Y.A.S.K. –
YET ANOTHER STENCIL KERNEL: A FRAMEWORK FOR HPC STENCIL CODE-GENERATION AND TUNING

Chuck Yount, Intel Corporation
Josh Tobin, Univ. of CA, San Diego
Alexander Breuer, Univ. of CA, San Diego
Alex Duran, Intel Corporation Iberia, Spain

SC16 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC16), Nov. 13, 2016

Outline

Background

- Example 3D stencil
- Performance considerations
- Optimization techniques and challenges

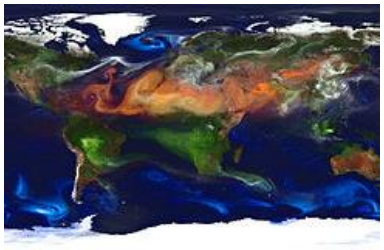
YASK

- Goal
- Stencil and loop compilers
- Automatic tuner

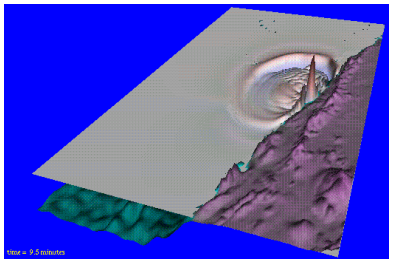
Summary

Stencil Computation

- Iterative kernels that update elements in a 1D, 2D, or 3D grid using a fixed pattern of neighboring elements
- Fundamental algorithm in many HPC algorithms and scientific simulations, e.g., finite-difference methods



Weather Simulation



Seismic Modeling

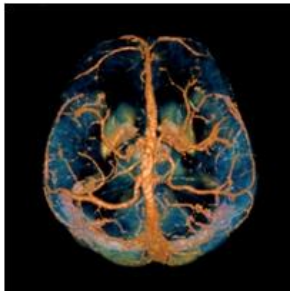
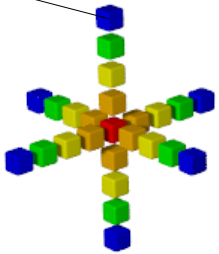


Image Processing

Images from https://commons.wikimedia.org

Example 1: 25-point 3D stencil

25 points from 3D grid u(t)



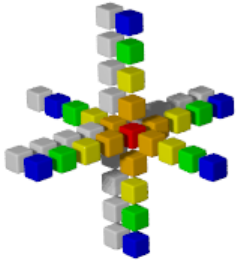
...are input to this formula

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

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

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

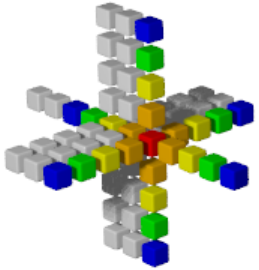
Example 3D stencil



Inner 3D loop iterates in x direction

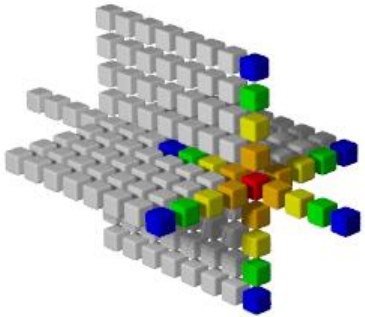
$u(t) \quad \rightarrow \quad u(t+1)$

Example 3D stencil



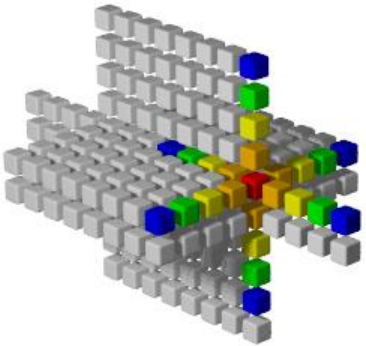
$u(t) \quad \rightarrow \quad u(t+1)$

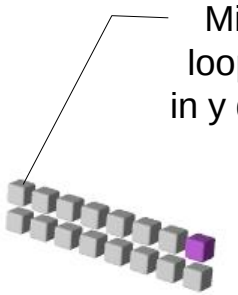
Example 3D stencil



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

Example 3D stencil

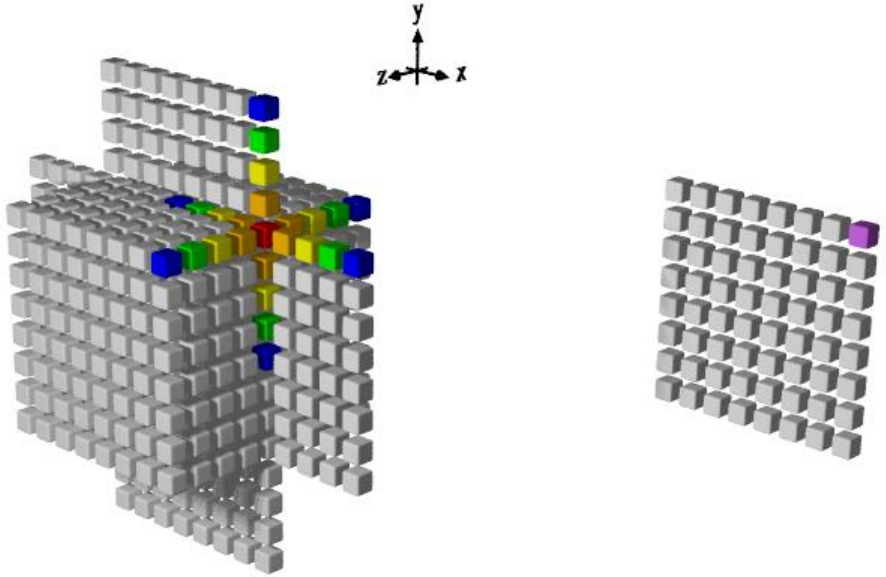




Middle 3D loop iterates in y dimension

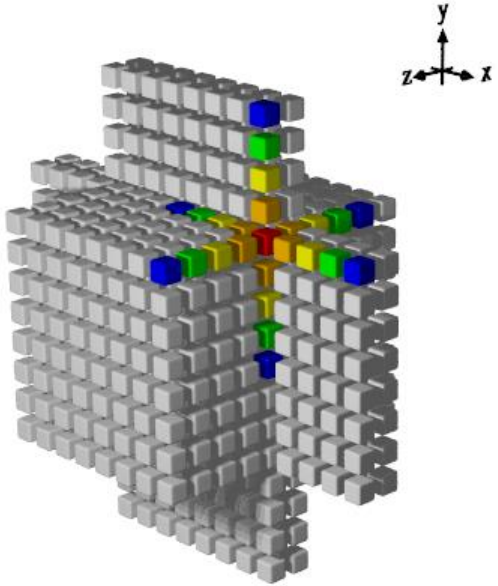
$u(t) \rightarrow u(t+1)$

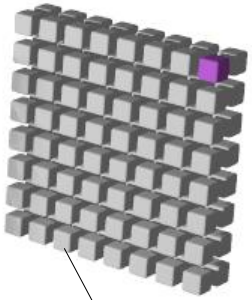
Example 3D stencil



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

Example 3D stencil

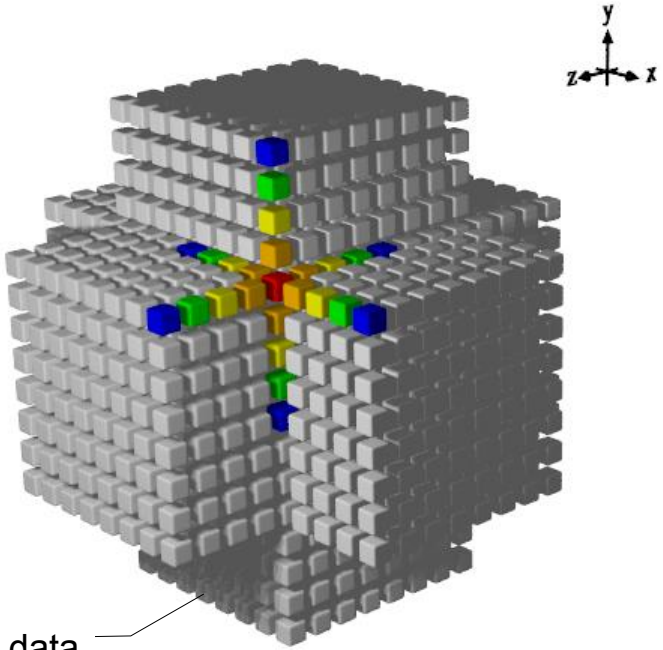




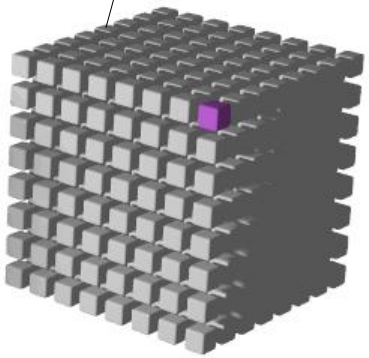
Outer 3D loop iterates in z dimension

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

Example 3D stencil



Entire problem domain



Repeat for u(t+2)…

"Halo" data regions

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

Performance considerations

Performance issues

- Stencils are often memory-bound
- Naïve implementation may read same input values multiple times for each time-step with poor cache locality

Common optimization techniques

- Reuse memory of older time steps
- Evaluate multiple neighboring results in parallel using SIMD
- Increase SIMD reuse in 2D or 3D via vector folding
- Evaluate results in blocks sized and shaped to maximize cache reuse (“cache-blocking”)
- Evaluate multiple blocks in parallel using hyper-threading and multi-core
- Extend spatial blocking concept to temporal blocking
- Divide problem across multiple nodes in a cluster

Stencil Implementation

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

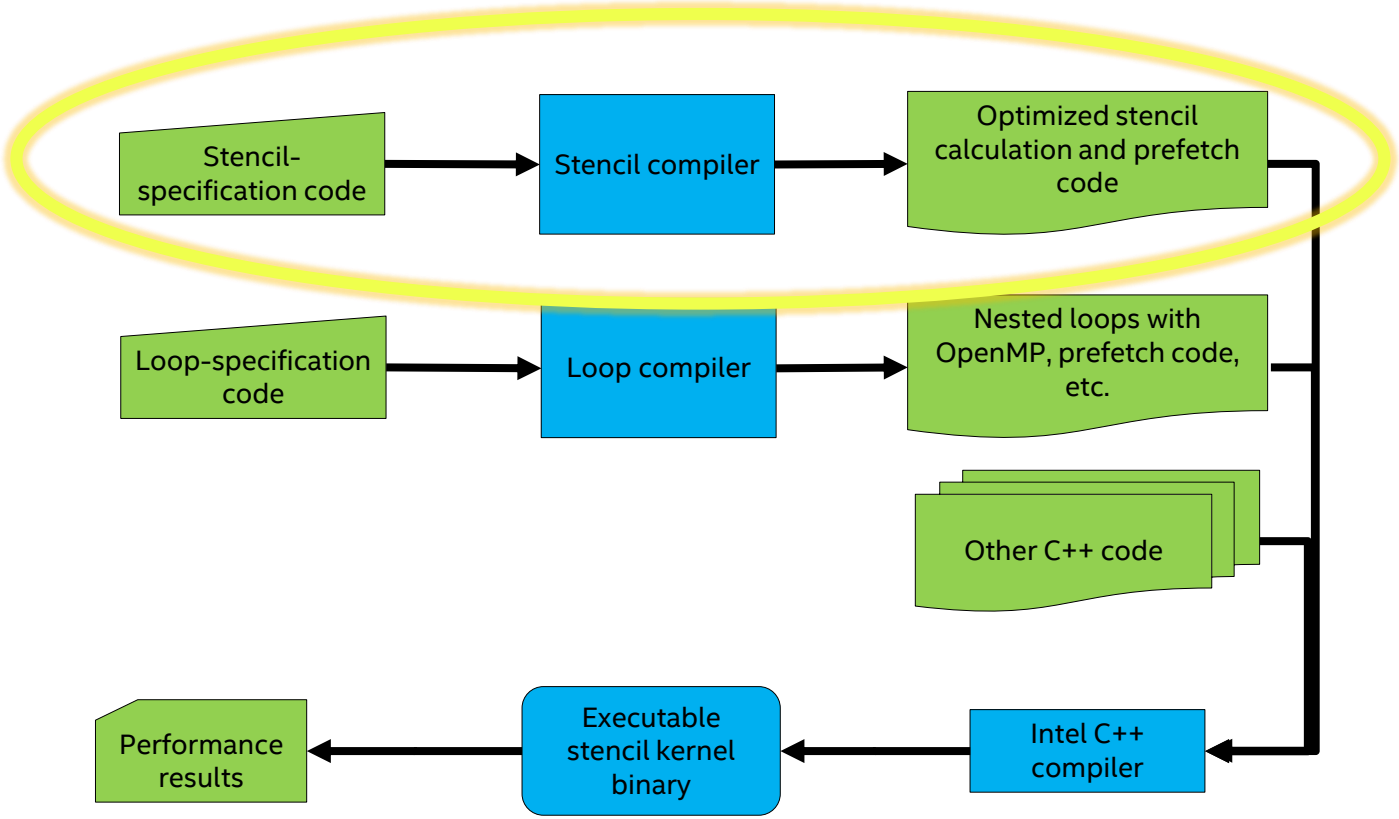
Original impetus

- Tool to evaluate the benefits of vector-folding on multiple stencil kernels
- Expanded to include other optimizations listed earlier

Goals

- Create high-performing code from a straightforward specification of stencil equations
- Provide a simple kernel to test stencil performance
  - Expose [most] optimization trade-off choices without requiring code changes
  - Automate searching through the optimization design space
- Provide ability to integrate code into larger applications

High-Level Flow



Stencil Compiler

Goal: automate the process of creating high-performance stencil computation code

Input: C++ scalar code following certain rules

- Inherit from a C++ abstract 'StencilBase' class to create a new stencil type
- Define the grid(s) to be used and the names of their dimensions, e.g., "t", "x", "y", "z"
- Create a mathematical expression to define how grid values are related

Process

- Compile stencil specification with existing code to create a new stencil-compiler executable
- Run executable, specifying any stencil parameters (e.g., radius), target architecture, etc.
- Code generator executes the stencil expression to create an abstract syntax tree (AST)
- AST is traversed, optimizations are applied to it, and optimized code is output

Output

- Efficient C++ function to calculate stencil with SIMD, vector-folding, sub-expression reuse, etc.
- Functions for prefetching to L1 and L2
- Declarations of required grids and calculated halo sizes

Example 2: Iso3DFD Stencil

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

Similar to earlier example, but a bit more complicated

- 51-point stencil
- Radius = 8
- Inputs from *t-1* and constant *v* grid
- 61 FP ops

Iso3DFD Stencil Specification for YASK

```
#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);   }

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

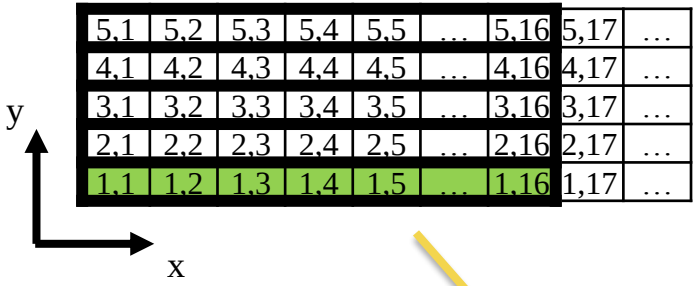
Declare grids and parameters

Define equation for pressure at t+1

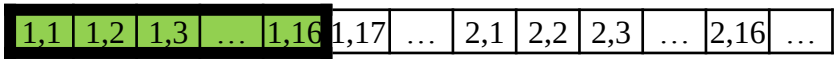
Example Stencil-Compiler Feature: Automatic Vector Folding

Background: Traditional 1D vectorization

Logical indices in 2D



Layout in memory (1D)

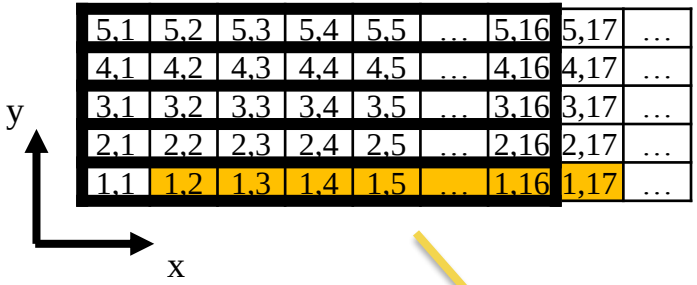


- Traditional 1D vectorization layout (16×1)
- First aligned vector (1,1 … 1,16) is shaded
- Read with simple and efficient aligned vector load

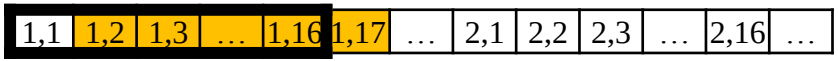
Automatic Vector Folding

Background: Traditional 1D vectorization

Logical indices in 2D



Layout in memory (1D)

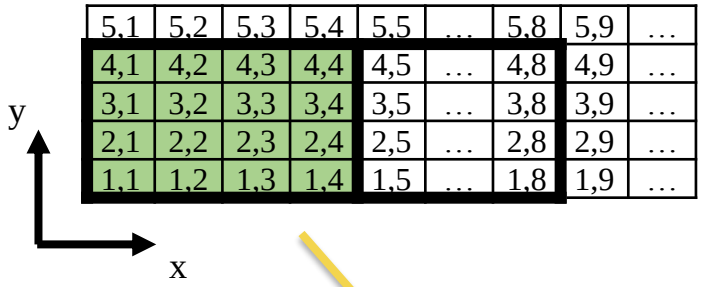


- Unaligned vector (1,2 … 1,17) is shaded
- Can use simple unaligned load or two aligned loads plus a simple shift instruction to create vector

Automatic Vector Folding

Example: 2D “4x4” vector folding

Logical indices in 2D



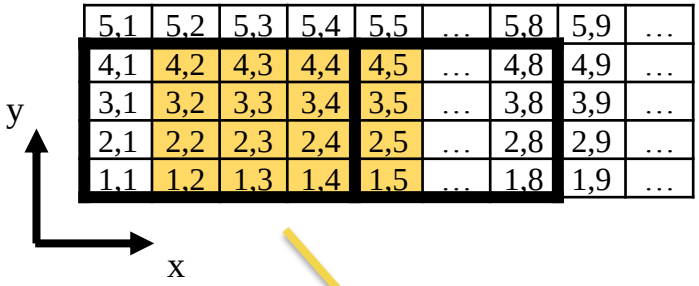
Layout in memory (1D)

- 2D vector-folding layout (4×4)
- First aligned vector (1,1 … 4,4) is shaded
- Read with simple and efficient aligned vector load

Automatic Vector Folding

Example: 2D “4x4” vector folding

Logical indices in 2D

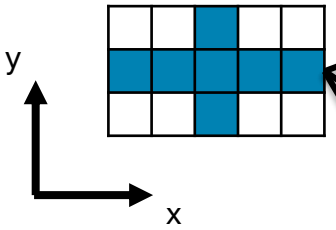


Layout in memory (1D)

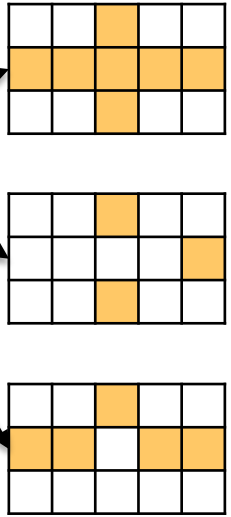
- Unaligned 4×4 vector (1,2 … 4,5) is shaded
- To read, two aligned vectors (1,1 … 4,4 and 1,5 … 4,8) are loaded, then 12 elements from the first and 4 from the second are assembled into a SIMD register using a permute instruction

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:

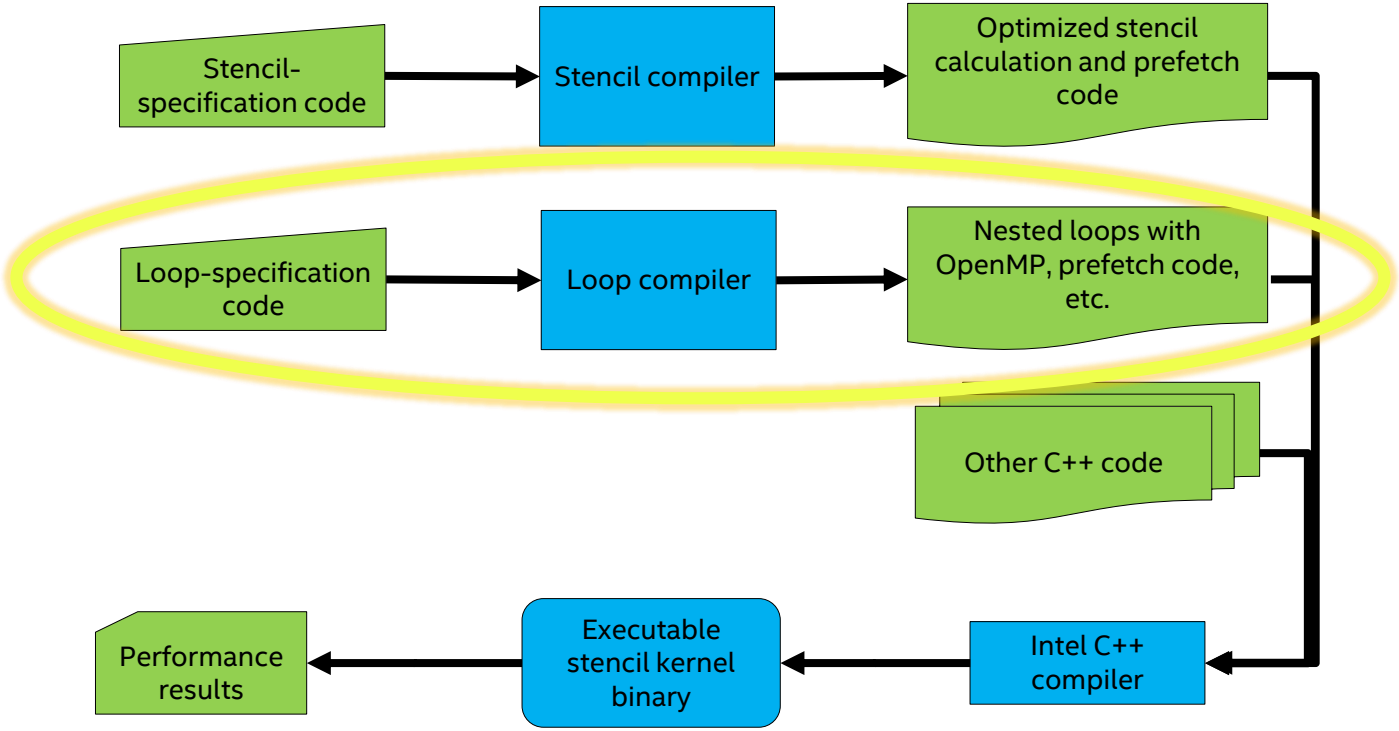


Full prefetch function loads all 7 cache lines

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

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

High-Level Flow

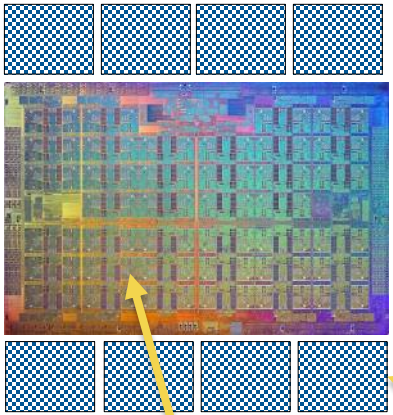


Loop Compiler

Goal: Generate code to visit each point of an n-dim space

- Input: Very simple DSL
  - Example: “omp loop(y,x) { prefetch(L1) loop(z) { calc(stencil); } }”
  - Can easily change loop paths, index ordering, other features
- Output: C++ code
  - Loops annotated with OMP as requested
  - Inner loop might generate several loops, e.g.,
    - Prefetch L2
    - Prefetch L1
    - Compute stencil and prefetch L2 and L1
    - Compute and prefetch L1 only to avoid over-prefetching L2
  - This example would use the correct automatically-generated compute and prefetch functions generated by the stencil compiler, depending on the inner-loop iteration direction

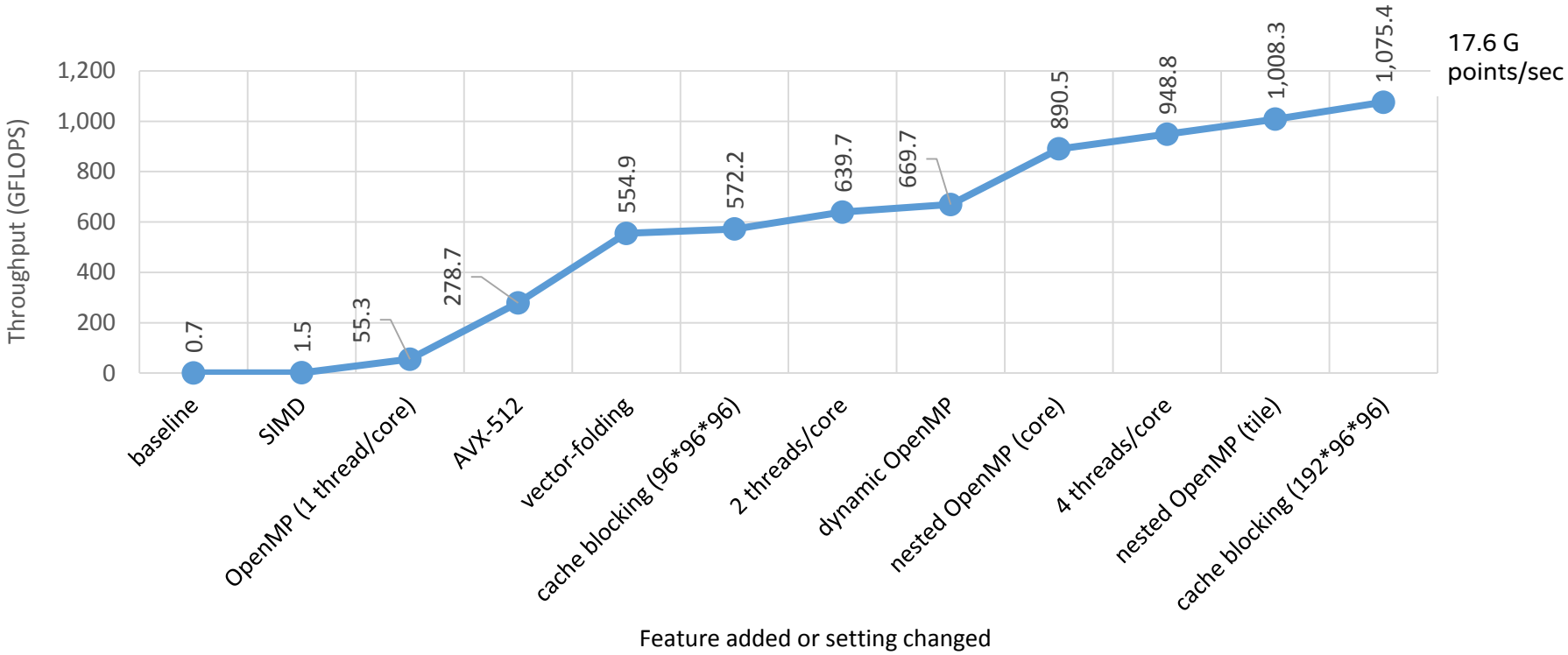
Platform: Intel® Xeon Phi™ Processor



- Up to 72 cores
- Tested part is the 7250 model with 68 cores
- AVX-512 ISA: 8 DP or 16 SP FP SIMD

- Up to 8 channels of on-package Multi-Channel Dynamic Random-Access Memory (MCDRAM)
- Tested part has 16GiB of MCDRAM

Example Optimizations Applied Manually 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.

Automatic Tuner

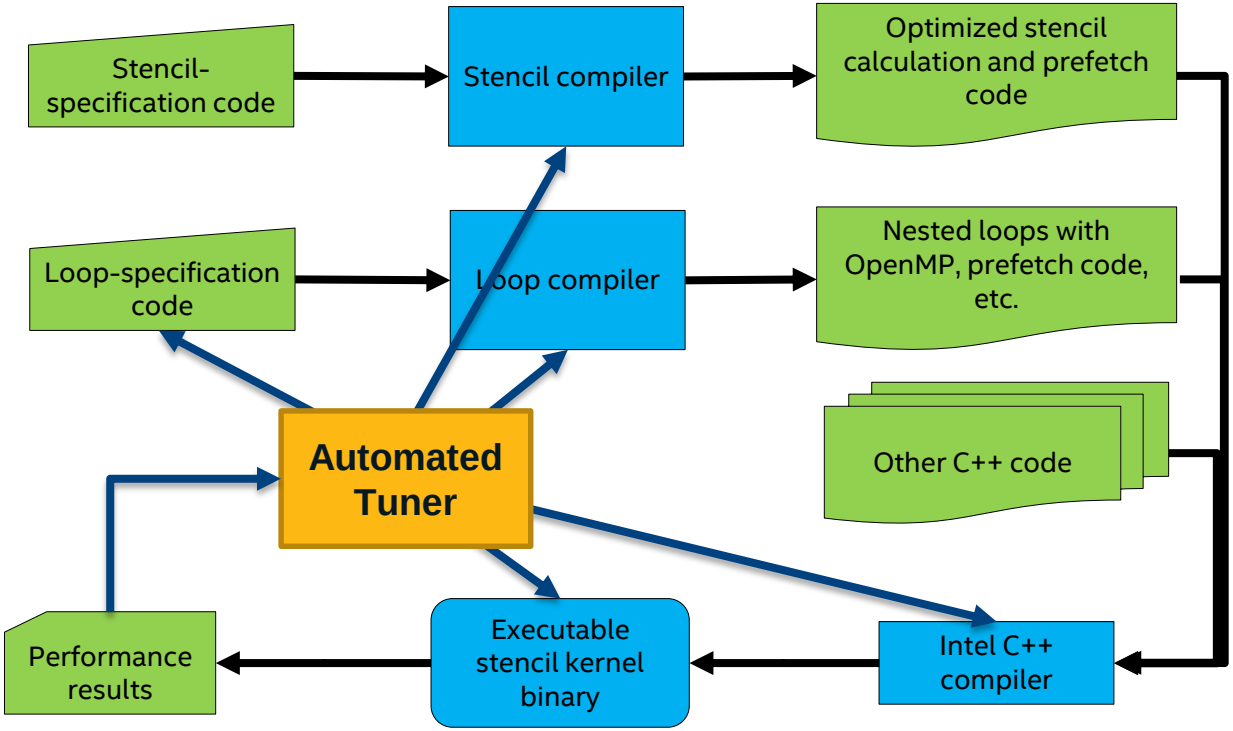
Challenge

- 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

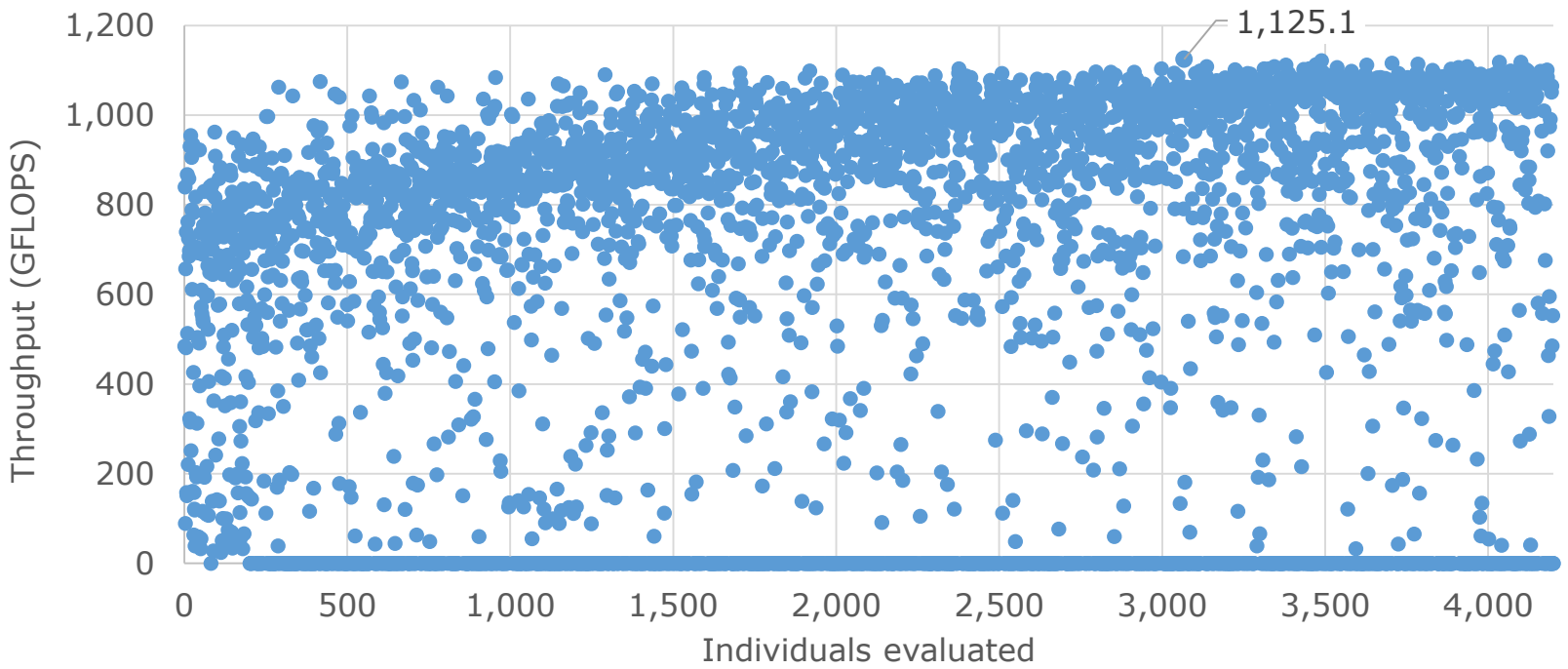
Solution

- 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

High-Level Flow with Tuner



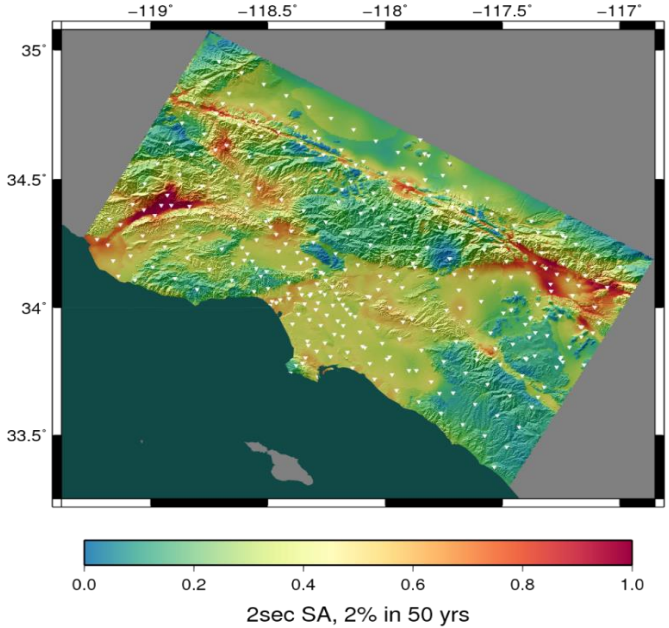
Example Optimizations Applied with Automatic Tuner to Iso3dfd



Example 3: 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
- First ever open source release this year (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 Numerics

Finite Difference code

- Using a staggered-grid scheme
- Fourth-order accurate in space and second-order accurate in time

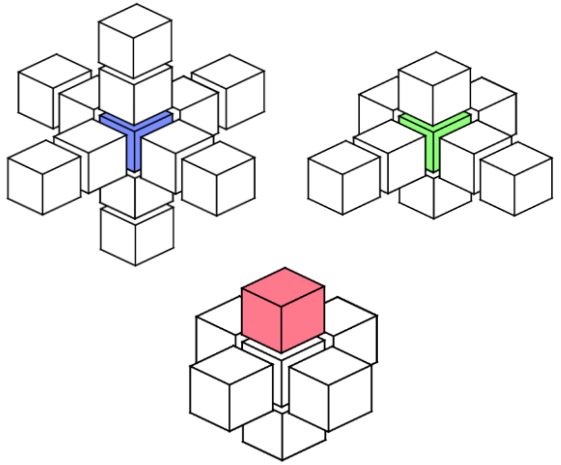
Fifteen grids updated in every time-step

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

Fifteen grids ⇒ fifteen stencils

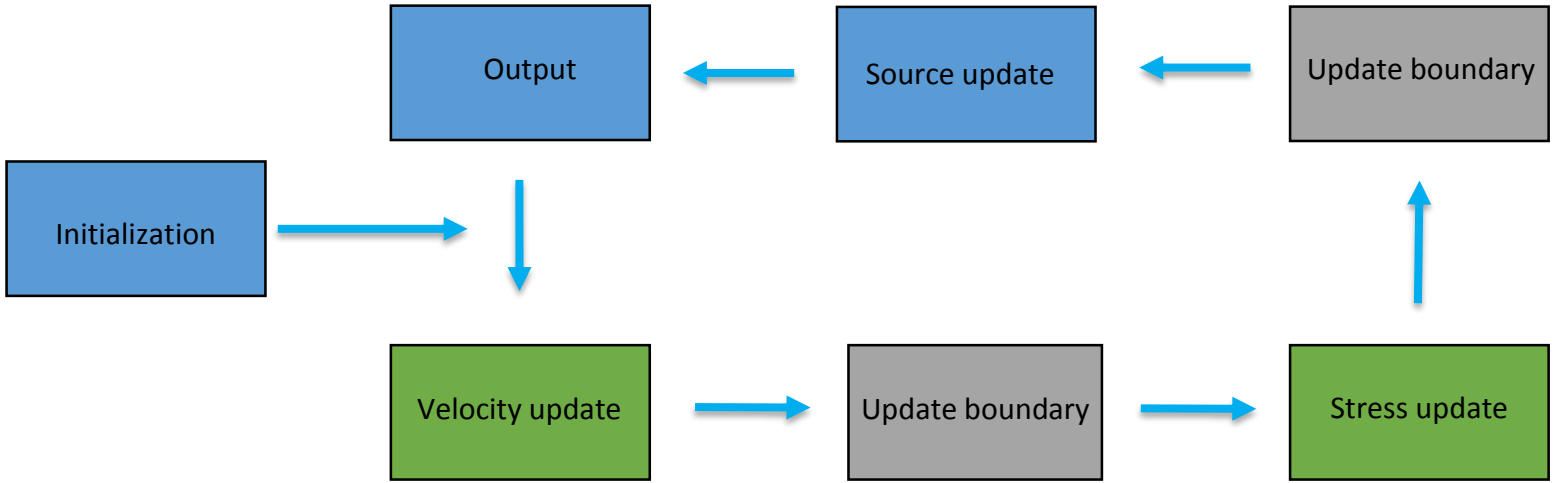
- Nine 13-point stencils
- Six 9-point stencils

Additionally, free surface boundary computation every time-step



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

AWC-ODC-OS Control Flow



Schematic diagram of AWP-ODC-OS control flow

- Green boxes are kernel stencil updates applied to all 15 updated grids
- Grey boxes are boundary updates that are stencils applied at the z=0 boundary
- All other procedures are in blue

Currently, grid data structures and velocity and stress update functions are provided by YASK and integrated into rest of application framework, which uses YASK APIs to access grid data

Preliminary AWC-ODC-OS performance

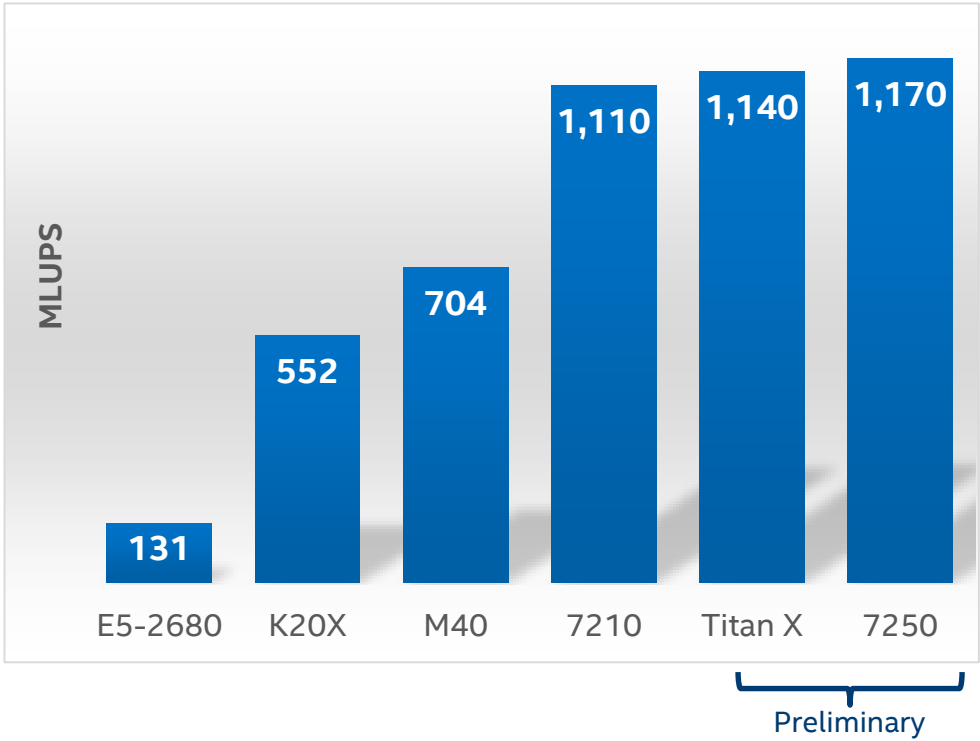
Numerics: Velocity + Frequency-dependent viscoelasticity and free-surface boundary conditions

Domain sizes chosen to use most of the available memory for each platform (MCDRAM for Intel Xeon Phi processors)

Performance measured in number of Lattice Update Points per Second, updating 15 grids

Compared time-to-solution per grid-point for:

- Single-socket Intel® Xeon® processor E5-2680 v3
- Intel Xeon Phi processor 7210 and 7250
- NVIDIA K20X, M40 and Titan X*



Summary

YASK motivation

- Plethora of techniques exist to optimize stencil code
- Difficult to code optimization and explore trade-offs

YASK enables the HPC programmer to

- Generate high-performing stencil code rapidly and accurately for Intel Xeon and Intel Xeon Phi processors
- Explore optimization options easily and automatically
- Integrate resulting optimizations into larger applications

Resources

Software available

- YASK: https://github.com/01org/yask (MIT OS license)
- AWP-ODC-OS: https://github.com/HPGeoC/awp-odc-os (BSD OS license)

Related collateral

- YASK article: https://software.intel.com/en-us/articles/recipe-building-and-running-yask-yet-another-stencil-kernel-on-intel-processors
- High Performance GeoComputing Lab: http://hpgeoc.sdsc.edu
- Source of AWP-ODC-OS information and data provided by UCSD: https://anl.app.box.com/v/IXPUG2016-presentation-13

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

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.

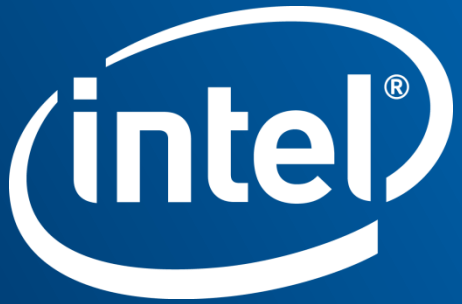
Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

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


intel®