Y.A.S.K.

Yet Another Stencil Kernel

Chuck Yount, Principal Engineer
Intel Corporation

Feb. 1, 2016

chuck.yount@intel.com

Legal Disclaimer & Optimization Notice

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

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

Optimization Notice

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

Notice revision #20110804

Outline

Overview

YASK Features

- Vector folding and the fold builder
- Loop-code generator
- Memory accessor
- Debug output

Using YASK

- Build and test
- Output
- Use model
- Run-time options
- Stencil, vectorization, loop, and advanced customization
- Collaboration

Overview

YASK: Yet Another Stencil Kernel

- Goal: facilitate exploration of the stencil-performance design space for Intel® Xeon Phi™ coprocessor or any Intel processor supporting the AVX-512 instruction set

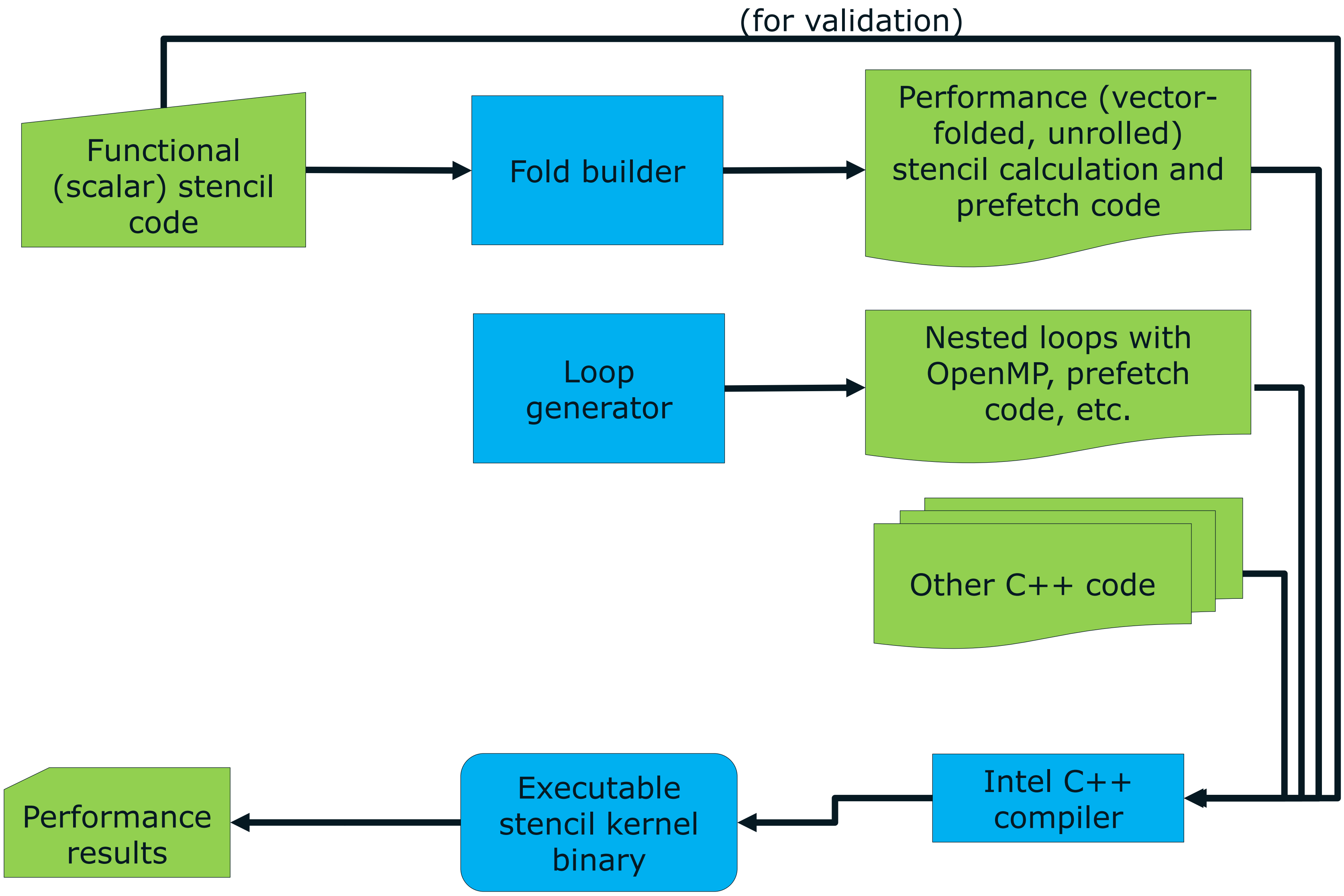
Features

- Supports trade-off studies for coding options for
  - Vector-folding
  - Cache blocking
  - Memory layout
  - Loop construction
  - And more
- Is a collection of C++ code, code-generators and other scripts
- Focused on single-node OpenMP optimizations (not MPI enabled at this time)

See also the related iso3dfd kernel (by Leo B, Cedric A, Philippe T)

YASK Features

High-level components



(for validation)
Functional (scalar) stencil code
Fold builder
Performance (vector-folded, unrolled) stencil calculation and prefetch code
Loop generator
Nested loops with OpenMP, prefetch code, etc.
Other C++ code
Intel C++ compiler
Executable stencil kernel binary
Performance results

Vector-folding introduction

Concept

- Store small 2D or 3D block of data into each vector
- Pros: reduces memory BW requirements compared to traditional 1D in-line vectors
- Cons: requires data pre-conditioning (element rearranging) and additional shift and blend operations preceding SIMD arithmetic operations

Results

- Significant speedup shown on Intel® Xeon Phi$^{\text{TM}}$ Coprocessor
- Combining with loop tiling enables even more speedup

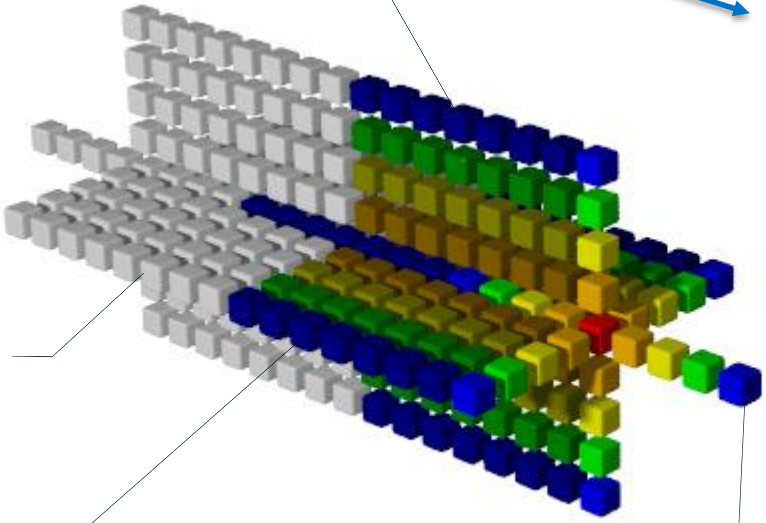
For more information

- Refer to paper on Vector Folding from HPCC 2015

Traditional in-line 1D vectorization

25-point 3D stencil input vectorized using 8-element vectors, each parallel to x-axis

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



Previous iteration

Current iteration

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

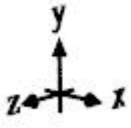
One 8-element vector output per iteration

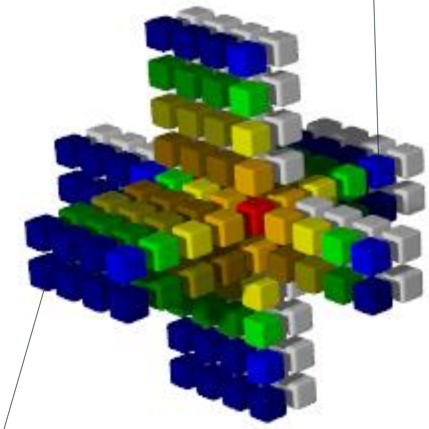
Steady-state memory BW = **17** new cache lines input to calculate each vector of output

*Assuming cache line size = vector size.

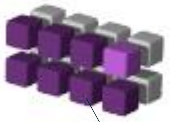
Reduce BW via vector folding

25-point 3D stencil input vectorized using 8-element vectors, *each containing a 4x2 grid in the x-y plane*





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



One 8-element (4x2) vector output per iteration

Need to read only 7 new cache lines for each iteration (vectors overlap in x-y dimensions within an iteration and in z dimension between iterations)

Steady-state memory BW = **7** new cache lines input to calculate each vector of output: **2.4x lower** than in-line

Fold-builder code generator

Goal: automate the tedious and error-prone process of creating high-performance stencil code

Input

- Inherit from a C++ abstract ‘StencilBase’ class to create a new stencil type
- Implement the ‘value’ method to define how one point is calculated from others
- Use loops, functions for coefficients, recursion, etc.

Process

- Compile code into fold-builder executable
- Run executable, specifying any stencil parameters (e.g., order), target architecture, etc.
- Code generator evaluates the ‘value’ method to create an abstract syntax tree (AST)
- AST is traversed, and optimized code is output

Output

- Efficient function to calculate stencil
  - Unrolled loops, intrinsics to construct unaligned vectors of points, etc.
  - Calls to memory accessor object
- Functions for prefetching to L1 and L2
  - Whole stencil, or just ‘leading edge’ in any direction

Example input stencil code

```
virtual GridValue value(Grid5d& u, int tW, int t0, int v0, int i, int j, int k) const {
  if (tW <= t0)
    return u(tW, v0, i, j, k);
  int tm1 = tW - 1;  // one timestep ago.
  int tm2 = tW - 2;  // two timesteps ago.

  // start with center value.
  GridValue v = _coeff[0] * value(u, tm1, t0, v0, i, j, k);

  // add values from x, y, and z axes.
  for (int r = 1; r <= _order/2; r++) {
    v += (
       // x-axis.
       value(u, tm1, t0, v0, i-r, j, k) +
       value(u, tm1, t0, v0, i+r, j, k) +

       // y-axis.
       value(u, tm1, t0, v0, i, j-r, k) +
       value(u, tm1, t0, v0, i, j+r, k) +

       // z-axis.
       value(u, tm1, t0, v0, i, j, k-r) +
       value(u, tm1, t0, v0, i, j, k+r)

     ) * _coeff[r];
  }

  // temporal and velocity components.
  v = (2.0 * value(u, tm1, t0, v0, i, j, k))
     - value(u, tm2, t0, v0, i, j, k) // value from previous time.
     + (v * _vel(i, j, k));       // velocity.

  return v;
}
```

Example output stencil code

```
void calc_stencil_vector (StencilContext& context, int t0, int v0,
                          long veci, long vecj, long veck)
{
  …
  // Aligned vector block from grid at t=t0+0 at point 0, 0, 0.
  realv vec2 = context.grid->readVec (t0 + 0, v0, veci, vecj, veck);
  …
  // Unaligned vector block from grid at t=t0+0 at point 0, -1, 0.
  realv vec9;
  realv_permute2 (vec9, ctrl_A3_..._B14, vec8, vec2);
  …
  realv vec63 = vec4 +
    (((vec5 * vec2) + ((vec6 + vec7 + vec9 + vec11 + vec13 + vec15) * vec16) +
      ((vec17 + vec18 + vec19 + vec20 + vec21 + vec22) * vec23) +
      ((vec24 + vec25 + vec26 + vec27 + vec28 + vec29) * vec30) +
      ((vec31 + vec32 + vec8 + vec10 + vec12 + vec14) * vec33) +
      ((vec34 + vec35 + vec37 + vec39 + vec41 + vec43) * vec44) +
      ((vec45 + vec46 + vec47 + vec48 + vec49 + vec50) * vec51) +
      ((vec52 + vec53 + vec54 + vec55 + vec56 + vec57) * vec58) +
      ((vec59 + vec60 + vec36 + vec38 + vec40 + vec42) * vec61)) * vec62);

  // Set final result at offset 0, 0, 0.
  context.grid->writeVec (vec63, t0 + 1, v0, veci, vecj, veck);
}
```

- The above code was created from the code on the previous slide using this command:
  ```
  ./foldBuilder -ts 1 -or 16 -st iso3dfd -cluster 1 1 1 -es 50 -p512 1 4 4
  ```
- See the foldBuilder help message for information on the options
- The above code is a small sample from almost 800 lines of code in the function
- The resulting compiled loop body contains 127 instrs, including 7 SIMD FMAs, 48 SIMD add/sub/muls, 12 VALIGND, and 12 VPERMI2D

Example output prefetch code

```
Void prefetch_L1_stencil_vector(StencilContext& context, int t0, int v0,
                                long veci, long vecj, long veck)
{
  const char *p = 0;

  // Aligned vector block from grid at t=t0 at point 0, 0, 0.
  p = (const char *) context.grid->getVecPtr (t0, v0, veci, vecj, veck);
  _mm_prefetch (p, L1);

  // Aligned vector block from grid at t=t0-1 at point 0, 0, 0.
  p = (const char *) context.grid->getVecPtr(t0-1, v0, veci, vecj,
veck);
  _mm_prefetch (p, L1);
  …
}
```

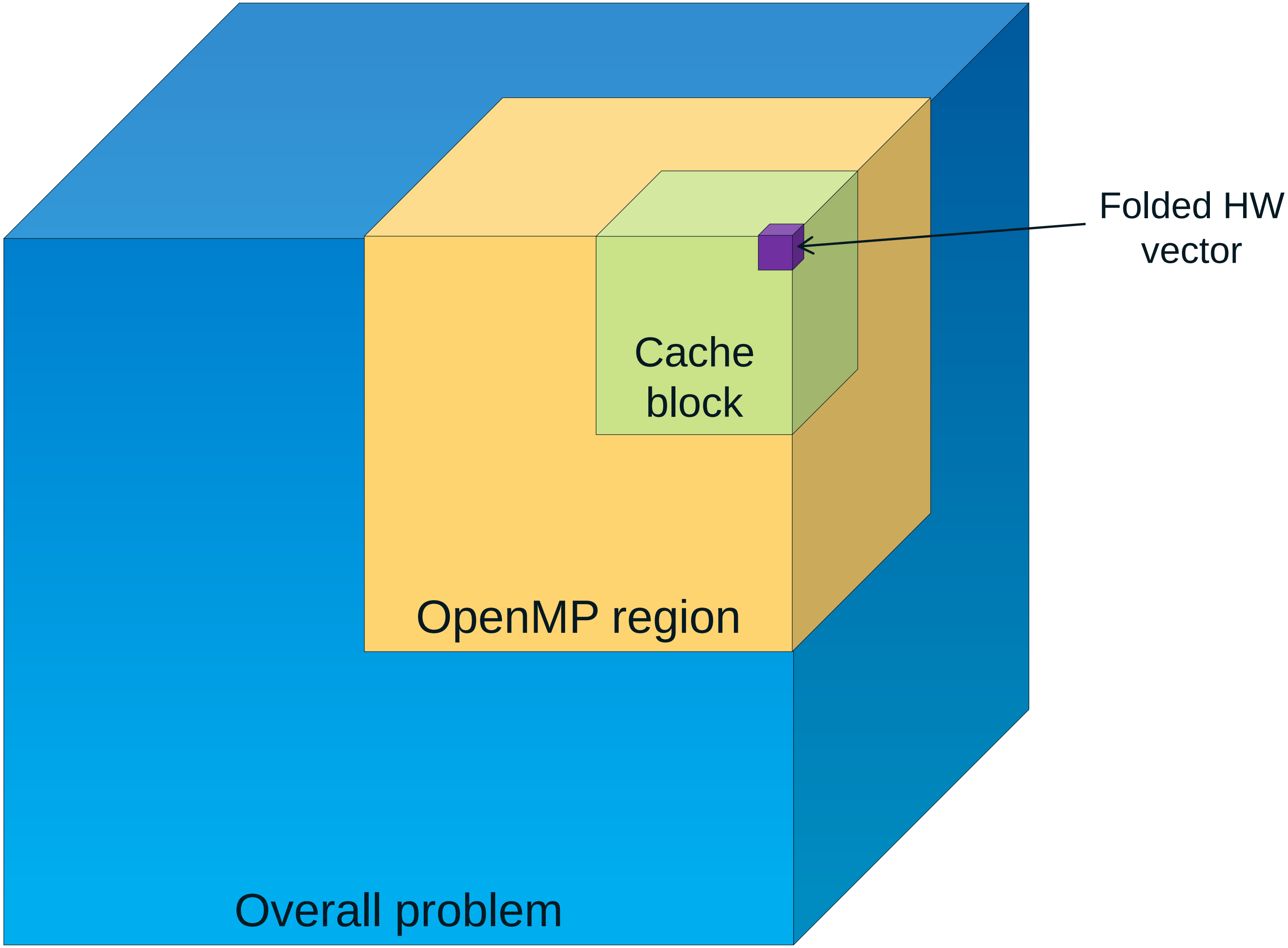
This prefetch code can be optionally and automatically called by code generated by the loop-code generator

Loop-code generator

Script that generates code for nested loops

- Input: Very simple DSL (domain-specific language)
  - omp loop(y) { crew loop(x) { loop(z) { calc(stencil); } } }
  - Can easily change loop types, index ordering
- Output: C++ code to be included in function bodies
  - Loops annotated with OMP/CREW as requested
  - Inner loop might generate several loops:
    - Prefetch L2
    - Prefetch L1
    - Calculate stencil and prefetch L2 and L1
    - Calculate and prefetch L1 only to avoid over-prefetching L2

Loops applied at multiple block levels



Memory accessor

- C++ classes to allocate and access 4D matrices of vectors of floats or doubles
  - Template parameter specifies ‘w’ dimension (number of 3D matrices)
  - Construction specifies ‘x, y, z’ dimensions and padding sizes; padding includes halos
  - Read and write access via methods: per vector for speed; per element for debug, validation, etc.
- Actual memory layout is encapsulated and defined via inheritance
  - Map 4-D (w, x, y, z) to 1-D mem address
    - 24 simple permutations of minor-to-major ordering
    - More complex mappings possible, e.g., tiling, space-filling curves
  - ‘w’ dimension is used for time and/or variable indices
- Uses concrete inheritance to allow inlining
  - Gives compiler full access to memory-layout formula
  - Allows common sub-expression elimination and other optimizations

Debug features

Can enable or disable various output by setting macros and rebuilding, e.g.,

- TRACE: print each stencil calculation
- TRACE_MEM: print each matrix read, write, prefetch, eviction
- TRACE_INTRINSICS: print before-and-after each permutation

Built-in memory-access tracker

- Models an infinite L1 or L2 cache
- Tracks reads, writes, prefetches, evictions
- Reports any un-prefetched read or un-read prefetch
- Reports summary stats
- Very useful for debugging prefetch code

Example cache-model output

```
modeling cache...
cache L2: redundant prefetch of 0x2aaabfa45a40 at line 193 after a read at line 85.
cache L2: redundant prefetch of 0x2aaabfa45a80 at line 193 after a read at line 85.
cache L2: redundant prefetch of 0x2aaabfa45a40 at line 195 after a prefetch at line 193.
cache L2: redundant prefetch of 0x2aaabfa45a40 at line 196 after a prefetch at line 195.
…
done modeling cache...
cache L2: read of 0x2aaabf9c3240 from line 85 without any eviction.
cache L2: read of 0x2aaabf9c3280 from line 85 without any eviction.
…
cache L2: prefetch of 0x2aaabfa53b00 from line 318 without any read.
cache L2: prefetch of 0x2aaabfa53b40 from line 318 without any read.
…
cache L2 stats:
 cur size = 324714 lines (19.8190 MB).
 max size = 324714 lines (19.8190 MB).
 ave size = 185126 lines (11.2992 MB).
 num reads = 722400.
  num reads of missing lines = 0.
  num lines read but never evicted = 321700.
 num prefetches = 1458800.
  num prefetches of lines never subsequently read = 3014.
  num prefetches of lines already in cache = 404686.
 num evictitions = 0.
  num evictions to non-existant lines = 0.
 num prefetches into L1 = 729400.
  num prefetches into L1 of missing lines = 0.
```

Using YASK

Initial build and test

Install

- Download the code from the ‘GIT REPO’ link at https://01.org/yask
- Install all the prerequisites from the README file

Build and run the default test program

- Type ‘make -arch *arch-code*’ per the README file
- Run the program using the stencil-run.sh script
  - Use the -mic option to run on a Xeon Phi coprocessor
  - Run under SDE to emulate hardware you don’t have
  - Run natively
- If it doesn’t build and/or run, check the prerequisites

Typical run and output

Settings are printed

- Sizes: problem, region, block, cluster and vector
- Stencil shape and order
- Other miscellaneous compile-time and run-time settings

A number of trials (default=3) are run

- Each trial executes a number of iterations (default=50)
- Time and throughput (million points per sec) are printed
- Best result across the trials is re-printed

Validation

- If the '-v *n*' option was used, *n* validation iteration(s) are run, and 'PASSED' or 'FAILED' is printed
- Validation is slow; run with a small problem size

Use model

Review

- YASK is a tool for exploring the stencil design space
- It is not a library

Typical usage model

- Identify stencil(s) used in your application
- Use existing stencil(s) in YASK or write your own
- Use YASK to find well-performing parameters
- Integrate the stencil code back into your application

Run-time options

Settings controlled from the ‘stencil.*arch*.exe’ binary

- Problem size: -d*
- Number of regions: -nregions*
- Cache-block size: -b*
- Padding: -p* (used to fine-tune data alignment across rows and columns)
- Number of trials and iterations: -t, -i, -v

Settings controlled from the ‘stencil-run.sh’ script

- Binary selection via ‘arch’ option
- OpenMP affinity
- Which Xeon Phi coprocessor or other host to use
- Number of CPU cores and threads
- Run with ‘-h’ option to get help

Stencil customization

Stencil Shape

- Use the ‘shape=*stencil-name*’ argument to the make command to select a stencil
  - The *stencil-name* string is passed to the foldBuilder tool
- Current provided stencils
  - ‘Iso3dfd’ implements the isotropic acoustic wave equation
  - ‘3axis’, ‘9axis’, ‘3plane’, and ‘cube’ implement common 3D symmetric shapes (defined in the vector-folding paper)
  - ‘Ave’ implements the simple 27-pt stencil from the miniGhost benchmark
- Write your own by modifying code in src/foldBuilder
  - Implement the StencilBase interface using the *Stencil.hpp files as examples
  - Modify genVecCode.cpp to add appropriate command-line options and instantiate your new stencil class

Stencil customization (cont.)

Stencil size

- Use the ‘order=*n*’ argument to the make command to select a stencil
  - The *n* value is passed to the ‘foldBuilder’ tool
  - Default=16; 2 for ‘ave’ stencil
- Current provided stencils
  - Any even value of *n* is allowed
- Write your own by modifying code in src/foldBuilder
  - Follow the existing examples to pass the ‘order’ parameter to your stencil code

Other parameters

- If you’re developing your own stencil, you can add more parameters similar to the ‘order’ one

Stencil customization (cont.)

Advanced

- The provided stencils assume uniformity across the entire 3D grid
  - The ‘foldBuilder’ tool evaluates the stencil code only from the origin to the extent of a vector
- Some stencil applications require special code at boundaries or other conditions
  - To achieve this using the ‘foldBuilder’ tool, you can provide a parameter to distinguish each condition, e.g., top boundary, bottom boundary, etc.
  - Then, you would need to generate separate code for each condition
  - For even more complex stencils, you may need to study and modify the ‘foldBuilder’ code beyond adding new stencils and command-line parameters

Vector-folding customization

Vector fold

- Use the ‘fold='*nx ny nz*'’ argument to the make command to control how much vectorization is done in each dimension
  - The 3 values are passed to the ‘foldBuilder’ tool
  - Example: ‘make fold='1 2 8'’ generate code using a 1x2x8 fold
- See the vector-folding paper for a detailed discussion

Vector cluster

- Use the ‘cluster='*nx ny nz*'’ argument to the make command to control how many vectors are calculated and output in each function call
  - The 3 values are passed to the ‘foldBuilder’ tool
  - The default is 1x1x1, or one HW vector

Loop-structure customization

The ‘gen-loops.pl’ script creates the loop-control code

- There are 3 loop-control codes
  - ‘Outer’ loops break the whole problem into OpenMP regions (typically, only one OpenMP region is used)
  - ‘Region’ loops break each OpenMP region into cache blocks
  - ‘Block’ loops iterate over each vector cluster in a cache block

Usage

- See the Makefile for default invocations or run ‘make -n’
- Run ‘./gen-loops.pl’ without any parameters to get help on more options: index ordering, OpenMP scheduling, etc.
- Run the script before the make command or specify the *LOOP_ARGS variables in the make command to override

Misc. advanced customization

More compile-time options to the make command

- Use ‘crew=*n*’ to enable (n=1) or disable (n=0) Intel Crew threading
  - If you get a link-time error that ‘kmp*’ symbols cannot be found, your compiler does not support crew; use ‘crew=0’
- Use ‘real_bytes=*n*’ to set the size of a float: n=4 for single-precision or n=8 for double-precision (default=4; 8 for ‘ave’ stencil)
- Use ‘MACROS='*macro-settings*'’ set CPP macros
  - ‘NUM_WORKS=*n*’ to create *n* work grids (default=1; 40 for ‘ave’ stencil)
  - ‘MATRIX_BASE=*class-name*’ to change the type for the memory interface object (default=RealMatrix4321)
  - ‘PFDL1=*n1* PFDL2=*n2*’ to change the prefetch distances (defaults=1,8); only used in the prefetch code generated from ‘gen-loops.pl’, not in compiler-generated prefetch code
  - Example: ‘make MACROS='PFDL1=2 PFDL2=15'’

Collaboration

Use the blog at https://01.org/yask to ask and/or answer questions

Submit useful changes for review via github

Contact the author of this presentation for further collaboration opportunities intel®