Optimization Strategies for Intel® Xeon Phi™ Coprocessors

Dr.-Ing. Michael Klemm
Software and Services Group
Intel Corporation
(michael.klemm@intel.com)

Legal Disclaimer & Optimization Notice

Agenda

- Porting Checklist
- Performance Tuning Utilities
- Performance Tuning Hints
- Advanced Performance Tuning

Agenda

- **Porting Checklist**
  - Choosing the right path
  - Host pre-work to characterize and correct
- **Performance Tuning Utilities**
- **Performance Tuning Hints**
- **Advanced Performance Tuning**

Pick the configuration to match the most efficient formulation of your algorithm

- **Choose the right vehicle**
- **Pick your path to performance**
- **Intel® Architecture families' different design goals**
  - Intel Xeon® architecture features large, fast, versatile cores
  - Intel® Xeon Phi™ architecture features smaller and slower but wider and numerous cores
- **Intel® Xeon processors and Intel® Xeon Phi™ coprocessors differ in their optimization plans to match these differences**

First decision: host vs. native vs. offload

- **Native only (running solely on the coprocessor)**
  - Easy to port: change some switches, add libs, drive with ssh or MPI
  - Scalar and IO-intensive parts run slowly
  - Memory limited to 8GB
- **Host with offload**
  - Better performance when serial/sequential fraction is significant
  - Enables memory footprint >> working set, e.g. via pipelining
  - Path to hybrid, since no reverse offload
  - Higher porting costs
  - Need identifiable hotspots to select for offload
- **"Hybrid" host + coprocessor**
  - Best performance for parallel cases
  - More effort to manage concurrency with async or multithreading

Use host-based profiling to identify vectorization/parallelism/offload candidates

- **Start with representative/reasonable workloads!**
- **Use VTune™ Amplifier XE to gather hot spot data**
  - Tells what functions account for most of the run time
  - Often, this is enough
    - But it does not tell you much about program structure
- **Alternately, profile functions & loops using Intel® Composer XE**
  - Build with options `-profile-functions -profile-loops=all -profile-loops-report=2`
  - Run the code (which may run slower) to collect profile data
  - Look at the resulting `dump` files, or open the `xml` file with the data viewer `loopprofileviewer.sh` located in the compiler `./bin` directory
  - Tells you
    - which loops and functions account for the most run time
    - how many times each loop executes (min, max and average)

Loop Profiler - Identify Time Consuming Loops / Functions to Optimize

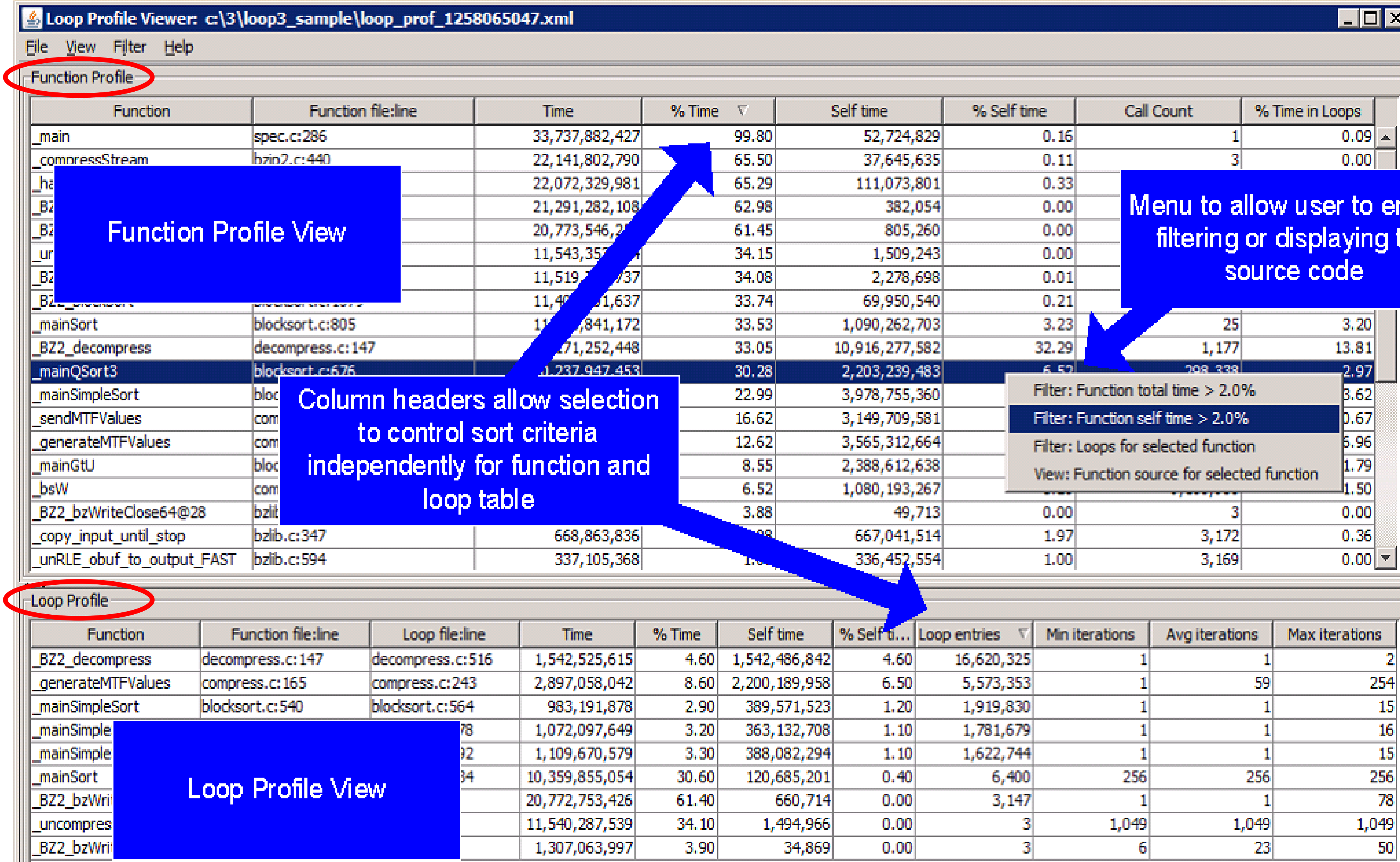
Enables targeting parallelization/optimization efforts to most significant code areas (hotspot identification)

- **Easy to use:**
  - Use compiler switches to add instrumentation to the application
    - Compiler instruments entry and exits of all loops and functions

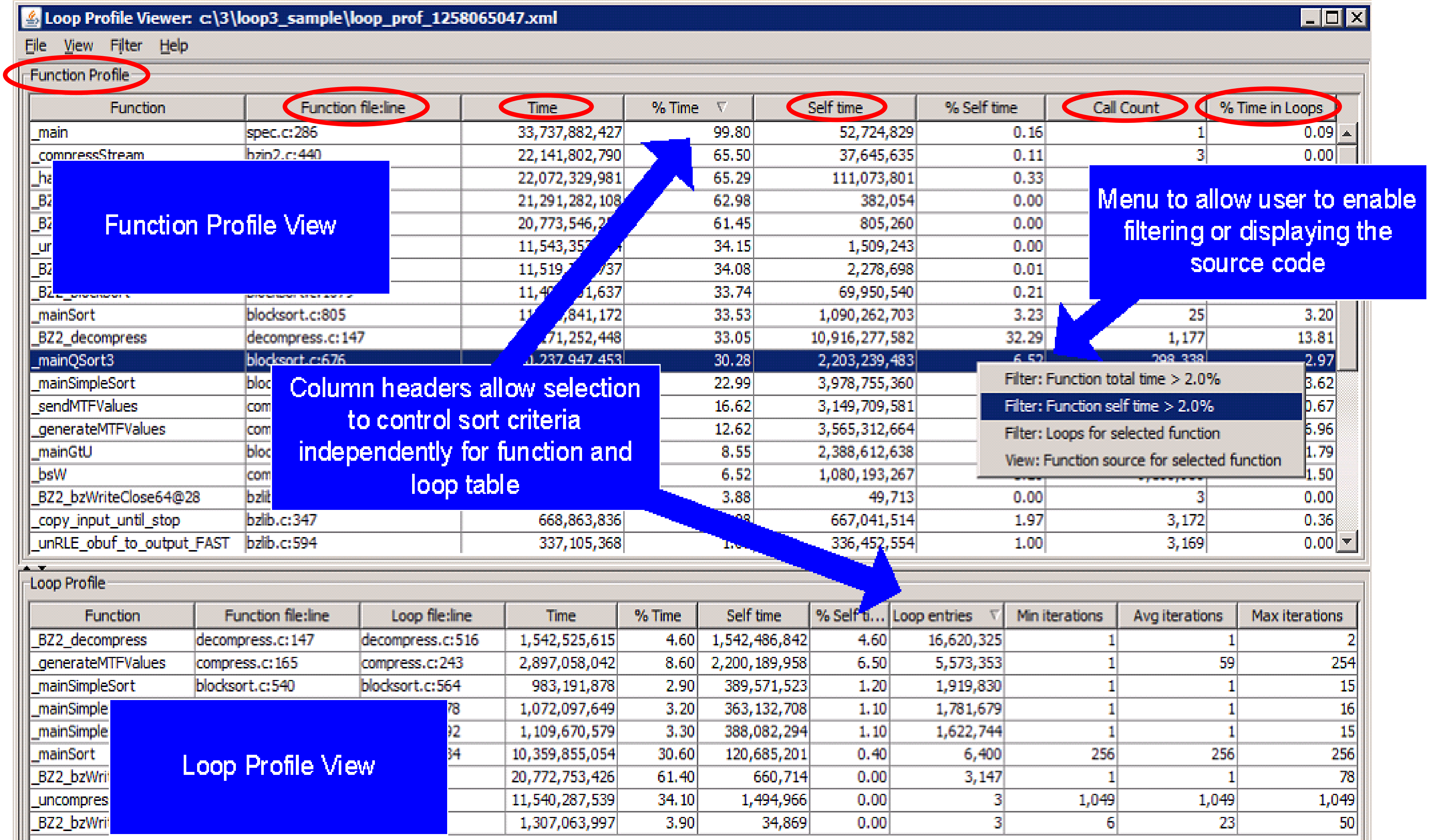
```
icc -O1 -profile-functions -profile-loops=all -profile-loops-report=2…
```

  - Running the application generates a report file with resulting counts
    - Both a human-readable text file (a table) and an XML-file are generated
  - Analyze data by looking at the raw text file, or use the GUI viewer shipped with compiler
- **Report file contains information such as:**
  - Call count of routines
  - Self-time of functions / loops
  - Total-time of functions / loops
  - Average, minimum, maximum iteration counter of loops !!

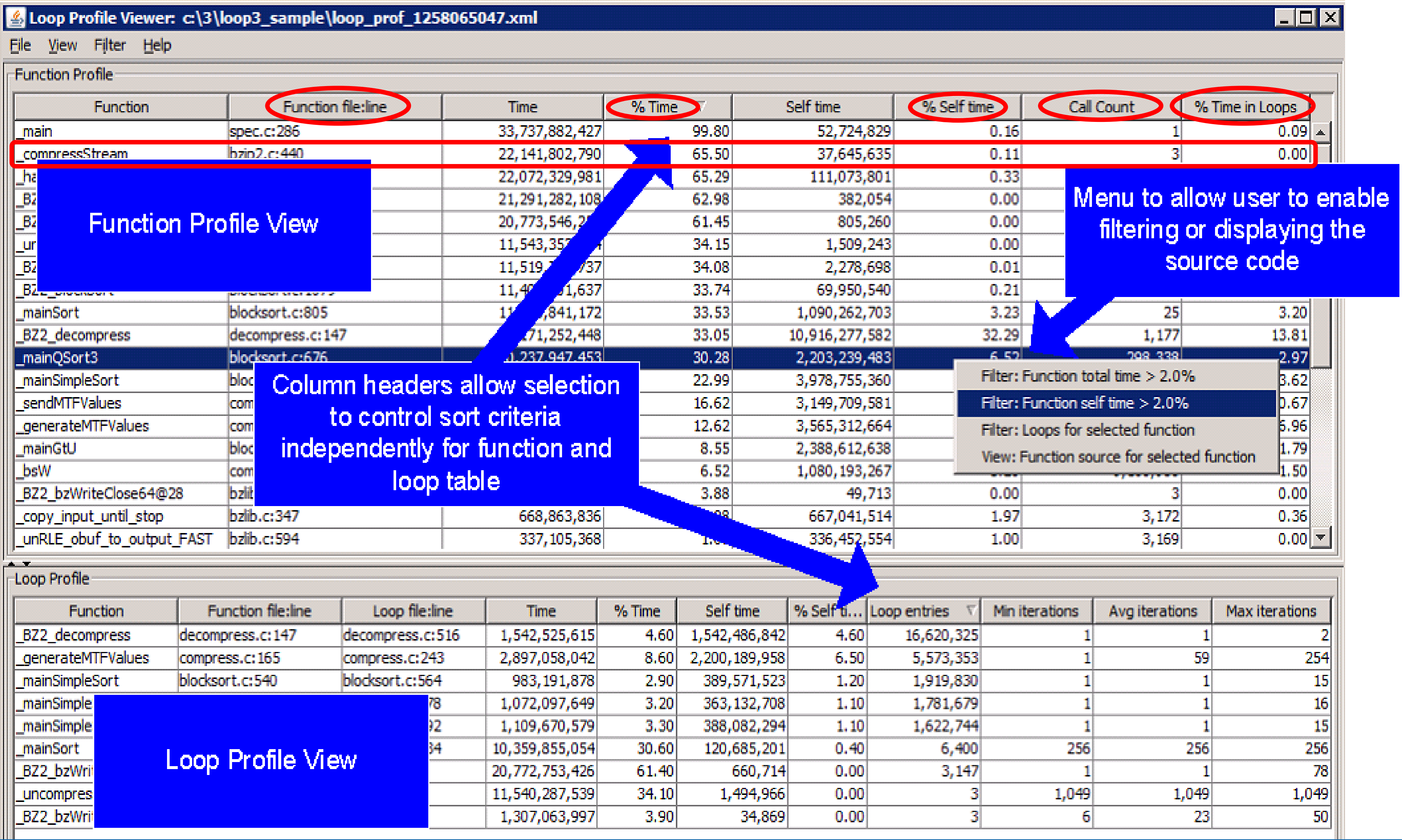
Loop Profile Data Viewer GUI



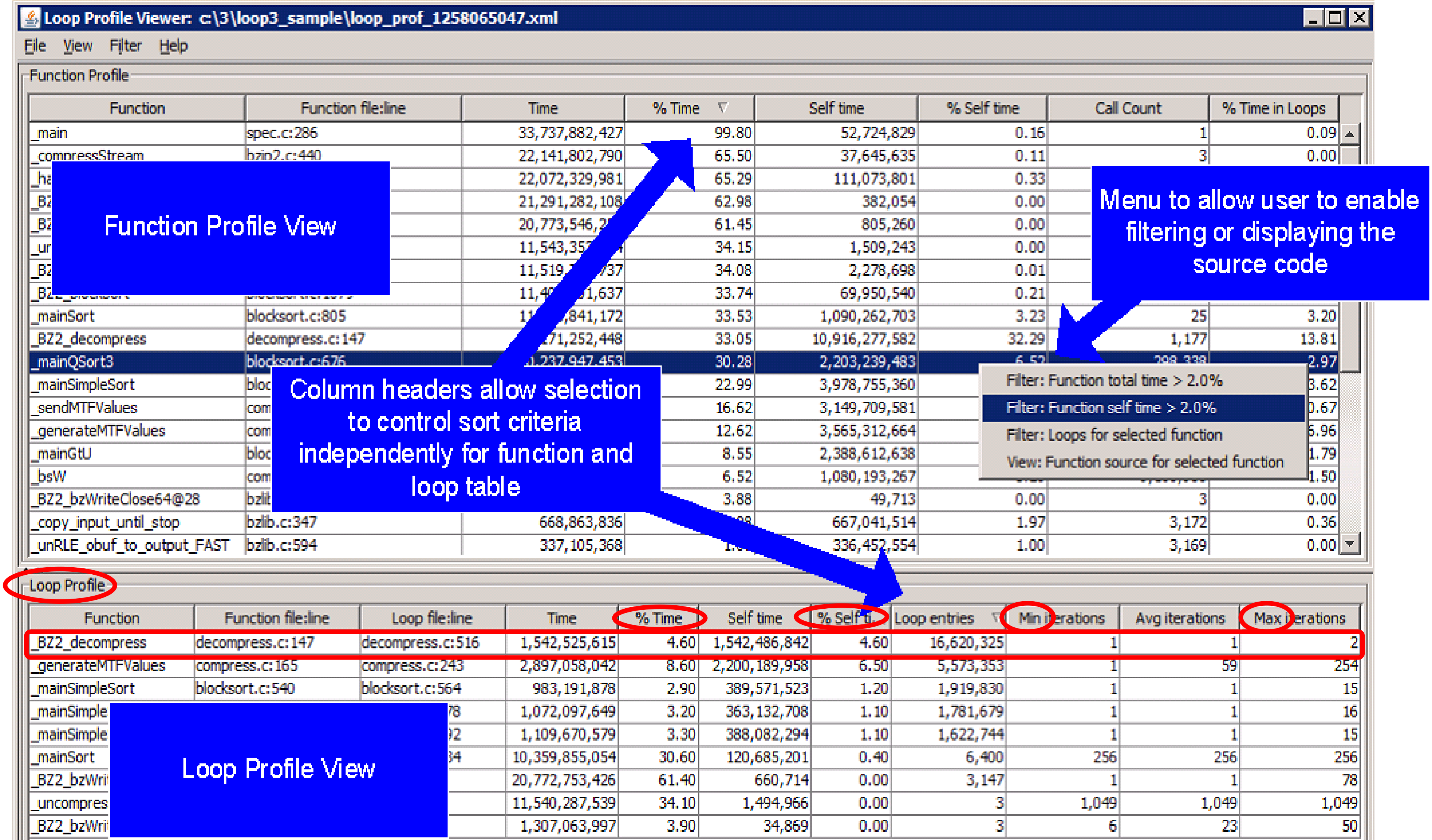
Loop Profile Data Viewer GUI



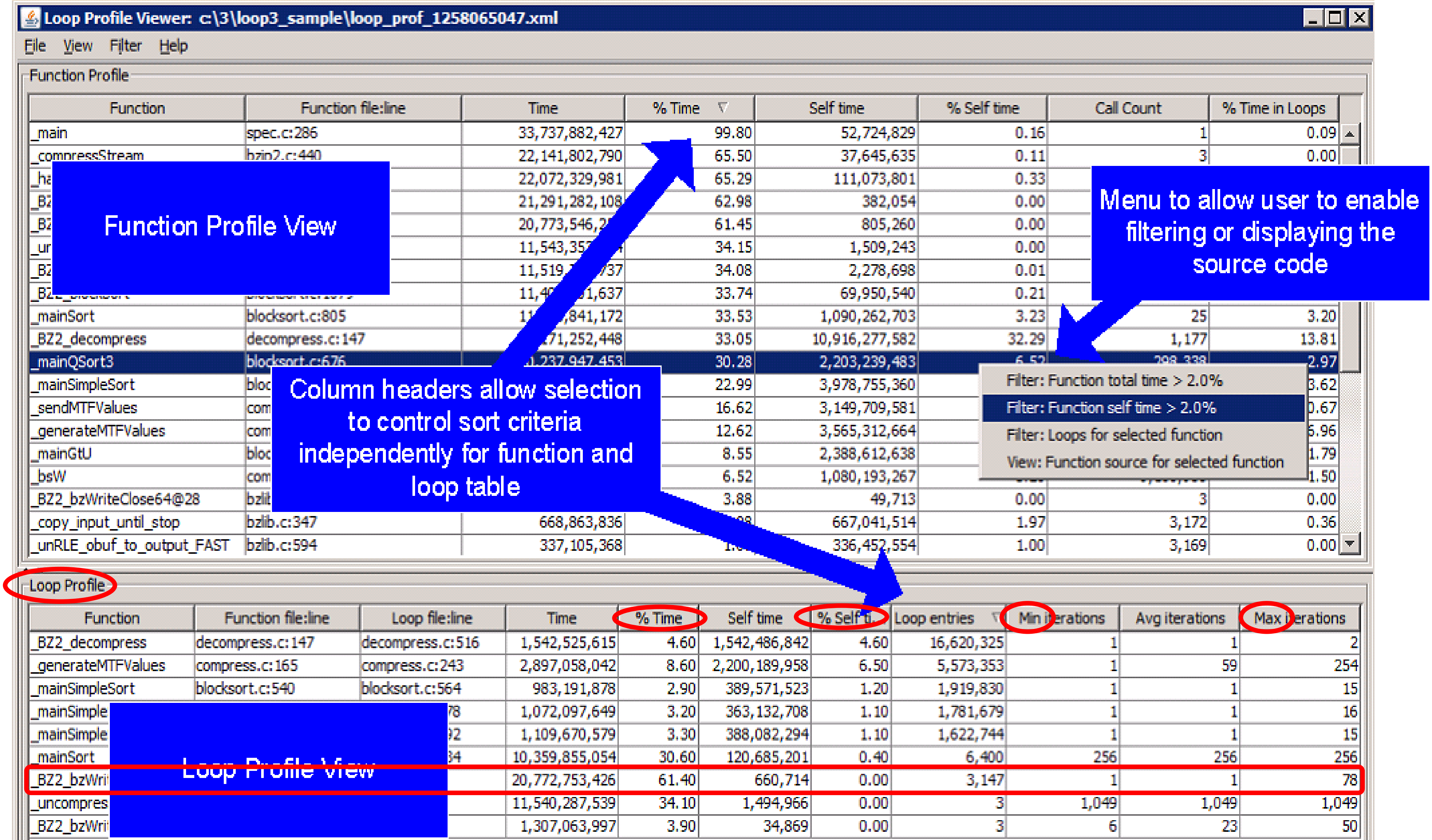
Loop Profile Data Viewer GUI



Loop Profile Data Viewer GUI



Loop Profile Data Viewer GUI



Correctness/Performance Analysis of Parallel code

- **Intel® Inspector XE and thread-reports in VTune™ Amplifier XE are not available (yet) for the Intel® Xeon Phi™ Architecture**
- **So…**
  - Use Intel® Inspector XE on your code with **offload disabled** (on host) to identify correctness errors (e.g., deadlocks, races)
    - Once fixed, then enable offload and continue debugging on the coprocessor
  - Use VTune Amplifier XE's parallel performance analysis tools to find issues on the host by running your program with **offload disabled**
    - Fix everything you can
    - Then study scaling on the coprocessor using lessons from host tuning to further optimize parallel performance
      - Be wary of synchronization when the number of threads becomes more than a handful
      - Also pay attention to load balance.

Agenda

- Porting Checklist
- **Performance Tuning Utilities**
  - Compiler static reports
  - Runtime library report
  - VTune™ Amplifier XE Event-Based Sample collections
- Performance Tuning Hints
- Advanced Performance Tuning

Sample HLO Report

icc -O3 -opt_report -opt_report_phase hlo

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
LOOP INTERCHANGE in loops at line: 15 17
Loopnest permutation ( 1 2 3 ) --> ( 3 2 1 )
…

Loop at line 7 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
Loop at line 15 unrolled and jammed by 4
Loop at line 16 unrolled and jammed by 4
…
```

Compiler Vectorization Report

```
35:     subroutine fd( y )
36:     integer :: i
37:     real, dimension(10), intent(inout) :: y
38:     do i=2,10
39:        y(i) = y(i-1) + 1
40:     end do
41:     end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_:  loop was not vectorized: existence of
vector dependence
```

Compiler Vectorization Report

- **Indicates whether each loop is vectorized**
  - Vectorized ≠ efficient
  - Compiler reports loop vectorized if any version w/vectorization exists
  - At runtime, scalar code may still be executed

- **Indicates reasons for not vectorizing**

- **Line numbers may not be what you'd expected**
  - Inlining
  - Loop distribution, interchange, unrolling, collapsing

When Vectorization Fails …

- Most frequent reason: Data dependencies
  - Simplified: Loop iterations must be independent
- Many other potential reasons
  - Memory alignment issues
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Loop not "countable"
    - E.g. upper bound not a run-time constant
  - Mixed data types (many cases now handled successfully)
  - Non-unit stride between elements
  - Loop body too complex (register pressure)
  - Vectorization seems inefficient
  - Many more … but less likely to occur

Characterization Tools

- **Compiler:**
  - Vectorization report
  - Optimization report
- **Compiler - runtime library reports**
  - OFFLOAD_REPORT, UNIX* time

OFFLOAD_REPORT

- **Dynamic report from compiler offload runtime**
  - Identifies each dynamic instance of an offload by <file, line>
  - Records host and coprocessor execution time for that function
- **Example**

OFFLOAD_REPORT=1
Reports execution times

OFFLOAD_REPORT=2
Adds size of offload data transfers

```
[Offload] [MIC 0] [File]            ma
[Offload] [MIC 0] [Line]            19
[Offload] [MIC 0] [CPU Time]        0.
[Offload] [MIC 0] [MIC Time]        0.

[Offload] [MIC 0] [File]            mic_mhdf.c
[Offload] [MIC 0] [Line]            162
[Offload] [MIC 0] [CPU Time]        0.956117 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]   134219804 (bytes)
[Offload] [MIC 0] [MIC Time]        0.491518 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]   4 (bytes)

[Offload] [MIC 0] [File]            mic_mhdf.c
[Offload] [MIC 0] [Line]            199
[Offload] [MIC 0] [CPU Time]        0.000000 (seconds)
[Offload] [MIC 0] [CPU->MIC Data]   0 (bytes)
[Offload] [MIC 0] [MIC Time]        0.000000 (seconds)
[Offload] [MIC 0] [MIC->CPU Data]   0 (bytes)
```

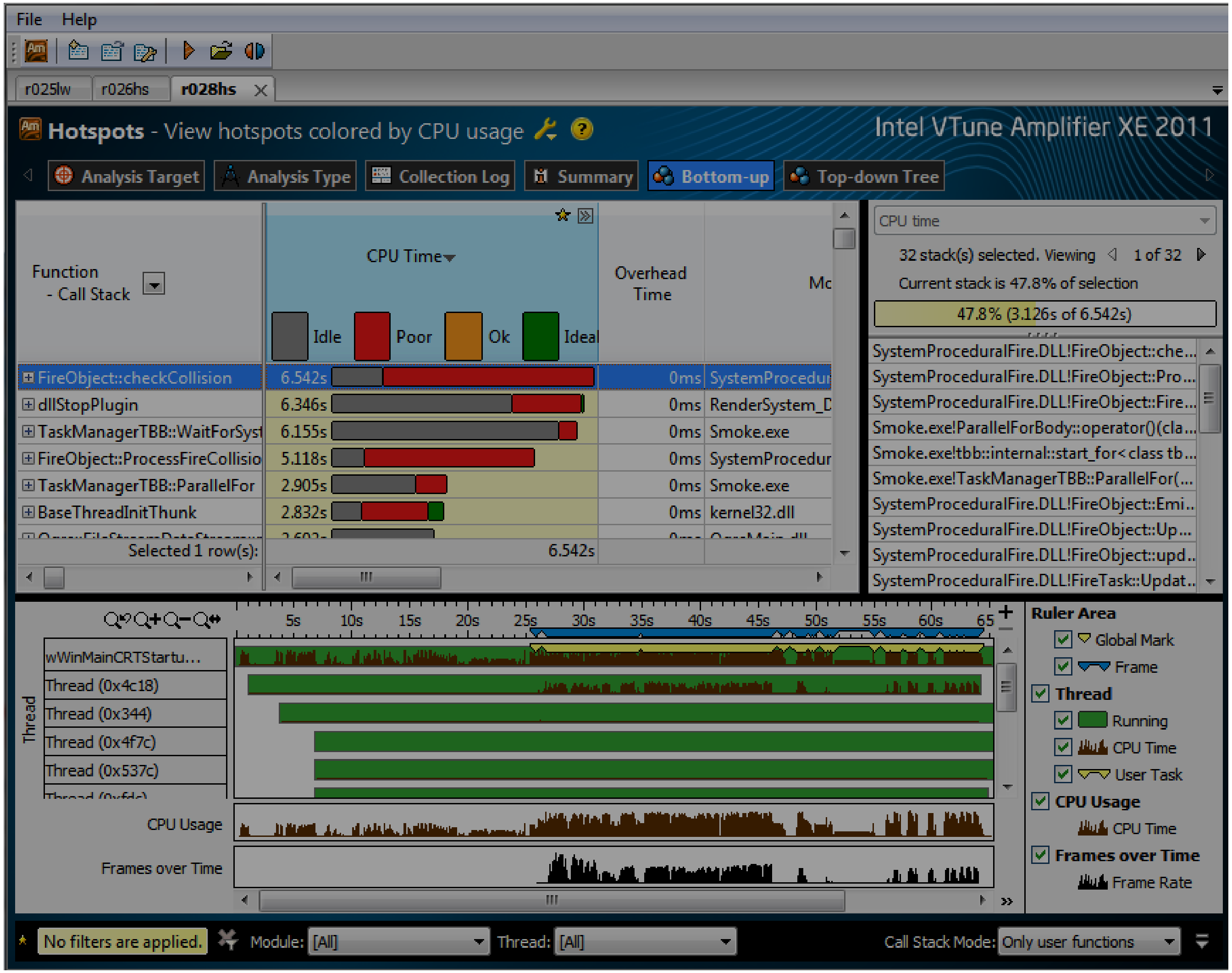
Characterization Tools

- **Compiler:**
  - Vectorization report
  - Optimization report
- **Compiler – runtime library reports**
  - OFFLOAD_REPORT, UNIX* time
- **VTune™ Amplifier XE**
  - Collecting HW performance monitoring data
  - Post-processing HW performance monitoring data
    - VTune Amplifier: hot spots
    - VTune Amplifier: time line
- **Intel® Trace Analyzer and Collector**

Collecting Hardware Performance Data

- **Hardware counters and events**
  - 2 counters in core, most are thread specific
  - 4 outside the core (uncore) that get no thread or core details
  - See PMU documentation for a full list of events
- **Collection**
  - Invoke from VTune Amplifier (or from SEP command line interface)
  - If collecting more than 2 core events, select multi-run for more precise results or the default multiplexed collection, all in one run
  - Uncore events are limited to 4 at a time in a single run
  - Uncore event sampling needs a source of PMU interrupts, e.g. programming cores to CPU_CLK_UNHALTED
- **Output files**
  - VTune Amplifier performance database

VTune™ Amplifier XE



Some useful events and metrics

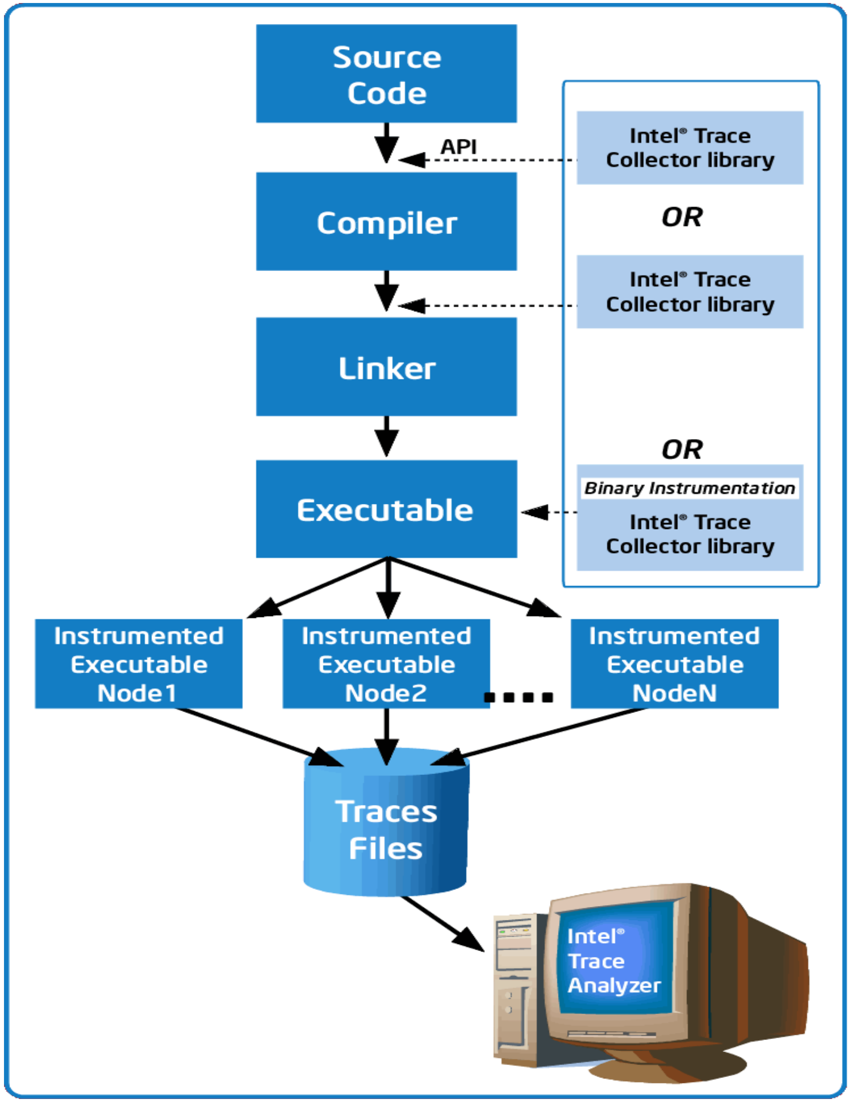
Scenario	Event name(s)
Wall-clock profiling	CPU_CLK_UNHALTED, INSTRUCTIONS_EXECUTED (or EXEC_STAGE_CYCLES)
Main memory bandwidth	L2_DATA_READ_MISS_MEM_FILL, L2_DATA_WRITE_MISS_MEM_FILL
L1 Cache misses	DATA_READ_MISS_OR_WRITE_MISS
TLB misses and page faults	DATA_PAGE_WALK, LONG_DATA_PAGE_WALK, DATA_PAGE_FAULT
Vectorized code execution	VPU_INSTRUCTIONS_EXECUTED, VPU_ELEMENTS_ACTIVE
Various hazards	BRANCHES_MISPREDICTED
Cycles per instruction	CPU_CLK_UNHALTED / INSTRUCTIONS_EXECUTED
Memory Bandwidth (used by all cores at once)	(L2_DATA_READ_MISS_MEM_FILL + L2_DATA_WRITE_MISS_MEM_FILL) * 64 / CPU_CLK_UNHALTED / Frequency

TODO!

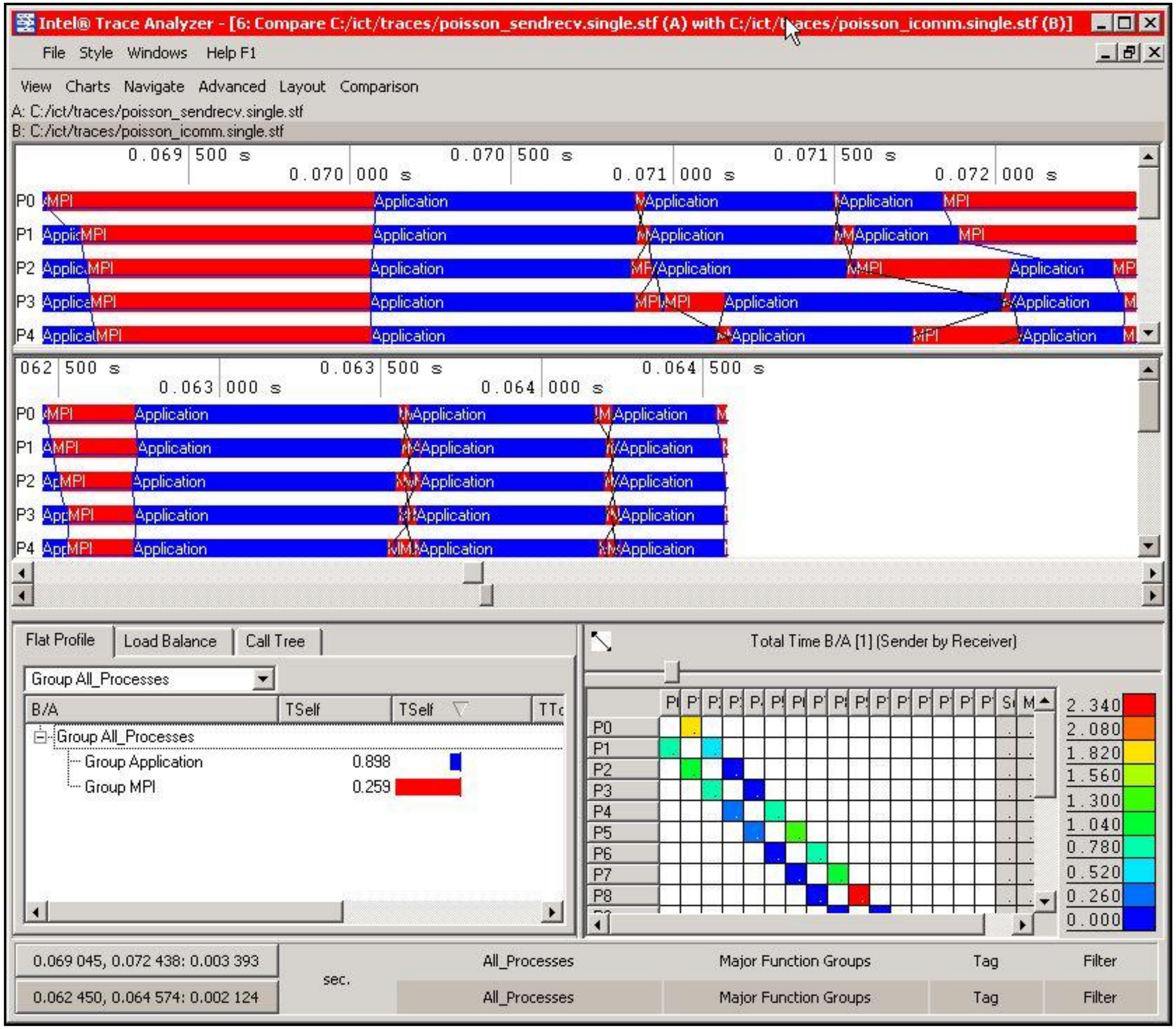
- **ITAC!!!!**

Intel® Trace Analyzer and Collector Overview

- **Intel® Trace Analyzer and Collector helps the developer:**
  - Visualize and understand parallel application behavior
  - Evaluate profiling statistics and load balancing
  - Identify communication hotspots
- **Features**
  - Event-based approach
  - Low overhead
  - Excellent scalability
  - Comparison of multiple profiles
  - Powerful aggregation and filtering functions
  - Fail-safe MPI tracing
  - Provides API to instrument user code
  - MPI correctness checking
  - Idealizer



Intel® Trace Analyzer and Collector



Compare the event timelines of two communication profiles

Blue = computation
Red = communication

Chart showing how the MPI processes interact

ITAC with Intel® Xeon Phi™ Coprocessors

- **Run with –trace flag (without linkage) to create a trace file**
  - MPI+Offload

    ```
    # mpiexec –trace -n 2 ./test
    ```
  - Coprocessor only

    ```
    # mpiexec –trace -n 2 -wdir /tmp
      -host 172.31.1.1 /tmp/test_hello.MIC
    ```
  - Symmetric

    ```
    # mpiexec –trace -n 2 -host michost./test_hello :
      -wdir /tmp -n 2 -host 172.31.1.1
      /tmp/test_hello.MIC
    ```
- **Flag "-trace" will implicitly pre-load libVT.so (which finally calls libmpi.so to execute the MPI call)**
- **Set VT_LOGFILE_FORMAT=stfsingle to create a single trace**

ITAC Compilation Support

- **Compile and link with "–trace" flag**

```
# mpiicc -trace -o test_hello test.c
# mpiicc -trace -mmic -o test_hello.MIC test.c
```

  - Linkage of libVT library
- **Compile with –tcollect flag**

```
# mpiicc -tcollect -o test_hello test.c
# mpiicc -tcollect -mmic -o test_hello.MIC test.c
```

  - Linkage of libVT library
  - Will do a full instrumentation of your code, i.e. All user functions will be visible in the trace file
  - Maximal insight, but also maximal overhead
- **Use the VT API of ITAC to manually instrument your code.**
- **Run as usual Intel® MPI program without "-trace" flag**

```
# mpiexec ...
```

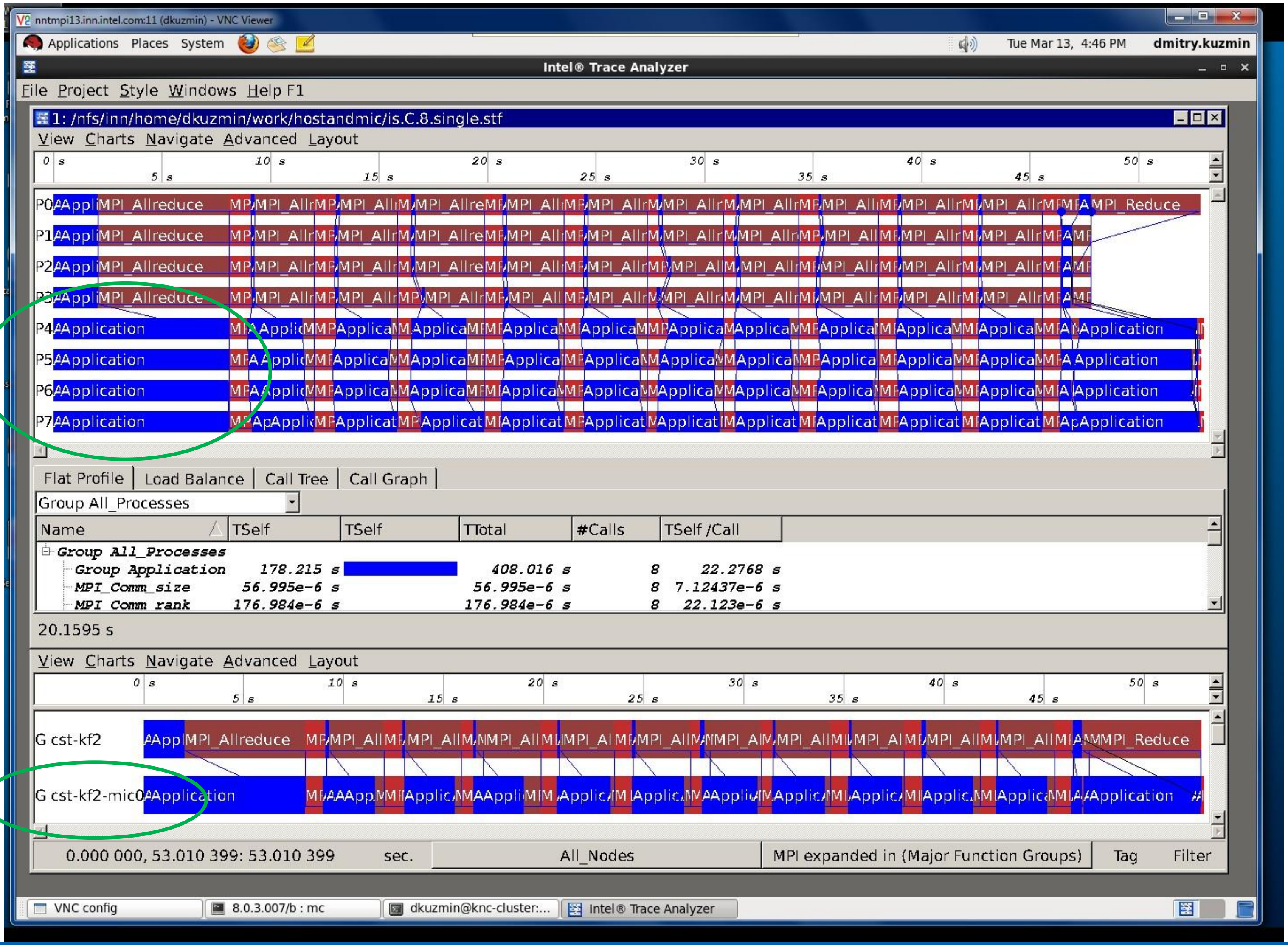
ITAC Analysis

- **Start the ITAC analysis GUI with the trace file (or load it)**

  ```
  # traceanalyzer test_hello.single.stf
  ```

- **Start the analysis, usually by inspection of the Flat Profile (default chart), the Event Timeline, and the Message Profile**
  - Select “Charts->Event Timeline”
  - Select “Charts->Message Profile”
  - Zoom into the Event Timeline
    - Click into it, keep pressed, move to the right, and release the mouse
    - See menu Navigate to get back
  - Right click the “Group MPI->Ungroup MPI”.

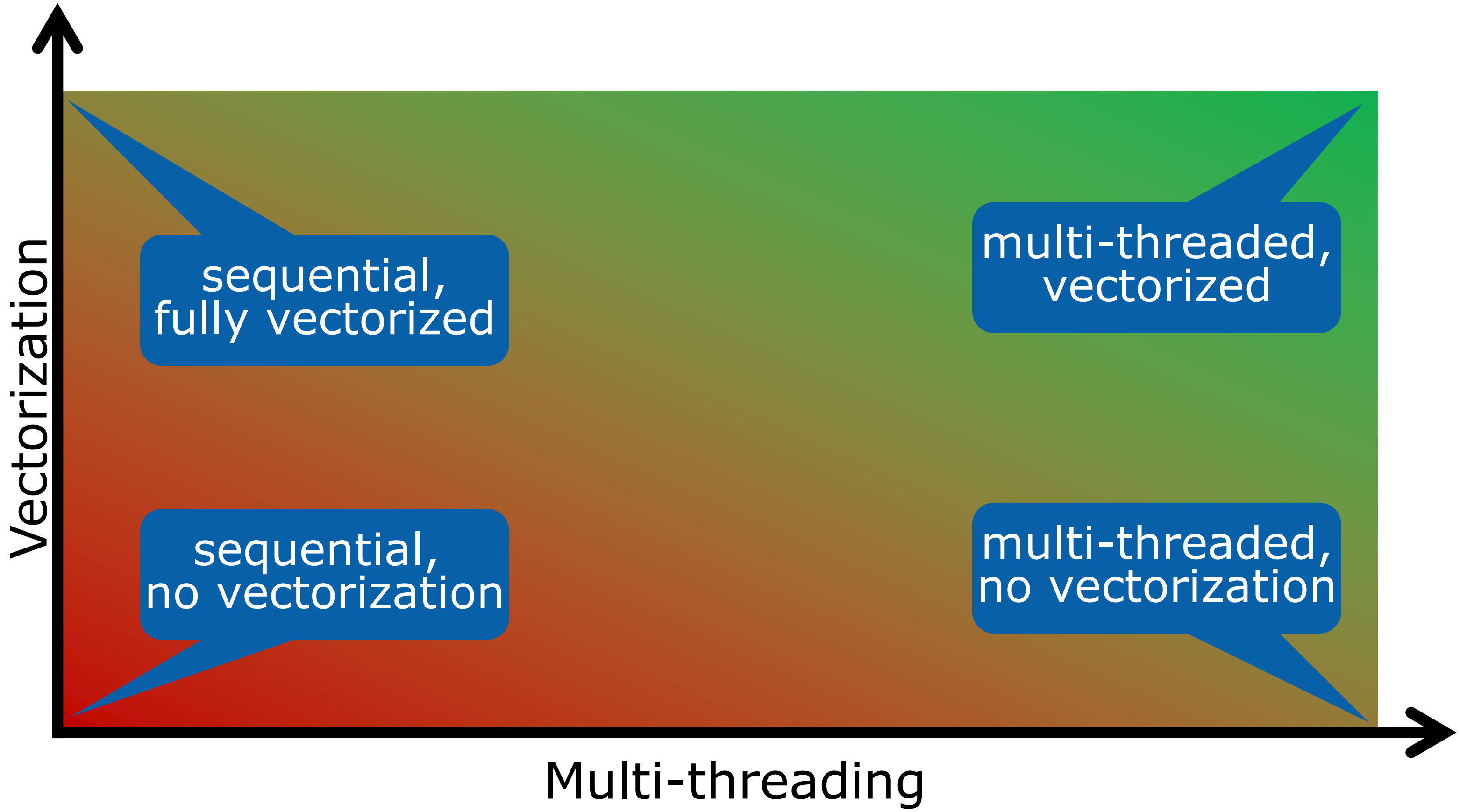
Full ITAC Functionality on Intel® Xeon Phi™



Agenda

- Porting Checklist
- Performance Tuning Utilities
- **Performance Tuning Hints**
  - High Threading And Vectorization are Key
  - Extreme parallelization is required
  - Thread optimization
  - Maximize the vectorizer
  - Architecture-specific hints
- Advanced Performance Tuning

High Threading And Vectorization are Key



- **Performance increasingly depends on both threading and vectorization**
- **Nothing new here: same qualities help host performance**

Extreme Parallelization is required to utilize all the cores

- **Use the appropriate threading model**
  - OpenMP*, Intel® Threading Building Blocks, Intel Cilk™ Plus, POSIX threads
- **Avoid sequential code as much as possible**
  - "Single-threaded" code
  - Avoid atomic operations (e.g. #pragma omp atomic)
  - Avoid locking operations (e.g. #pragma omp critical)
  - Avoid barriers (e.g. #pragma omp barrier)
- **Fuse parallel loops where possible**
  - Manually by merging loop bodies
  - Conceptually by using "nowait" for OpenMP worksharing constructs

Thread Optimization is minimizing overhead and balancing loads

- **Use one MPI rank per coprocessor core, OpenMP* within core**
  - OpenMP synchronization is faster within a core than across cores
- **OMP_NUM_THREADS**
  - Balance MPI and OMP thread parallelism for target
- **`#pragma omp for collapse (n)`**
  - Increase thread-parallelism for utilization, load balance
- **KMP_AFFINITY**
  - Try balanced to avoid OS collision and avoid migration

Thread Optimization

- **Correct affinity essential on Intel® Xeon Phi™ coprocessor**
  - 32 registers of 512 bits make up 2 KB of register file to swap in and out on context switches
  - You want to keep threads on the same (logical) core!
- **KMP_AFFINITY**
  - scatter distribute threads as far apart as possible
  - compact keep threads close to each other
  - balanced mix between scatter and compact
  - proclist specify own set of cores to utilize
- **Example:**
  - export MIC_KMP_AFFINITY=scatter
  - export MIC_KMP_AFFINITY=explicit,proclist=[7,17,19],verbose

Maximize the Vectorizer

- **Unleash the vectorizer by providing context information**
  - #pragma ivdep, #pragma vector always
  - Intel® Cilk Plus vectorization pragmas (#pragma simd)
  - Intel Cilk Plus array notation (a[0:7] = b[1:7] * c[2:7])
  - Avoid aliasing and let the compiler know it
    - "restrict" keyword
    - -ansi-alias
    - -fno-alias
    - Use Fortran ☺

Sample: `simd` Pragmas

```
float sprod(float *a, float *b, int n)
{
  float sum = 0.0f;
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

```
void sprod(float *a, float *b, int n)
{
  float sum = 0.0f;
#pragma simd vectorlength(16) reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

Transformations – Enable Parallelism

- **Loop interchange**
  – Better memory locality raises instruction- and pipeline-parallelism

- **Enable vectorization**
  – Simplify reductions, especially when they appear inside a conditional
    - Turn accumulator into a temp that's declared outside of the loop
    - Accumulate into that temp in the loop
    - Add that temp to the real accumulator outside the loop

  – Avoid constructors in a loop, by extending scope of stack variables to outside a loop, and converting return values to structs

Loop Reordering Example

```
#include <stdio.h>
void mmul(double *a, int lda, double *b, int ldb, double *c, int ldc, int n)
{
   /* &a(i,j) = a + lda * j + i */
   for (int i = 0; i < n; ++i)
Line 6   for (int j = 0; j < n; ++j)
Line 7      for (int k = 0; k < n; ++k)
Line 8          c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
}
```

k=1, k=2, …

```
$ icc -mmic -vec-report3 serialmmul.cc
serialmmul.cc(7): (col. 10) remark: loop was not vectorized: existence of vector dependence.
serialmmul.cc(8): (col. 14) remark: vector dependence: assumed FLOW dependence between c line
   8 and b line 8.
serialmmul.cc(8): (col. 14) remark: vector dependence: assumed ANTI dependence between b line
   8 and c line 8.
         ⋮
serialmmul.cc(6): (col. 7) remark: loop was not vectorized: not inner loop.
```

Problem: all iterations of inner loop (`k`) modify the same element of `c`, and there is no guarantee that `a`, `b`, and `c` do not point to the same memory – classic data race

Loop Reordering Example

```
   #include <stdio.h>
   void mmul(double *a, int lda, double *b, int ldb, double *c, int ldc, int n)
   {
      /* &a(i,j) = a + lda * j + i */
      for (int j = 0; j < n; ++j)
Line 6   for (int k = 0; k < n; ++k)
Line 7      for (int i = 0; i < n; ++i)  // moved to inside loop from outside
Line 8         c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
   }
   $ icc -mmic -vec-report3 interchangemmul.cc
   interchangemmul.cc(7): (col. 10) remark: LOOP WAS VECTORIZED.
   interchangemmul.cc(7): (col. 10) remark: loop skipped: multiversioned.
   interchangemmul.cc(6): (col. 7) remark: loop was not vectorized: not inner loop.
   interchangemmul.cc(5): (col. 4) remark: loop was not vectorized: not inner loop.
```

Each iteration of inner loop (`i`) modifies a different element of `c`, and they are far apart. This implementation looked promising enough for the compiler to go ahead and perform run-time data race checks between c[] and a[]/b[] (“multiversioned” – one version for when c[] overlaps with a[]&b[], one when it does not).

Vectorization Reports - Getting Advice From

```
$ icc -mmic -guide-vec=4 serialmmul.cc
GAP REPORT LOG OPENED ON Wed Mar 30 13:58:35 2011

remark #30761: Add -parallel option if you want the compiler to generate
   recommendations for improving auto-parallelization.

serialmmul.cc(7): remark #30536: (LOOP) Add -fargument-noalias option for better
   type-based disambiguation analysis by the compiler, if appropriate (the
   option will apply for the entire compilation). This will improve
   optimizations such as vectorization for the loop at line 7. [VERIFY] Make
   sure that the semantics of this option is obeyed for the entire compilation.
   [ALTERNATIVE] Another way to get the same effect is to add the "restrict"
   keyword to each pointer-typed formal parameter of the routine "mmul". This
   allows optimizations such as vectorization to be applied to the loop at line
   7. [VERIFY] Make sure that semantics of the "restrict" pointer qualifier is
   satisfied: in the routine, all data accessed through the pointer must not be
   accessed through any other pointer.
Number of advice-messages emitted for this compilation session: 1.
END OF GAP REPORT LOG
$
```

Vectorization Reports - Success by Using Advice From `–guide-vec`

```
   #include <stdio.h>
   void mmul(double * restrict a, int lda, double * restrict b, int ldb,
             double * restrict c, int ldc, int n)
   {
      /* &a(i,j) = a + lda * j + i */
Line 5 for (int i = 0; i < n; ++i)
Line 6     for (int j = 0; j < n; ++j)
Line 7         for (int k = 0; k < n; ++k)
                   c[j * ldc + i] += a[k * lda + i] * b[j * ldb + k];
   }
   $ icc  -mmic -restrict -vec-report3 restructmmul.cc
   restructmmul.cc(5): (col. 4) remark: PERMUTED LOOP WAS VECTORIZED.
   restructmmul.cc(7): (col. 10) remark: loop was not vectorized: not inner loop.
   restructmmul.cc(6): (col. 7) remark: loop was not vectorized: not inner loop.
```

Compiler, realizing that `a`, `b`, and `c` point to different memory, decides it can safely reorder the loops in order to vectorize. Because of `restrict`, the compiler no longer emits a multiversioned loop. This helps lower code size and eliminates the overhead of run-time data race check

Loop and Memory Optimizations

- **Loop trip counts**
  - Improves quality of compiler optimizations such as prefetching, vectorization
  - Can use `#pragma loop_count(n)`, or
    `#pragma loop_count min(n),max(n),avg(n)`
  - Loop profiling not available on Intel® Xeon Phi™ architecture

- **Page sizes**
  - Use libhugetlbfs to force use of 2M pages for non-offloaded data
  - Use the environment variable MIC_USE_2MB_BUFFERS to force runtime to allocate offloaded data into 2MB pages
  - Use mmap to selectively control size

Alignment essential for vectorization

- **Align the data AND tell the compiler**
  - In most cases, static compiler does not have the alignment information of references inside loops, so does extra work to cover misalignment
  - Align the data using alignment attributes, using _mm_malloc, using Fortran option -align array64byte, etc.
  - Tell compiler about alignment using a clause before the vector-loop
    - assume_aligned clause, vector aligned pragma, etc.
- **Mechanisms**

```
__declspace(align(64)) float array[SIZE];
#pragma vector aligned
__assume_aligned(p1,64);
__assume(n1%16==0);
void *__offload_myoSharedAlignedMalloc(size_t size,
size_t alignment);
#pragma offload target(mic) align(64)
```

Advice specific to Intel® Xeon Phi coprocessor

- **Floating point**
  - Use single vs. double precision where possible
  - Use various precision controls where applicable: -imf-*, -[no-]prov-*
  - Rewrite "/const" as "*1/const"
- **Signed vs. unsigned 32b integers**
- **Convert to using 32b vs. 64b ints wherever possible**
  - More elements per SIMD vector
  - Enable vectorization for scatter/gather
  - Enable vectorization for type conversion
- **Avoid scatter/gather where possible**
  - Array of Structures to Structure of Arrays (AoS → SoA)
  - Special-case code to cover unit stride if it's a common occurrence

Agenda

- Porting Checklist
- Performance Tuning Utilities
- Performance Tuning Hints
- **Advanced Performance Tuning**
  - Think about what you expect to achieve
  - Eviction control
  - Inlining control

Think about what you expect to achieve and then measure how close you come

- **Algorithm**
- **Threading**
- **Vectorization and compute-bound limits**
- **Memory**

Algorithm

- **Convince yourself that**
  - the algorithm will thread-scale without serialization,
  - is vectorizable,
  - and fits in memory

- **Consider alternate algorithms that are more suitable**

Threading

- **Degree of parallelism**
  - Assure that fraction of the app that's parallel is very high
  - Assure that the degree of thread parallelism is adequate
  - Check for serialization, e.g. locks
- **OpenMP overheads**
  - Look at VTune™ Amplifier time line for load balance
  - Look at VTune Amplifier hot spots for overhead time in libiomp
- **Tweak number of threads and thread affinity**
  - Find the best value of OMP_NUM_THREADS and KMP_AFFINITY
  - Try 2-3 threads per core
  - Try KMP_AFFINITY=balanced,granularity=fine
- **Threads per core hints (experimental)**
  - -mCG_lrb_num_threads={2,3,4} can bring 5-35% gains
  - Name will change if/when made an official feature

Vectorization and Compute-bound Limits

- **Deep dive on reasons why hot loops don't vectorize profitably**
  - Look at messages, patterns, idioms
- **Compare against compute-bound limit**
- **Check assembly code**
  - Compare path length and generated code with expectations
  - Assure vectorization by checking for vector instructions
- **Check degree of vectorization with PMU data**
  - % vector instructions: VPU_INSTRUCTIONS_EXECUTED/INSTRUCTIONS_EXECUTED
  - Avg. % elements used per vector: VPU_ELEMENTS_ACTIVE/INSTRUCTIONS_EXECUTED

Memory

- Check assembly for gathers/scatters, change data structures or code to avoid them
- Compare against bandwidth limit
- Check L2 and L1 cache miss ratios. Loop interchange, tile and change data structures as necessary to increase locality.
- Check & tune prefetching, particularly for gathers and scatters
- Reason about what the best page size is: 4K or 2M. Use libhugetlbfs or mmap if appropriate. Check TLB miss rates against expected access patterns.

Eviction Control

- **Streaming data trashes cache, doesn't need residency**
  - Mark with #pragma vector nontemporal
  - clevict can be used to evict cache lines sooner and at a higher rate than HW can
  - Intel® Xeon® processor: MOVNTQ
  - Intel Xeon Phi™ coprocessor: clevict0, clevict1
- **-mGLOB_default_function_attrs="clevict_level=N"**

  where N = 0, 1, 2 or 3 (default is 3 on Intel® Xeon Phi™ Architecture)

  0 - do not generate clevict

  1 - generate clevict0, from L1

  2 - generate clevict1, from L2

  3 - generate L1 and L2 clevict

Inlining Control - Pragmas

- **Statement-specific inline pragmas**
  - #pragma inline [recursive] – hint, subject to heuristics
  - #pragma forceinline [recursive] – dictate, whenever possible
  - #pragma noinline – dictate
  - When placed before a C/C++ statement, applies to all calls and statements nested within that statement
  - There are corresponding directives for Fortran

Inlining Controls – Compiler Switches

-[no-]inline-factor=n,

- Specifies % multiplier that should be applied to the following inlining options that define upper limits, i.e. n=200 means multiply upper limits by 2
  - -[no-]inline-min-size=n
  - -[no-]inline-max-size=n
  - -inline-max-per-routine=n
  - -inline-max-per-compile=n

-inline-forceinline

- Specifies that an inlined routine should be inlined whenever the compiler can do so

Questions?