v(t+\frac{\Delta t}{2}) - v(t-\frac{\Delta t}{2})}{\Delta t}$$

$$\partial_t \sigma\left(t + \frac{\Delta t}{2}\right) \approx \frac{\sigma(t+\Delta t) - \sigma(t)}{\Delta t}$$

2. Derivativ ted in both time an dimension shown)

3. Equations are expanded to finite-difference stencils (this is one of 15 stencils for AWP-ODC staggered-grid formulation)

$$\underset{(t+1)}{\sigma_{xx}[i,j,k]} = \sigma_{xx}[i,j,k] + \frac{\Delta t}{h}\left[(2\mu + \right. \quad {}_y + \lambda D_z]$$

$$\lambda = 8(\lambda[i,j,k] + \lambda[i$$

$$\lambda[i,j - \quad ,k] + \ldots$$

$$\mu = 8($$

$$-1,k]+$$

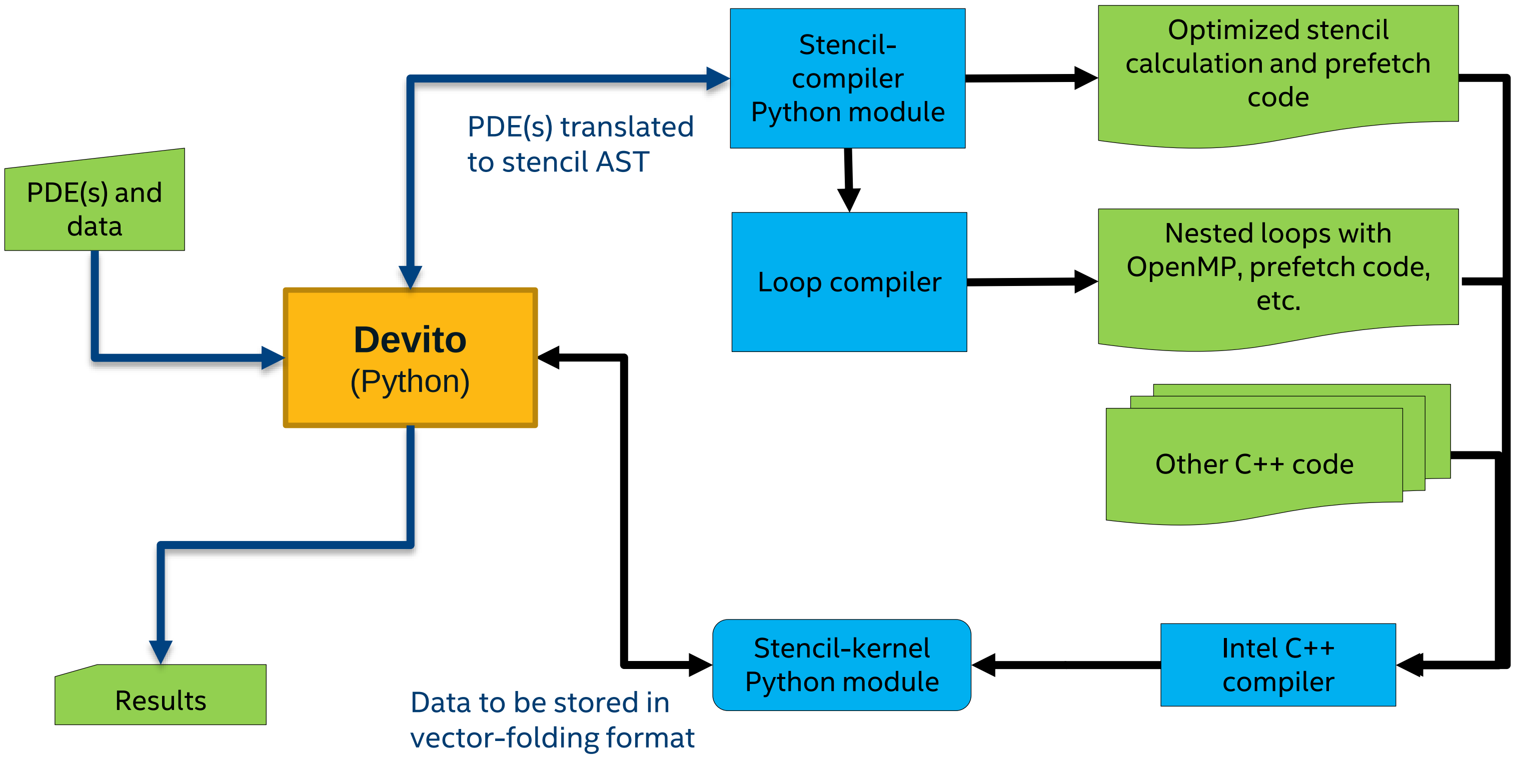
$$\mu[i+1,j,k-1]+$$

$$\mu[i \quad -1,k-1] + \mu[i+1,j-1,k-1])^{-1}$$

$$D_x = c_1(v_x[i+1,j,k] - v_x[i,j,k]) +$$
$$c_2(v_x[i+2,j,k] - v_x[i-1,j,k])$$
$$D_y = c_1(v_y[i,j,k] - v_y[i,j-1,k]) +$$
$$c_2(v_y[i,j+1,k] - v_y[i,j-2,k])$$
$$D_z = c_1(v_z[i,j,k] - v_z[i,j,k-1]) +$$
$$c_2(v_z[i,j,k+1] - v_z[i,j,k-2])$$

4. Stencils are coded and tuned for HPC clusters (our focus)

YASK High-Level Flow with Devito



PDE(s) and data
Devito
(Python)
PDE(s) translated to stencil AST
Stencil-compiler Python module
Optimized stencil calculation and prefetch code
Loop compiler
Nested loops with OpenMP, prefetch code, etc.
Other C++ code
Intel C++ compiler
Stencil-kernel Python module
Data to be stored in vector-folding format
Results

Future Work and Opportunities

Additional workloads (skills: HPC workload familiarity)

- Provide more real-world example stencils in YASK distribution
- Adapt more applications to use YASK

Add useful interfaces (skills: C++)

- Add file and/or network I/O to read/write data
- Integrate with data-visualization package

Improve cluster performance (skills: OpenMP, MPI, C++)

- Reorganize work to allow MPI communication to occur in parallel with computation
- Enable temporal wave-front computation on multiple MPI ranks

Improve open-source package (skills: SW eng.)

- Design and document C++ and Python APIs
- Write better documentation
- Add automated testing and performance-evaluation processes

Improve stencil robustness (skills: compiler internals, polyhedral model)

- Enable more complex stencil expressions to be efficiently compiled
- Evaluate expression dependencies correctly
- Stretch: replace compilation with JIT binary or LLVM IL code-generator

Resources

Software available

- YASK: https://github.com/01org/yask (MIT OS license) with several example stencils
- ICL's Devito: https://github.com/opesci/devito (MIT OS license) with tutorials and examples
- UCSD's AWP-ODC-OS: https://github.com/HPGeoC/awp-odc-os (BSD OS license)

Related collateral

- Devito project: http://www.opesci.org/devito-public
- High Performance GeoComputing Lab (AWP): http://hpgeoc.sdsc.edu
- Email chuck.yount@intel.com for copies of other conference papers and presentations

Summary

HPC Stencil Code

- Use numerical methods to approximate solutions to differential equations

Intel® Xeon Phi™ Processors (Knights Landing)

- New AVX-512 instruction set architecture
- Up to 72 cores of 4 hyper-threads each
- High-bandwidth on-package MCDRAM

Intel® Xeon® Scalable Processors (Skylake)

- Brings AVX-512 to Xeon product line
- Up to 28 cores per socket; 2 and 4-socket platforms available

Tuning Stencil Code for the Xeon Phi with YASK

- Framework for rapid prototyping and tuning
- Turnkey solution for application of multiple complex performance features

Y*A*S*K
Yet Another Stencil Kernel

BACKUP

Iso3DFD Stencil Example

$$
\begin{aligned}
p(t+1,i,j,k) \leftarrow{} & 2p(t,i,j,k) - p(t-1,i,j,k) + \\
& v(i,j,k)\Bigg(c_0 p(t,i,j,k) + \\
& \quad \sum_{r=1}^{8} c_r \Big[p(t,i-r,j,k) + p(t,i+r,j,k) + \\
& \qquad p(t,i,j-r,k) + p(t,i,j+r,k) + \\
& \qquad p(t,i,j,k-r) + p(t,i,j,k+r) \Big] \Bigg)
\end{aligned}
$$

- 51-point stencil
- 16th order accurate in space, 2nd order accurate in time
- 61 FP ops

YASK Input Specification for Iso3DFD Stencil

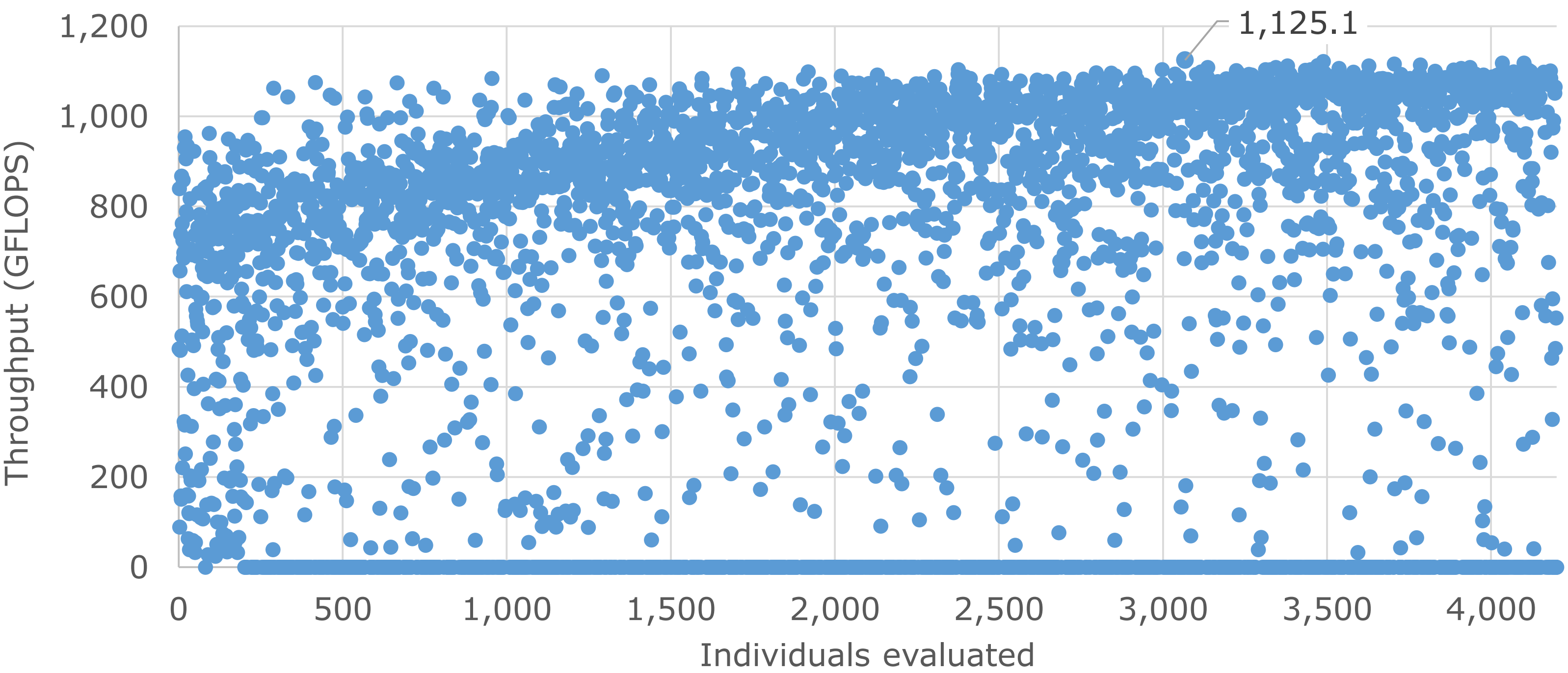
```
#include "StencilBase.hpp"
class Iso3dfdStencil : public StencilRadiusBase {
protected:
    Grid pressure;    // time-varying 3D pressure grid.
    Grid vel;         // constant 3D vel grid.
    Param coeff;      // stencil coefficients.
public:
    Iso3dfdStencil(StencilList& stencils, int radius=8) :
      StencilRadiusBase("iso3dfd", stencils, radius) {
        INIT_GRID_4D(pressure, t, x, y, z);
        INIT_GRID_3D(vel, x, y, z);
        INIT_PARAM_1D(coeff, r, radius + 1);    }
```

Declare grids and coefficient array

```
    virtual void define(const IntTuple& offsets) {
        GET_OFFSET(t); GET_OFFSET(x); GET_OFFSET(y); GET_OFFSET(z);
        GridValue np = pressure(t, x, y, z) * coeff(0);
        for (int r = 1; r <= _radius; r++) {
          np += coeff(r) *
            (pressure(t, x-r, y, z) + pressure(t, x+r, y, z) +
             pressure(t, x, y-r, z) + pressure(t, x, y+r, z) +
             pressure(t, x, y, z-r) + pressure(t, x, y, z+r));   }
        np = (2.0 * pressure(t, x, y, z))
          - pressure(t-1, x, y, z)  // subtract pressure at t-1.
          + (np * vel(x, y, z));    // add velocity term.
        pressure(t+1, x, y, z) IS_EQUIV_TO v;
    }
};
```

Define equation for pressure at t+1

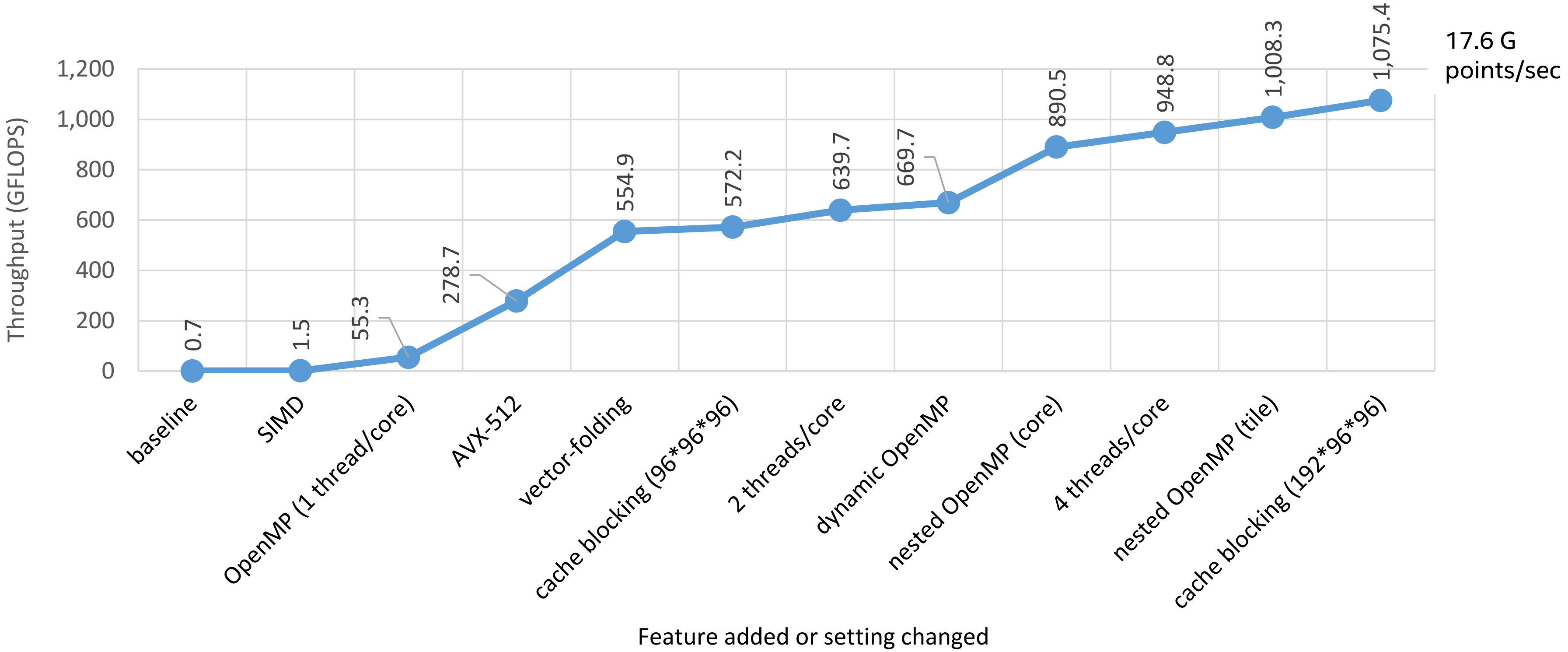
YASK Auto-tuner Applied to Iso3dfd



18.4 G points/sec

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 www.intel.com/benchmarks. **Intel measurements** as of Oct., 2016 on Intel® Xeon Phi™ processor 7250 with 16 GiB MCDRAM, 96 GiB DDR4. See complete configuration details on "Configuration" slide.

YASK Optimizations Applied to Iso3DFD



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 **www.intel.com/benchmarks**. **Intel measurements** as of Oct., 2016 on Intel® Xeon Phi™ processor 7250 with 16 GiB MCDRAM, 96 GiB DDR4. See complete configuration details on "Configuration" slide.

Content on this slide courtesy of UCSD

AWP-ODC Numerics

Finite Difference code

- Staggered-grid scheme
- Fourth-order accurate in space and second-order accurate in time

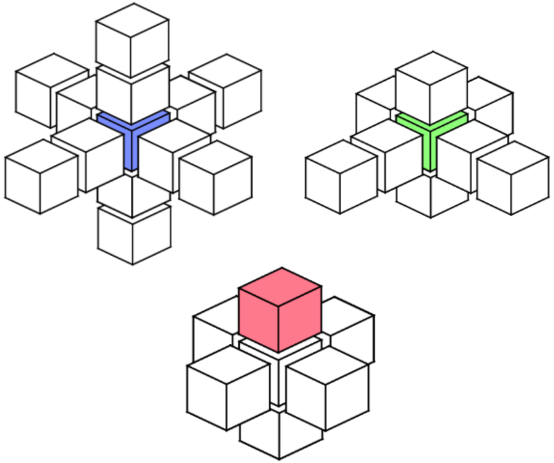
Fifteen grids updated in every time-step

- 3 velocity grids
- 6 stress grids
- 6 grids for auxiliary memory-variables required for accurate high-frequency simulation

Fifteen stencils

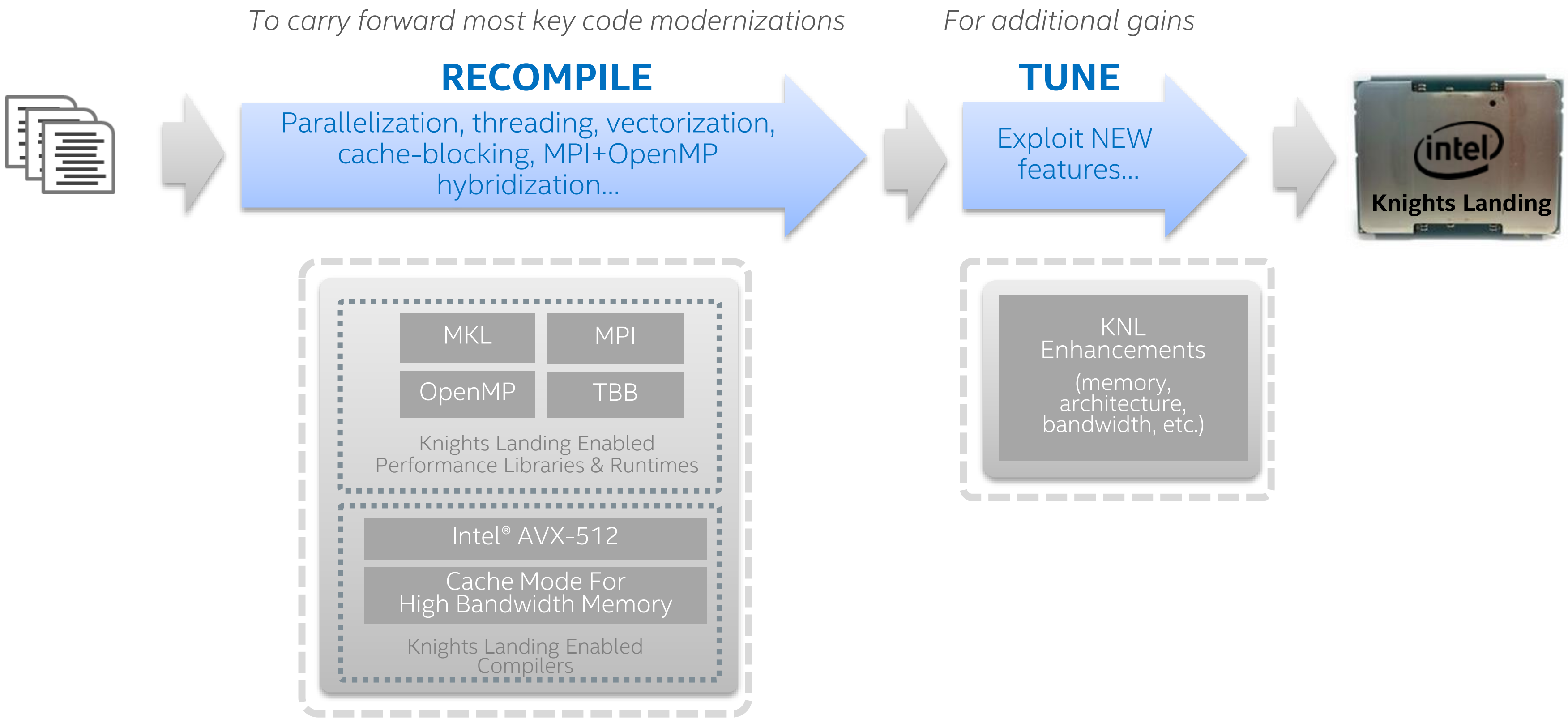
- Nine 13-point stencils
- Six 9-point stencils

Free-surface boundary computation every time-step



AWP-ODC stencils, starting from top left: velocity/stress update, memory variable stencil, boundary stencil

Today's Code Investment Carries Forward



To carry forward most key code modernizations

RECOMPILE

Parallelization, threading, vectorization, cache-blocking, MPI+OpenMP hybridization...

For additional gains

TUNE

Exploit NEW features...

MKL

MPI

OpenMP

TBB

Knights Landing Enabled Performance Libraries & Runtimes

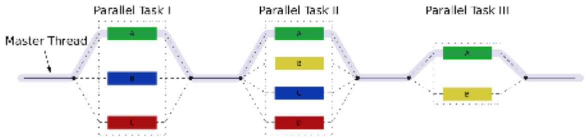
Intel® AVX-512

Cache Mode For High Bandwidth Memory

Knights Landing Enabled Compilers

KNL Enhancements (memory, architecture, bandwidth, etc.)

Thread Scaling for Stencils



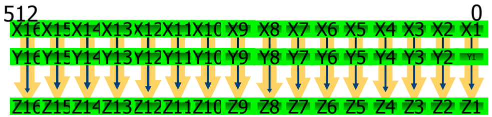
Algorithm characteristics

- Threading stencils is often straight-forward within a single grid and time-step
  - Many stencils update elements in one grid at a given time-step based only on elements in other grids or the same grid at previous time-steps
  - Some updates across multiple grids may be independent within a time-step
  - In these cases, all elements can be updated in parallel trivially
- Threading across multiple dependent grids and time-steps is more challenging
  - Requires more complex synchronization to observe dependencies

Techniques

- Example techniques to implement dependent threading include temporal wave-fronts and diamond tiling
- Threading software
  - Older code tends to use multiple single-threaded MPI tasks even on a single node
  - Often does not scale well to many threads available on KNL socket (up to 288)
  - More modern code uses OpenMP or MPI+OpenMP or MPI w/shared memory on a node
  - More advanced threading may include task scheduling to avoid global synchronization

Vector Scaling for Stencils


512
0
X16 X15 X14 X13 X12 X11 X10 X9 X8 X7 X6 X5 X4 X3 X2 X1
Y16 Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1
Z16 Z15 Z14 Z13 Z12 Z11 Z10 Z9 Z8 Z7 Z6 Z5 Z4 Z3 Z2 Z1

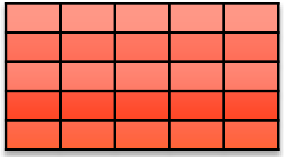
Algorithm characteristics

- The nature of “stencils” is application of a fixed pattern to multiple elements
- Elements within a grid are usually independent as discussed earlier
- Thus, SIMD vectorization can be applied in a straight-forward manner

Techniques

- Straight-forward vectorization along one dimension often results in many cache-line reads, many unused elements, and low reuse between vectors
- “Vector-folding” is a technique to vectorize in two or more dimensions, increasing reuse and thus decreasing memory-bandwidth requirements
  - Speed-ups of >1.5x have been observed in several real-world stencils
  - See HPCC’15 paper “Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation”

Cache Blocking for Stencils



Algorithm characteristics

- Stencil codes are very often memory-bound
- Stencil equations typically input multiple neighboring values (increasing with accuracy)
- These factors make cache-blocking critical for high performance

Techniques

- Most cache-blocking is implemented via simple loop-tiling with each OpenMP thread working on separate tiles
- Advanced techniques leverage sharing of KNL caches between threads
  - Each L1 data cache is shared by 4 hyper-threads in a core
  - Each L2 cache is shared by 2 cores in a tile
  - Tiles can be sub-divided into slabs or similar partitions, and threads that share these caches can cooperate within them, increasing reuse and decreasing evictions
- To leverage the MCDRAM cache shared by all cores, an addition level of tiling may be used
  - See PMBS'16 paper "Effective Use of Large High-Bandwidth Memory Caches in HPC Stencil Computation via Temporal Wave-Front Tiling"
- In addition, prefetching data into L1 and/or L2 cache may improve performance

Fabric Scaling for Stencils

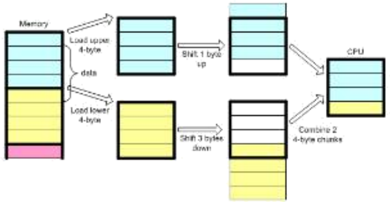
Algorithm characteristics

- As with threading and SIMD, independence of solutions within a time-step facilitate partitioning across nodes
- Access to multiple neighboring values that are common across partitions requires synchronizing data

Techniques

- Use of “halo” or “ghost” regions is the most common solution to reduce communications within a time-step as shared data is accessed
  - Halos must be exchanged between nodes to keep data consistent
  - Application of tiling across time-steps requires more sophisticated exchanges, usually consisting of exchanging more data but less often
- MPI is the most common software used, but other alternatives are in the works
- Global synchronization can cause under-utilization of nodes on large clusters and/or on clusters with nodes of heterogeneous performance

Data Layout for Stencils



Algorithm characteristics

- Many problems consist of multi-dimensional domains across multiple grids, which could be implemented naïvely with a multi-dimensional array-of-structures (AoS)
- Access to many neighboring elements and/or grids may cause translation look-aside buffer (TLB) pressure when multiple pages are accesses

Techniques

- Structure-of-arrays layout (SoA) is typical for multi-grid problems to enable vectorization
- Options to reduce TLB pressure
  - Using huge pages, e.g., via transparent huge pages (THP) in Linux
  - Reordering index nesting in grids, e.g., TXYZ → XYTZ
  - Tiling the layout itself
- To benefit from vector-folding discussed earlier, a data-layout transformation to a grid of folded vectors is essential

Experimental Configurations

Configuration details: YASK HPC Stencils, iso3DFD Kernel

Intel® Xeon Phi™ processor 7250: Intel® Xeon Phi™ processor 7250, 68 cores, 272 threads, 1400 MHz core freq. (Turbo On), MCDRAM 16 GiB, DDR4 96GiB 2400 MHz, quad cluster mode, MCDRAM flat memory mode, Red Hat* Enterprise Linux Server release 6.7

Configuration details: YASK HPC Stencils, AWP-ODC Kernel

Intel® Xeon® processor E5-2680 v3: Single Socket Intel® Xeon® processor E5-2680 v3, 2.5 GHz (Turbo Off) , 12 Cores, 12 Threads (HT off), DDR4 128GiB, CentOS* 6.7

Intel® Xeon Phi™ processor 7210: Intel® Xeon Phi™ processor 7210, 64 cores, 256 threads, 1300 MHz core freq. (Turbo On), MCDRAM 16 GiB, DDR4 96GiB 2133 MHz, quad cluster mode, MCDRAM flat memory mode, CentOS* 7.2

Intel® Xeon Phi™ processor 7250: Intel® Xeon Phi™ processor 7250, 68 cores, 272 threads, 1400 MHz core freq. (Turbo On), MCDRAM 16 GiB, DDR4 96GiB 2400 MHz, quad cluster mode, MCDRAM flat memory mode, CentOS* 7.2

NVIDIA Tesla* K20X (Kepler): Part number 900-22081-0030-000, 1x GK110 CPU, 2688 cores, 732 MHz core freq, 6GiB 2.6GHz GDDR5

NVIDIA M40 (Maxwell): Part number TCSM40M-PB, 3072 cores, 948 MHz base freq, 12 GiB GDDR5

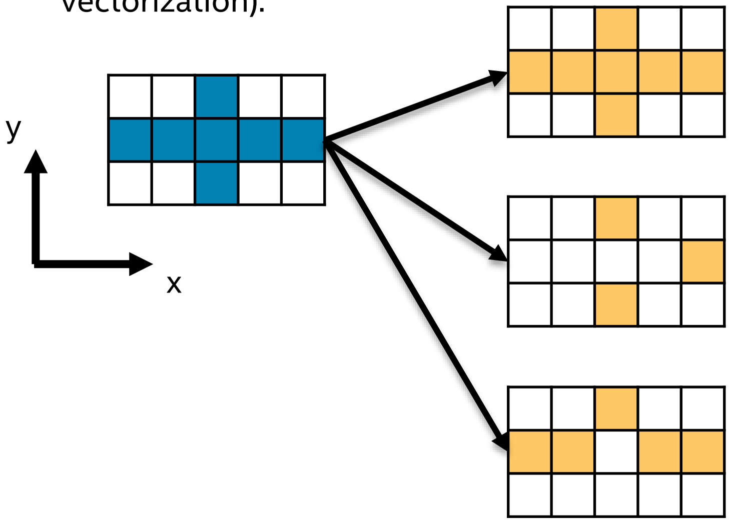
NVIDIA Titan X (Pascal): 3072 cores, 1000 MHz base freq, 12 GiB GDDR5

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

Example Stencil-Compiler Feature: Automatic Prefetch Generation

This example stencil reads from 7 cache lines (after vectorization):

The stencil compiler generates the following prefetch functions:



Full prefetch function loads all 7 cache lines

X-direction prefetch function loads only these 3 leading cache lines

Y-direction prefetch function loads only these 5 leading cache lines

Worldwide Training

Colfax training with access to a 36-node cluster

Intel® Modern Code

Intel® Modern Code

Drive faster breakthroughs through faster code: Get more results on your hardware today and carry your code forward to the future.

Intel® Parallel Computing Centers

Intel® Parallel Computing Centers (Intel® PCC)

Universities, institutions, and labs that are leaders in their field, focusing on modernizing apps.

***FREE**…Worldwide Training and Teaching Resources*

Parallel Programming Reference Books

Parallel Programming and Optimization with Intel Xeon Phi Coprocessors
Structured Parallel Programming
High Performance Parallelism Pearls
High Performance Parallelism Pearls
Intel® Xeon Phi™ Coprocessor High Performance Programming
Intel Xeon Phi Processor High Performance Programming Knights Landing Edition
MK

Commercial ISVs Embracing Intel® Xeon Phi™ processor Family

IXPUG Community Forum

THE INTEL® XEON PHI™ USERS GROUP

The Intel® Xeon Phi™ User's Group (IXPUG) is an independent global users group whose mission is to provide a forum for the free exchange of information that enhances the usability and efficiency of scientific and technical applications running on large High Performance Computing (HPC) systems using the Intel® Xeon Phi™ processor. IXPUG is administered by representatives of member sites that operate large Phi-based HPC systems.

- **IXPUG Monthly Tuning Meetings:** conference calls to inform the Intel® Xeon Phi™ processor community of relevant updates and share techniques, results, and methodologies.

Bio

Chuck received his PhD degree in ECE from Carnegie Mellon University in Pittsburgh, Pennsylvania, USA. He is currently a Principal Engineer in the Software and Services Group at Intel Corporation. His work includes developing analysis and optimization techniques for HPC applications on many-core products including the YASK open-source software framework for stencil-code optimization.

Abstract

Stencil computation is an important class of algorithms used in a large variety of scientific-simulation applications, especially those arising from finite-difference solutions of differential equations representing the behavior of physical phenomenon such as heat dispersion or seismic activity. This talk provides a brief review of stencil computation and Intel® Xeon® and Xeon Phi™ processors, and it describes the YASK (Yet Another Stencil Kernel) framework that simplifies the tasks of defining stencil functions, generating high-performance code targeted for various Intel platforms, and running tuning experiments. A couple of example YASK features are explained, performance results are given, and future work is described.