An Introduction to the Intel® Xeon Phi™ Coprocessor

INFIERI-2013 - July 2013

Leo Borges (leonardo.borges@intel.com)
Intel Software & Services Group

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Intel in High-Performance Computing

Dedicated, Renowned Applications Expertise

Large Scale Clusters for Test & Optimization

Tera-Scale Research

Exa-Scale Labs

Broad Software Tools Portfolio

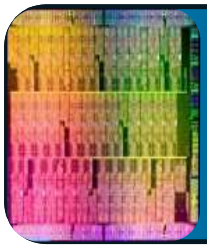
Defined HPC Application Platform

Cluster Ready

Platform Building Blocks

Manufacturing Process Technologies

Leading Performance, Energy Efficient



Many Integrated Core Architecture

A long term commitment to the HPC market segment

HPC Processor Solutions

Multi-Core

Xeon®

General Purpose Architecture

Leadership Per Core Performance

FP/core CAGR via AVX

Multi-Core CAGR

EN	EP	EP 4S	Xeon EX
General purpose perf/watt	Max perf/watt w/ Higher Memory BW / freq and QPI ideal for HPC	Additional compute density	Additional sockets & big memory

Many-Core

Intel® Xeon Phi™ Coprocessor

Trades a "big" IA core for multiple lower performance IA cores resulting in higher performance for a subset of highly parallel applications

Common Intel Environment

Portable code, common tools

Optimization Notice

Highly Parallel Applications Markets, Types, & Hardware


Energy &oil exploration


Digital content creation


Climate modeling & weather prediction

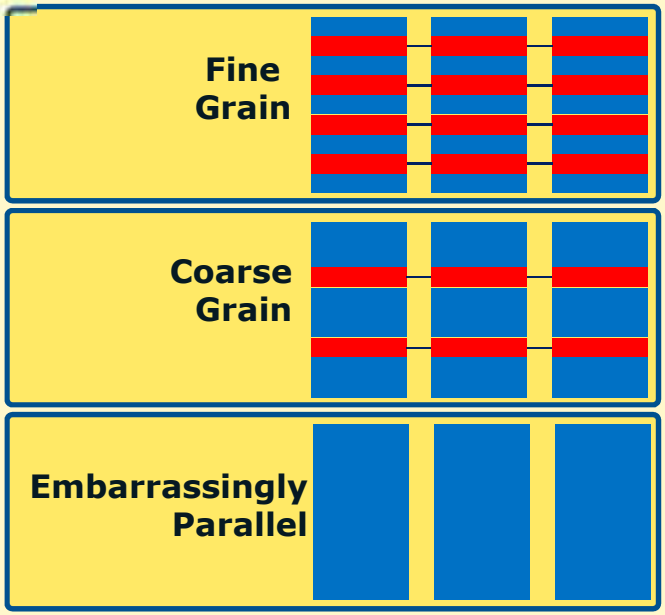

Financial analyses, trading


Medical imaging and biophysics


Computer Aided Design & Manufacturing


C/C++
Fortran

Parallel Application Types



Communication
Compute Process

Highly Parallel Compute Kernels

Black-Scholes Sparse/Dense Matrix Mult ...

FFTs Vector Math LU Factorization

Hardware Options for Parallel Application Types

OR

Intel® MIC Architecture

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

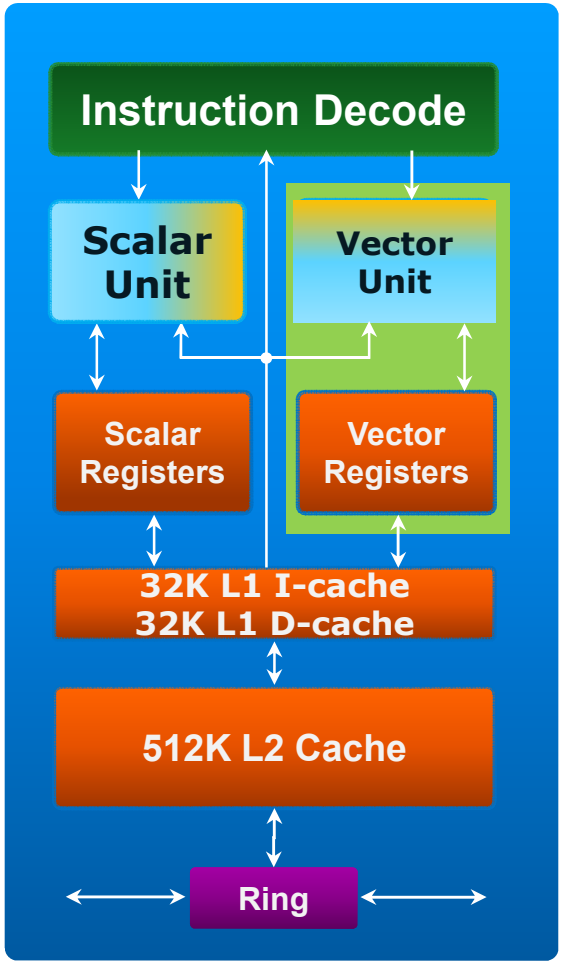
MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit



>50 in-order cores

- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

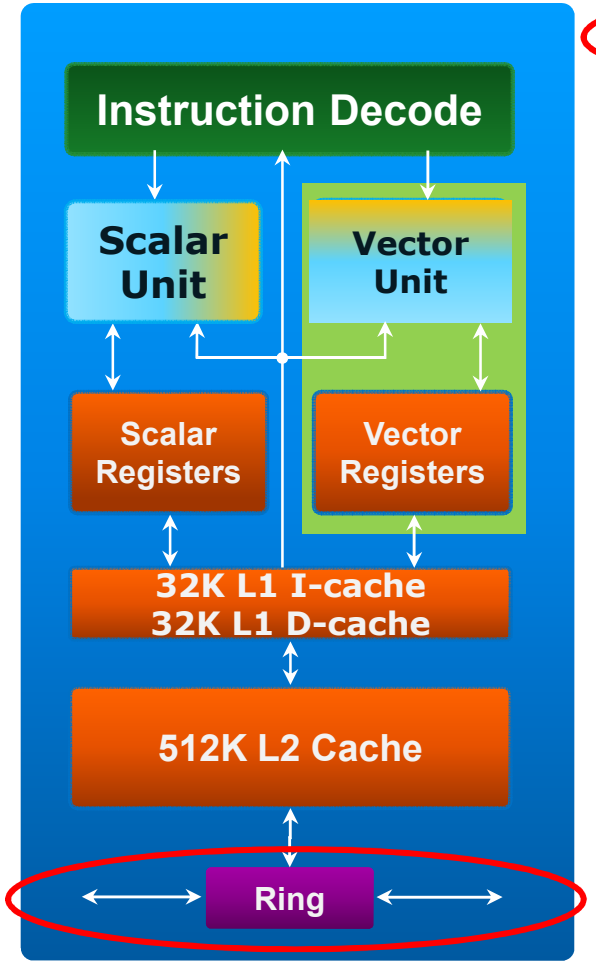
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Optimization Notice

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit



>50 in-order cores

- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

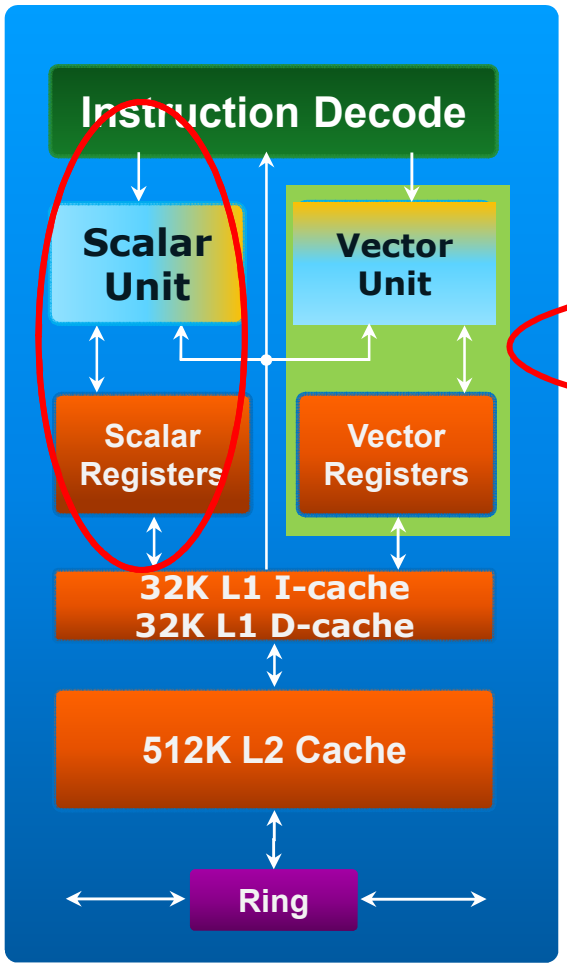
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Optimization Notice

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit



>50 in-order cores

- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

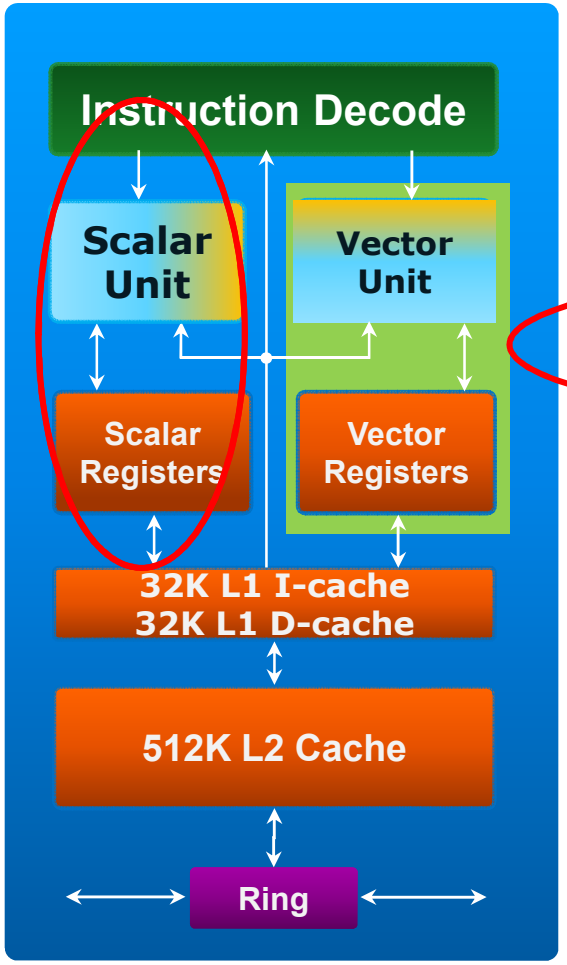
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Optimization Notice

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit



>50 in-order cores

- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

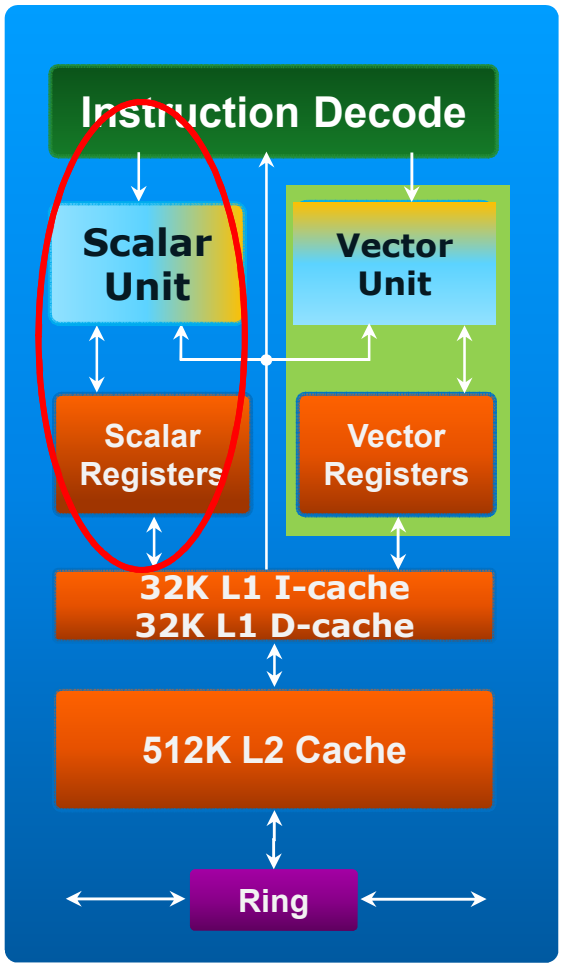
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Optimization Notice

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread execution unit



>50 in-order cores

- Ring interconnect

64-bit addressing

Scalar unit based on Intel® Pentium® processor family

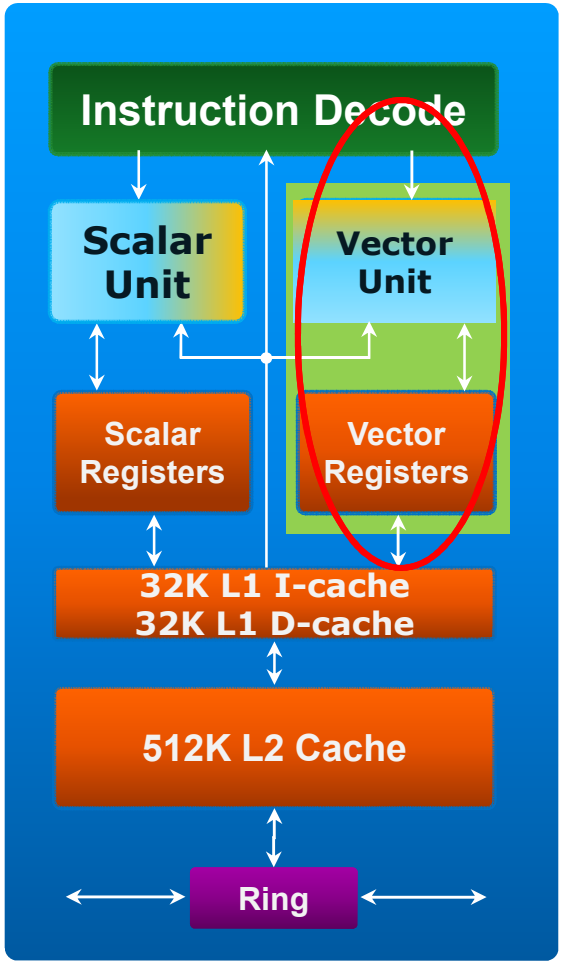
- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Optimization Notice

Each Intel® Xeon Phi™ Coprocessor core is a fully functional multi-thread vector unit



Optimized

- Single and Double precision

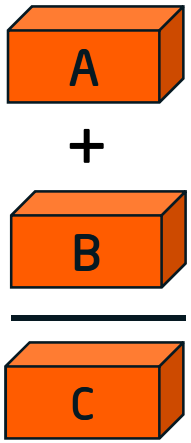
All new vector unit

- 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
- 32 512-bit wide vector registers
  - Hold 16 singles or 8 doubles per register

Fully-coherent L1 and L2 caches

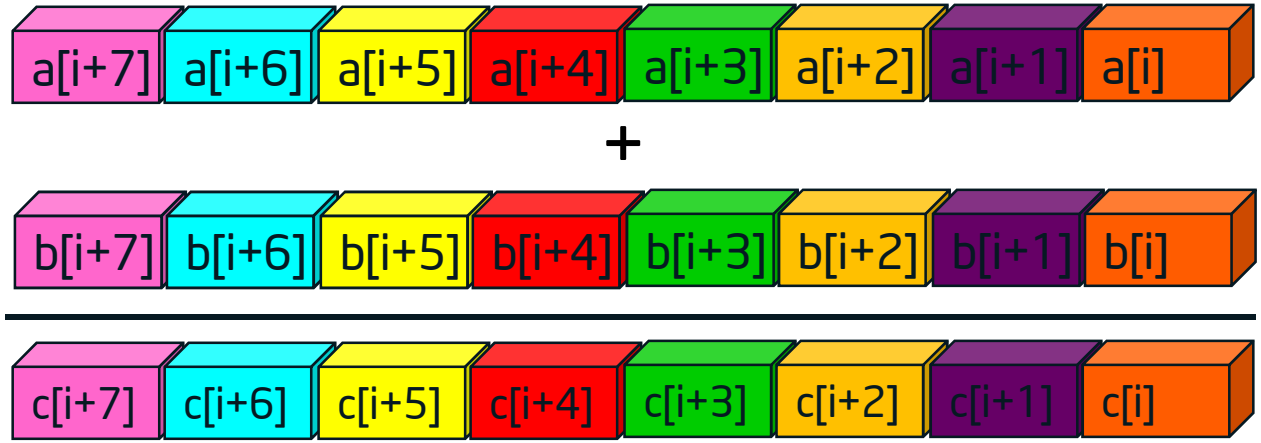
Optimization Notice

Reminder: Vectorization, What is it?
(Graphical View)



Scalar
- One Instruction
- One Mathematical Operation

```
for (i=0;i<=MAX;i++)
   c[i]=a[i]+b[i];
```

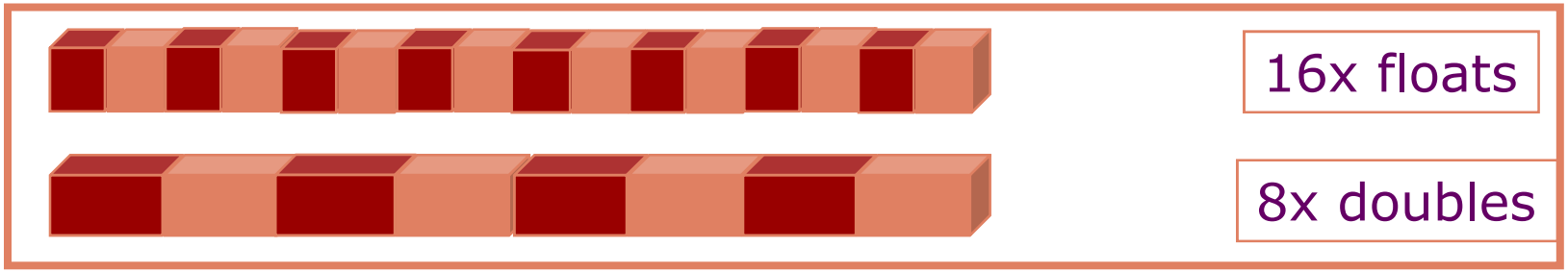


Vector
- One Instruction
- Eight Mathematical Operations[1]

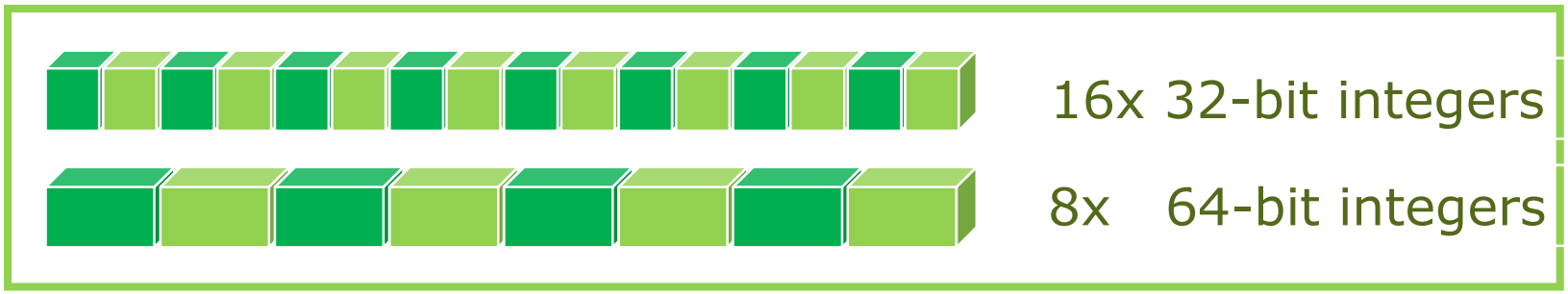
1. Number of operations per instruction varies based on the which SIMD instruction is used and the width of the operands

Data Types for Intel® MIC Architecture

now



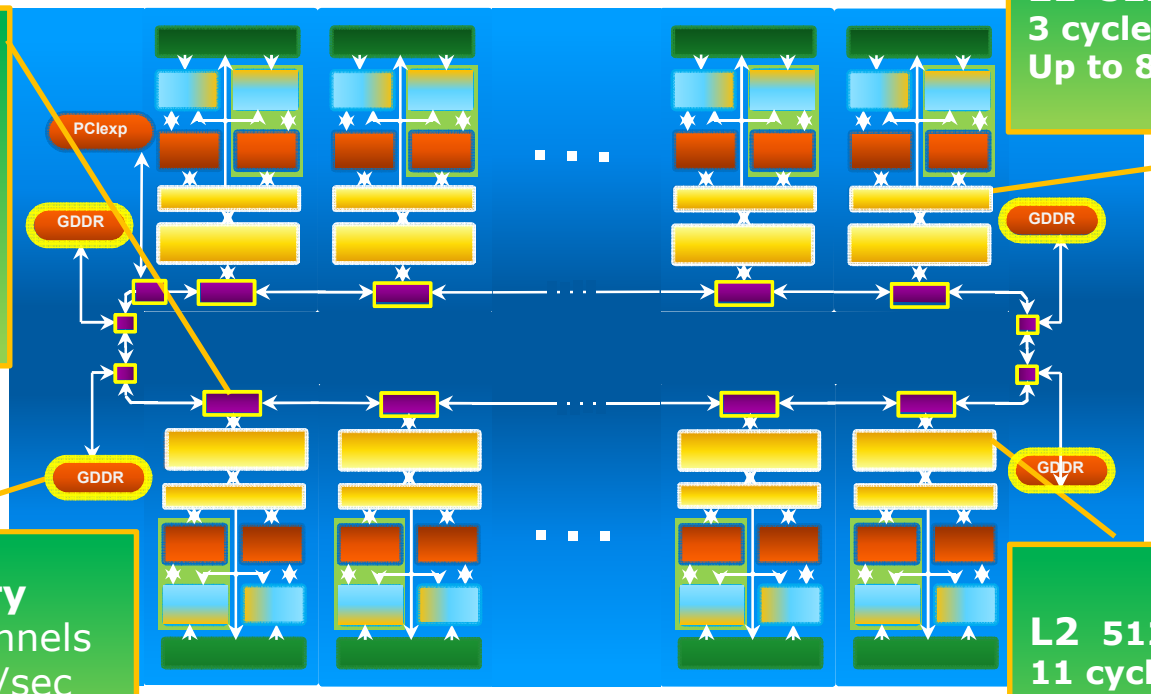
now



Takeaway: Vectorization is very important

Individual cores are tied together via fully coherent caches into a bidirectional ring

Bidirectional ring
115 GB/sec
Distributed Tag Directory (DTD) reduces ring snoop traffic
PCIe port has its own ring stop



L1 32K I- D-cache per core
3 cycle access
Up to 8 concurrent accesses

GDDR5 Memory
16 memory channels
- Up to 5.5 Gb/sec
8 GB 300ns access

L2 512K cache per core
11 cycle best access
Up to 32 concurrent accesses

Takeaway: Parallelization and data placement are important

Optimization Notice

Intel® Xeon Phi™ Coprocessor x100 Family Reference Table

Processor Brand Name	Codename	SKU #	Form Factor, Thermal	Board TDP (Watts)	Max # of Cores	Clock Speed (GHz)	Peak Double Precision (GFLOP)	GDDR5 Memory Speeds (GT/s)	Peak Memory BW	Memory Capacity (GB)	Total Cache (MB)	Enabled Turbo	Turbo Clock Speed (GHz)
Intel® Xeon Phi™ Coprocessor x100	Knights Corner	7120P	PCIe Card, Passively Cooled	300	61	1.238	1208	5.5	352	16	30.5	Y	1.333
		7120X	PCIe Card, No Thermal Solution	300	61	1.238	1208	5.5	352	16	30.5	Y	1.333
		5120D	PCIe Dense Form Factor, No Thermal Solution	245	60	1.053	1011	5.5	352	8	30	N	N/A
		3120P	PCIe Card, Passively Cooled	300	57	1.1	1003	5.0	240	6	28.5	N	N/A
		3120A	PCIe Card, Actively Cooled	300	57	1.1	1003	5.0	240	6	28.5	N	N/A
		Previously Launched and Disclosed											
		5110P*	PCIe Card, Passively Cooled	225	60	1.053	1011	5.0	320	8	30	N	N/A

Please refer to our technical documentation for Silicon stepping information

Optimization Notice

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

More Cores. Wider Vectors. Performance Delivered.

Intel® Parallel Studio XE 2013 and Intel® Cluster Studio XE 2013

More Cores

Multicore Many-core

50+ cores

Wider Vectors

128 Bits

256 Bits

512 Bits

Scaling Performance Efficiently

Serial Performance

Task & Data Parallel Performance

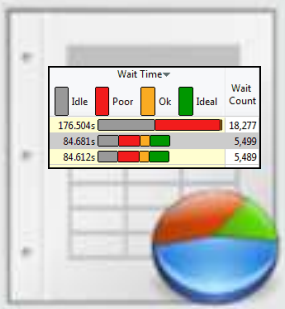
Distributed Performance

- Industry-leading performance from advanced compilers
- Comprehensive libraries
- Parallel programming models
- Insightful analysis tools

Comprehensive set of SW tools

Code Analysis

Advisor XE
VTune Amplifier XE
Inspector XE
Trace Analyzer



Libraries & Compilers

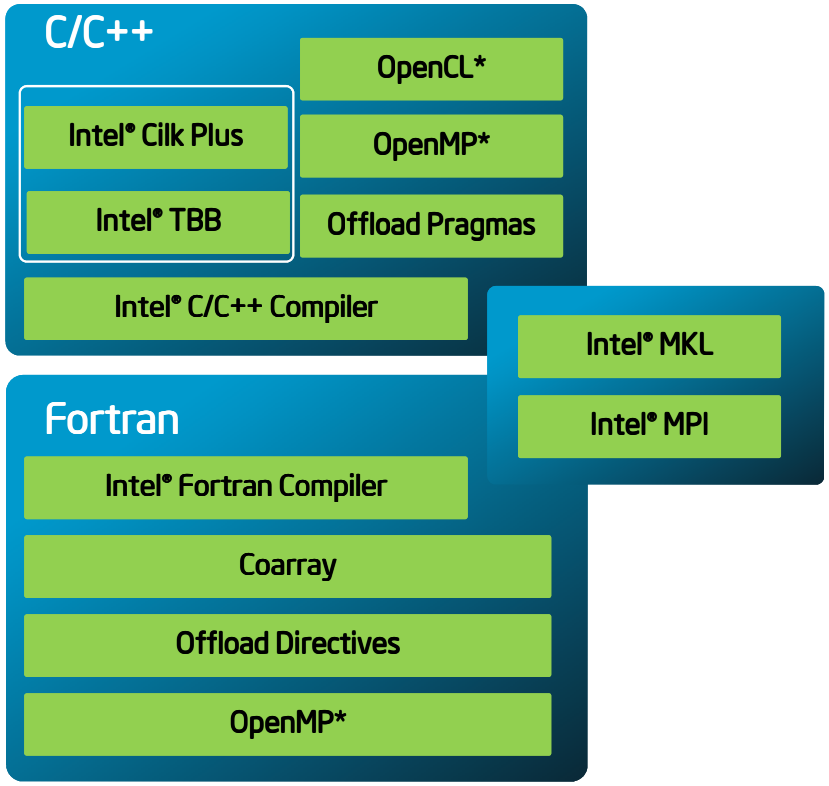
Math Kernel Library
Integrated Performance Primitives
Intel Compilers

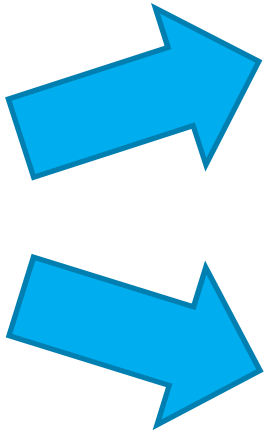
Programming Models

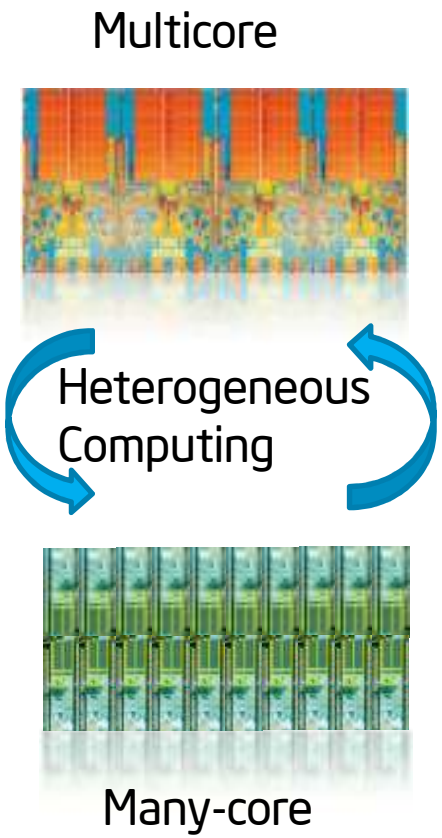
Intel Cilk Plus
Threading Building Blocks
OpenMP
OpenCL
MPI
Offload/Native/MYO

Preserve Your Development Investment

Common Tools and Programming Models for Parallelism







Develop Using Parallel Models that Support Heterogeneous Computing

Optimization Notice

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations:

- Native
- Offload
  - Explicit block data transfer
  - Offloading with Virtual Shared Memory

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Spectrum of Programming & Execution Models

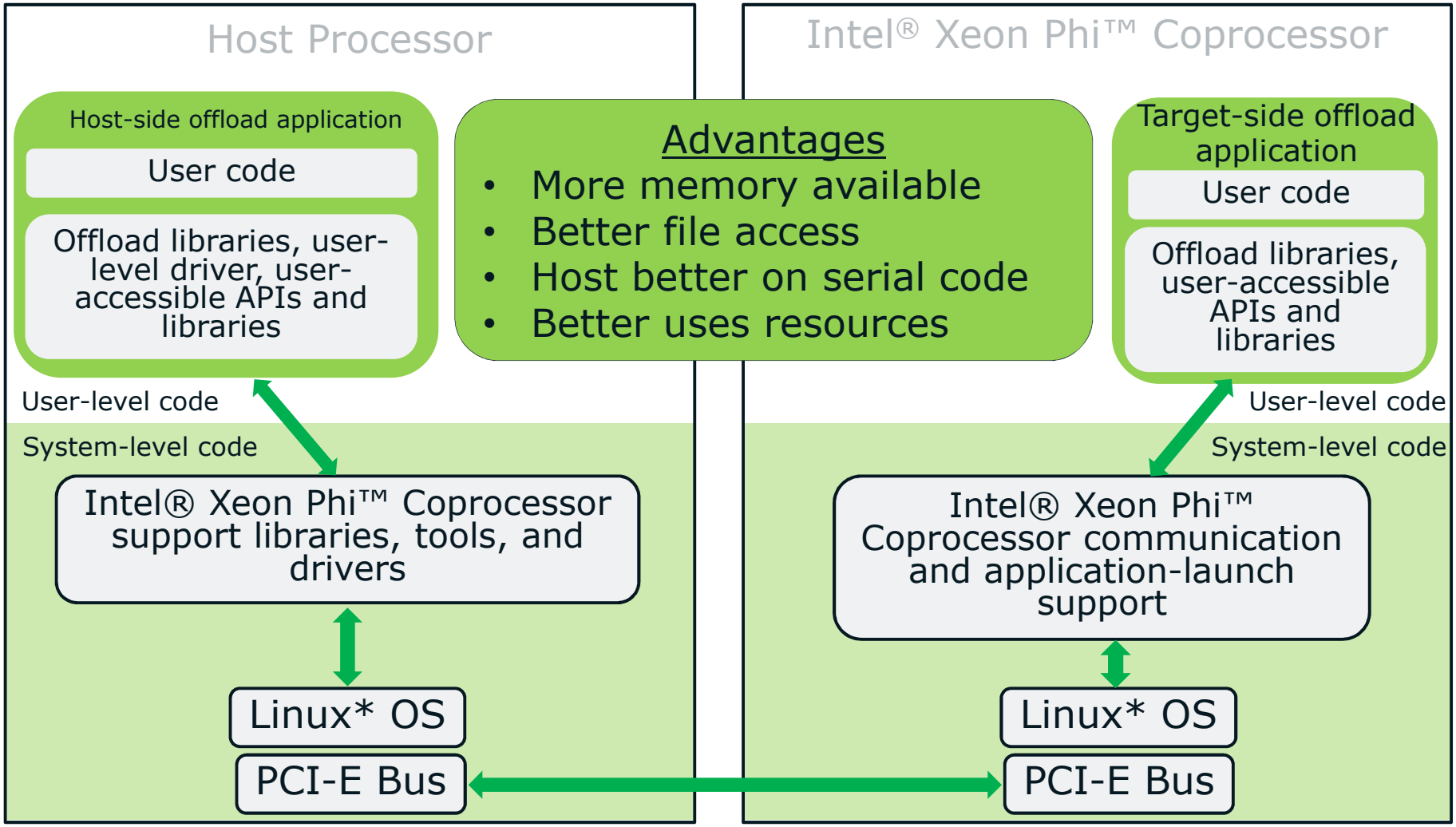
Multicore Centric
(Intel® Xeon® processors)

Many-core Centric
(Intel® Many Integrated Core co-processors)

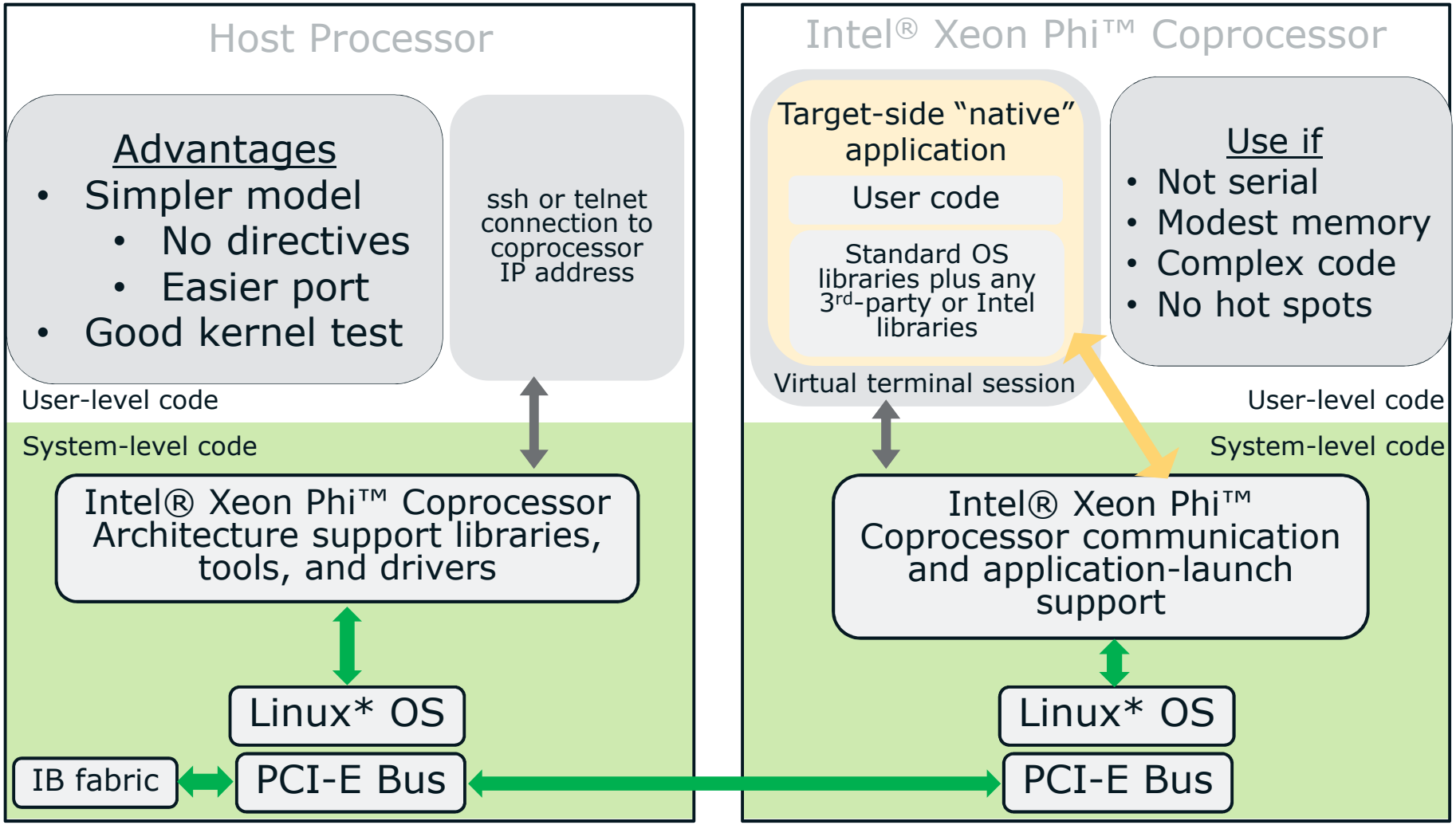
Multi-core-hosted	Offload	Symmetric	Many-core-hosted
General purpose serial and parallel computing	*Codes with highly-parallel phases*	*Codes with balanced needs*	*Highly-parallel codes*
Multicore: *Main( ) Foo( ) MPI_*()*	Multicore: *Main( ) Foo( ) MPI_*()*	Multicore: *Main( ) Foo( ) MPI_*()*	Multicore:
Many-core:	Many-core: *Foo( )*	Many-core: *Main( ) Foo( ) MPI_*()*	Many-core: *Main() Foo( ) MPI_*()*

Range of Models to Meet Application Needs

Intel® Xeon Phi™ Coprocessor runs *either* as an accelerator for offloaded host computation



Or Intel® Xeon Phi™ Coprocessor runs as a native or MPI* compute node via IP or OFED



The Intel® Manycore Platform Software Stack (Intel® MPSS) provides Linux* on the coprocessor

Authenticated users can treat it like another node

```
ssh mic0 top
    Mem: 298016K used, 7578640K free, 0K shrd, 0K buff, 100688K cached
    CPU:  0.0% usr  0.3% sys  0.0% nic 99.6% idle  0.0% io  0.0% irq  0.0% sirq
    Load average: 1.00 1.04 1.01 1/2234 7265
      PID  PPID USER     STAT   VSZ %MEM CPU %CPU COMMAND
     7265  7264 fdkew    R     7060  0.0  14  0.3 top
       43     2 root     SW       0  0.0  13  0.0 [ksoftirqd/13]
     5748     1 root     S     119m  1.5 226  0.0 ./sep_mic_server3.8
     5670     1 micuser  S    97872  1.2   0  0.0 /bin/coi_daemon --coiuser=micuser
     7261  5667 root     S    25744  0.3   6  0.0 sshd: fdkew [priv]
     7263  7261 fdkew    S    25744  0.3 241  0.0 sshd: fdkew@notty
     5667     1 root     S    21084  0.2   5  0.0 /sbin/sshd
     5757     1 root     S     6940  0.0  18  0.0 /sbin/getty -L -l /bin/noauth 1152
        1     0 root     S     6936  0.0  10  0.0 init
     7264  7263 fdkew    S     6936  0.0   6  0.0 sh -c top
```

Intel MPSS supplies a virtual FS and native execution

```
sudo scp /opt/intel/composerxe/lib/mic/libiomp5.so root@mic0:/lib64
scp a.out mic0:/tmp
ssh mic0 /tmp/a.out my-args
```

Add `-mmic` to compiles to create native programs

```
icc -O3 -g -mmic -o nativeMIC myNativeProgram.c
```


Optimization Notice

Alternately, use the offload capabilities of Intel® Composer XE to access coprocessor

Offload directives in source code trigger Intel Composer to compile objects for both host and coprocessor

```
#pragma offload target(mic) inout(A:length(2000))          C/C++
!DIR$ OFFLOAD TARGET(MIC) INOUT(A: LENGTH(2000))           Fortran
```

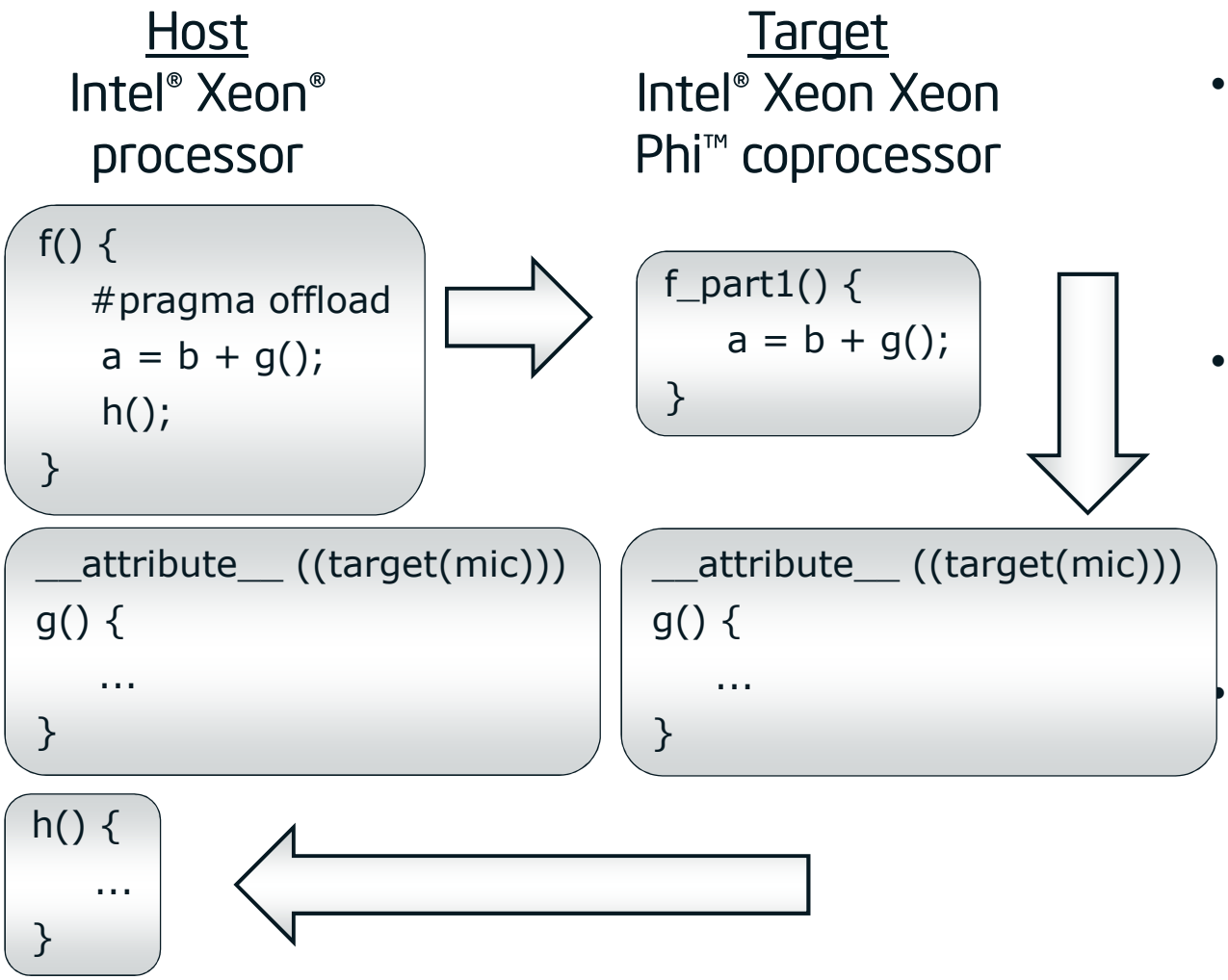
When the program is executed and a coprocessor is available, the offload code will run on that target

- Required data can be transferred explicitly for each offload
- Or use Virtual Shared Memory (_Cilk_shared) to match virtual addresses between host and target coprocessor

Offload blocks initiate coprocessor computation and can be synchronous or asynchronous

```
#pragma offload_transfer target(mic) in(a: length(2000)) signal(a)
!DIR$ OFFLOAD_TRANSFER TARGET(MIC) IN(A: LENGTH(2000)) SIGNAL(A)
_Cilk_spawn _Cilk_offload asynch-func()
```

Offload directives are independent of function boundaries



Execution

- If at first offload the target is available, the target program is loaded
- At each offload if the target is available, statement is run on target, else it is run on the host
- At program termination the target program is unloaded

Example: Compiler Assisted Offload

- Offload section of code to the coprocessor.

```
float pi = 0.0f;
#pragma offload target(mic)
#pragma omp parallel for reduction(+:pi)
for (i=0; i<count; i++) {
   float t = (float)((i+0.5f)/count);
   pi += 4.0f/(1.0f+t*t);
}
pi /= count;
```

- Offload any function call to the coprocessor.

```
#pragma offload target(mic) \
    in(transa, transb, N, alpha, beta) \
    in(A:length(matrix_elements)) \
    in(B:length(matrix_elements)) \
    in(C:length(matrix_elements)) \
    out(C:length(matrix_elements) alloc_if(0))
    {
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
              &beta, C, &N);
    }
```

Example: Compiler Assisted Offload

- An example in Fortran:

```
!DEC$ ATTRIBUTES OFFLOAD : TARGET( MIC ) :: SGEMM
!DEC$ OMP OFFLOAD TARGET( MIC ) &
!DEC$ IN( TRANSA, TRANSB, M, N, K, ALPHA, BETA, LDA, LDB, LDC ), &
!DEC$ IN( A: LENGTH( NCOLA * LDA )), &
!DEC$ IN( B: LENGTH( NCOLB * LDB )), &
!DEC$ INOUT( C: LENGTH( N * LDC ))
CALL SGEMM( TRANSA, TRANSB, M, N, K, ALPHA, &
                A, LDA, B, LDB BETA, C, LDC )
```

Example – share work between coprocessor and host using OpenMP*

```
omp_set_nested(1);
#pragma omp parallel private(ip)
{
#pragma omp sections
 {
#pragma omp section
/*      use pointer to copy back only part of potential array,
       to avoid overwriting host */
#pragma offload target(mic) in(xp) in(yp) in(zp) out(ppot:length(np1))
#pragma omp parallel for private(ip)
  for (i=0;i<np1;i++)  {
    ppot[i] = threed_int(x0,xn,y0,yn,z0,zn,nx,ny,nz,xp[i],yp[i],zp[i]);
  }
#pragma omp section
#pragma omp parallel for private(ip)
  for (i=0;i<np2;i++)  {
    pot[i+np1] =
  threed_int(x0,xn,y0,yn,z0,zn,nx,ny,nz,xp[i+np1],yp[i+np1],zp[i+np1]);
  }
 }
}
```

Top level, runs on host
Runs on coprocessor
Runs on host

Pragmas and directives mark data and code to be offloaded and executed

	C/C++ Syntax
Offload pragma	`#pragma offload <clauses> <statement>` Allow next statement to execute on coprocessor or host CPU
Variable/function offload properties	`__attribute__((target(mic)))` Compile function for, or allocate variable on, both host CPU and coprocessor

Entire blocks of data/code defs	`#pragma offload_attribute(push, target(mic))`
	
	`#pragma offload_attribute(pop)`
	Mark entire files or large blocks of code to compile for both

	Fortran Syntax
Offload directive	`!dir$ omp offload <clauses> <statement>` Execute OpenMP* parallel block on coprocessor
	`!dir$ offload <clauses> <statement>` Execute next statement or function on coproc.
Variable/function offload properties	`!dir$ attributes offload:<mic> :: <ret-name> OR <var1,var2,…>` Compile function or variable for CPU and coprocessor
Entire code blocks	`!dir$ offload begin <clauses>` `!dir$ end offload`

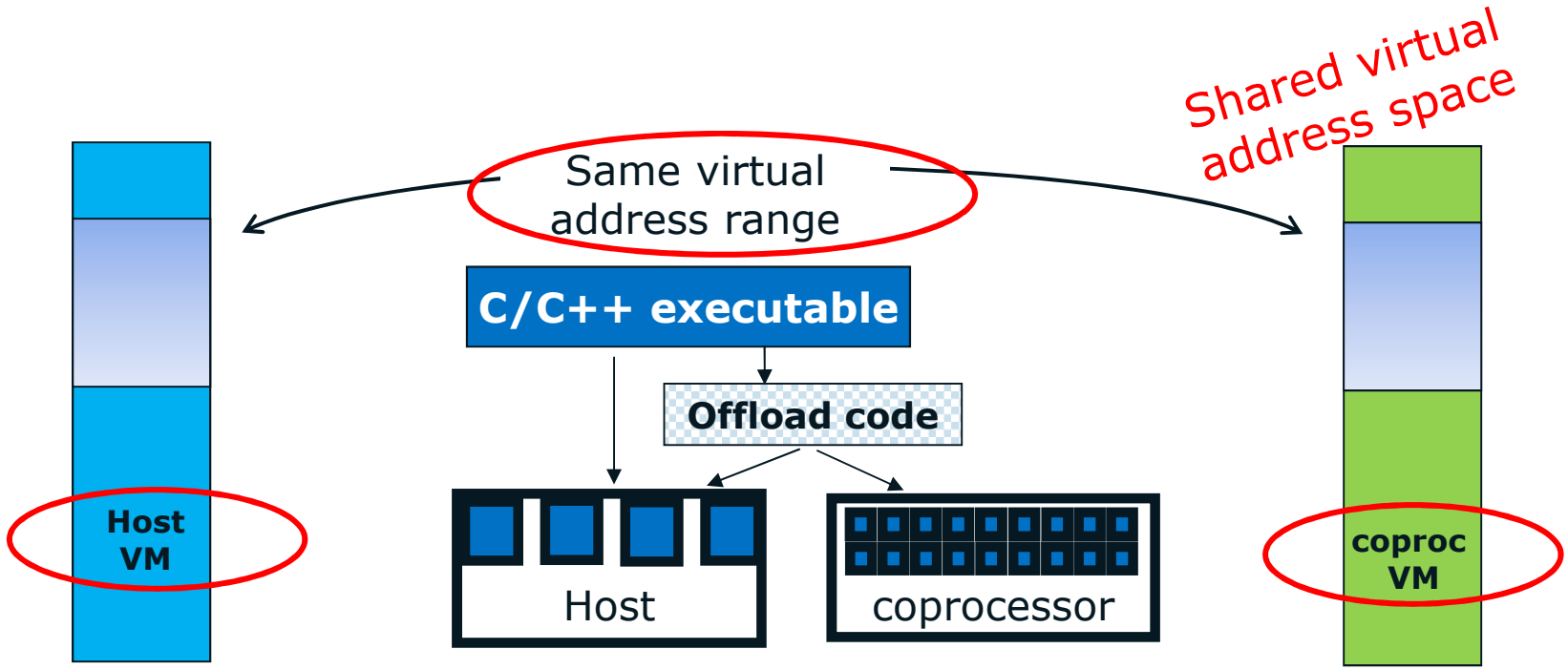
Options on offloads can control data copying and manage coprocessor dynamic allocation

Clauses	Syntax	Semantics
Multiple coprocessors	`target(mic[:unit] )`	Select specific coprocessors
Conditional offload	`if (condition) / manadatory`	Select coprocessor or host compute
Inputs	`in(var-list modifiers`$_{opt}$`)`	Copy from host to coprocessor
Outputs	`out(var-list modifiers`$_{opt}$`)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers`$_{opt}$`)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers`$_{opt}$`)`	Data is local to target
Modifiers		
Specify copy length	`length(N)`	Copy N elements of pointer's type
Coprocessor memory allocation	`alloc_if ( bool )`	Allocate coprocessor space on this offload (default: TRUE)
Coprocessor memory release	`free_if ( bool )`	Free coprocessor space at the end of this offload (default: TRUE)
Control target data alignment	`align ( N bytes )`	Specify minimum memory alignment on coprocessor
Array partial allocation & variable relocation	`alloc ( array-slice )` `into ( var-expr )`	Enables partial array allocation and data copy into other vars & ranges

To handle more complex data structures on the coprocessor, use Virtual Shared Memory

An identical range of virtual addresses is reserved on both host an coprocessor: changes are shared at offload points, allowing:

- Seamless sharing of complex data structures, including linked lists
- Elimination of manual data marshaling and shared array management
- Freer use of new C++ features and standard classes



Example: Virtual Shared Memory

- Shared between host and Xeon Phi

```
// Shared variable declaration
_Cilk_shared T in1[SIZE];
_Cilk_shared T in2[SIZE];
_Cilk_shared T res[SIZE];

_Cilk_shared void compute_sum()
{
    int i;
    for (i=0; i<SIZE; i++) {
        res[i] = in1[i] + in2[i];
    }
}

(...)

// Call compute sum on Target
_Cilk_offload compute_sum();
```

Virtual Shared Memory uses special allocation to manage data sharing at offload boundaries

Declare virtual shared data using _Cilk_shared allocation specifier

Allocate virtual dynamic shared data using these special functions:

```
_Offload_shared_malloc(), _Offload_shared_aligned_malloc(),
_Offload_shared_free(), _Offload_shared_aligned_free()
```

Shared data copying occurs automatically around offload sections

- Memory is only synchronized on entry to or exit from an offload call
- Only modified data blocks are transferred between host and coprocessor

Allows transfer of C++ objects

- Pointers are transportable when they point to "shared" data addresses

Well-known methods can be used to synchronize access to shared data and prevent data races within offloaded code

- E.g., locks, critical sections, etc.

This model is integrated with the Intel® Cilk™ Plus parallel extensions

Note: Not supported on Fortran - available for C/C++ only

Data sharing between host and coprocessor can be enabled using this Intel® Cilk™ Plus syntax

What	Syntax
Function	`int _Cilk_shared f(int x){ return x+1; }` Code emitted for host and target; may be called from either side
Global	`_Cilk_shared int x = 0;` Datum is visible on both sides
File/Function static	`static _Cilk_shared int x;` Datum visible on both sides, only to code within the file/function
Class	`class _Cilk_shared x {…};` Class methods, members and operators available on both sides
Pointer to shared data	`int _Cilk_shared *p;` *p* is local (not shared), can point to shared data
A shared pointer	`int *_Cilk_shared p;` *p* is shared; should only point at shared data

Entire blocks of code	`#pragma offload_attribute( push, _Cilk_shared)`
	
	`#pragma offload_attribute(pop)`
	Mark entire files or blocks of code _Cilk_shared using this pragma

Intel® Cilk™ Plus syntax can also specify the offloading of computation to the coprocessor

Feature	Example
Offloading a function call	`x = _Cilk_offload func(y);` `func` executes on coprocessor if possible
	`x = _Cilk_offload_to (card_num) func(y);` `func` ***must*** execute on specified coprocessor or an error occurs
Offloading asynchronously	`x = _Cilk_spawn _Cilk_offload func(y);` `func` executes on coprocessor; continuation available for stealing

Offloading a parallel for-loop	`_Cilk_offload _Cilk_for(i=0; i<N; i++){`
	`  a[i] = b[i] + c[i];`
	`}`
	Loop executes in parallel on coprocessor.
	The loop is implicitly "un-inlined" as a function call.

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Options for Thread Parallelism

Intel® Math Kernel Library

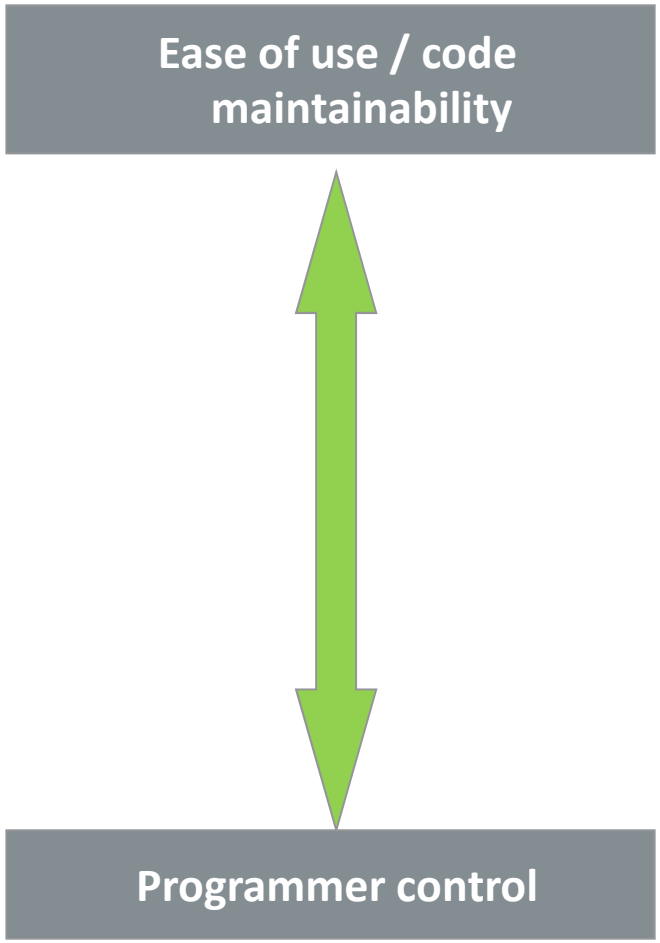
OpenMP*

Intel® Threading Building Blocks

Intel® Cilk™ Plus

OpenCL*

Pthreads* and other threading libraries



Choice of unified programming to target Intel® Xeon® and Intel® Xeon Phi™ Architecture!

Optimization Notice

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism: OpenMP

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

OpenMP* on the Coprocessor

- The basics work just like on the host CPU
  - For both native and offload models
  - Need to specify -openmp
- There are 4 hardware thread contexts per core
  - Need <u>at least</u> 2 x ncore threads for good performance
    - For all except the most memory-bound workloads
    - Often, 3x or 4x (number of available cores) is best
    - Very different from hyperthreading on the host!
    - -opt-threads-per-core=n advises compiler how many threads to optimize for
  - If you don't saturate all available threads, be sure to set KMP_AFFINITY to control thread distribution

OpenMP defaults

- OMP_NUM_THREADS defaults to
  - 1 x ncore for host (or 2x if hyperthreading enabled)
  - 4 x ncore for native coprocessor applications
  - 4 x (ncore-1) for offload applications
    - one core is reserved for offload daemons and OS

  - Defaults may be changed via environment variables or via API calls on either the host or the coprocessor

Target OpenMP environment (offload)

Use target-specific APIs to set for coprocessor target only, e.g.

omp_set_num_threads_target() (called from host)

omp_set_nested_target() etc

- Protect with #ifdef __INTEL_OFFLOAD, undefined with –no-offload
- Fortran: USE MIC_LIB and OMP_LIB C: #include <offload.h>

Or define MIC – specific versions of env vars using

MIC_ENV_PREFIX=MIC (no underscore)

- Values on MIC **<u>no longer default to values on host</u>**
- Set values specific to MIC using

export MIC_OMP_NUM_THREADS=120 (all cards)

export MIC_2_OMP_NUM_THREADS=180 for card #2, etc

export MIC_3_ENV="OMP_NUM_THREADS=240|KMP_AFFINITY=balanced"

Stack Sizes for Coprocessor

For the main thread, (thread 0), default stack limit is 12 MB

- In offloaded functions, stack is used for local or automatic arrays and compiler temporaries
- To increase limit, export MIC_STACKSIZE (e.g. =100M)
  - default unit is K (Kbytes)
- For native apps, use ulimit –s (default units are Kbytes)

For worker threads: default stack size is 4 MB

- Space only needed for those local variables or automatic arrays or compiler temporaries for which each thread has a private copy
- To increase limit, export OMP_STACKSIZE=10M (or as needed)
- Or use dynamic allocation (may be less efficient)

Typical error message if stack limits exceeded:

offload error: process on the device 0 was terminated by SEGFAULT

Thread Affinity Interface

Allows OpenMP threads to be bound to physical or logical cores

- export environment variable KMP_AFFINITY=
  - physical — use all physical cores before assigning threads to other logical cores (other hardware thread contexts)
  - compact — assign threads to consecutive h/w contexts on same physical core (eg to benefit from shared cache)
  - scatter — assign consecutive threads to different physical cores (eg to maximize access to memory)
  - balanced — blend of compact & scatter (currently only available for Intel® MIC Architecture)
- Helps optimize access to memory or cache
- Particularly important if all available h/w threads not used
  - else some physical cores may be idle while others run multiple threads
- See compiler documentation for (much) more detail

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism: TBB

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Intel® Threading Building Blocks

Widely used C++ template library for parallelism

C++ Library for parallel programming

- Takes care of managing multitasking

Runtime library

- Scalability to available number of threads

Cross-platform

- Windows, Linux, Mac OS* and others

http://threadingbuildingblocks.org

Intel® Threading Building Blocks

Generic Parallel Algorithms
Efficient scalable way to exploit the power of multi-core without having to start from scratch

Concurrent Containers
Common idioms for concurrent access
- a scalable alternative serial container with a lock around it

TBB Flow Graph

Task scheduler
The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency

Thread Local Storage
Scalable implementation of thread-local data that supports infinite number of TLS

Synchronization Primitives
User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables

Miscellaneous
Thread-safe timers

Threads
OS API wrappers

Memory Allocation
Per-thread scalable memory manager and false-sharing free allocators

parallel_for usage example

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
using namespace tbb;

class ChangeArray{
    int* array;
public:
    ChangeArray(int* a): array(a) {}
    void operator()(const blocked_range<int>& r) const {
        for (int i = r.begin(); i != r.end(); i++) {
            Foo (array[i]);
        }
    }
};

int main (){
    int a[n];
    // initialize array here…
    parallel_for (blocked_range<int>(0, n), ChangeArray(a));
    return 0;
}
```

ChangeArray class defines a for-loop body for parallel_for

blocked_range – TBB template representing 1D iteration space

As usual with C++ function objects the main work is done inside operator()

A call to a template function parallel_for<Range, Body>: with arguments
Range → blocked_range
Body → ChangeArray

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism: MKL

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

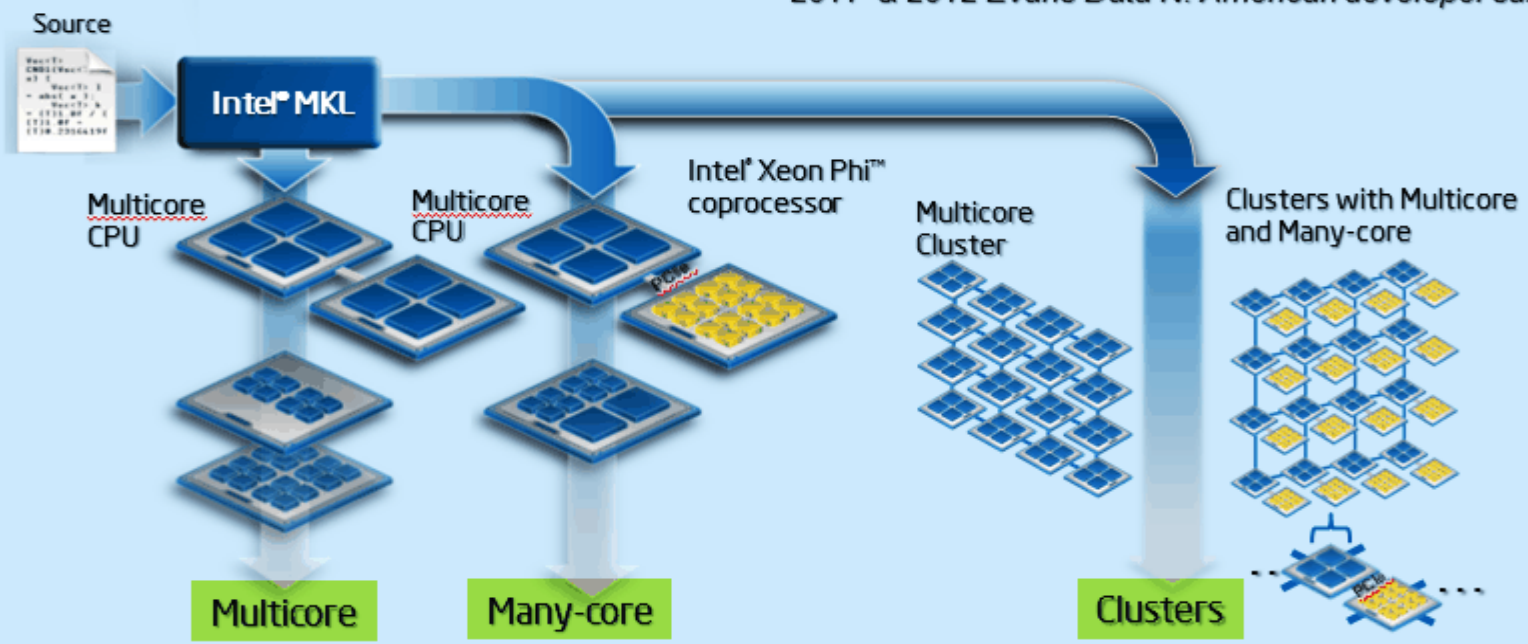
Profiling: Intel® Trace Analyzer and Collector

Conclusions

Intel® MKL is industry's leading math library *

Linear Algebra	Fast Fourier Transforms	Vector Math	Vector Random Number Generators	Summary Statistics	Data Fitting
•BLAS •LAPACK •Sparse solvers •ScaLAPACK	•Multidimensional (up to 7D) •FFTW interfaces •Cluster FFT	•Trigonometric •Hyperbolic •Exponential, Logarithmic •Power / Root •Rounding	•Congruential •Recursive •Wichmann-Hill •Mersenne Twister •Sobol •Neiderreiter •Non-deterministic	•Kurtosis •Variation coefficient •Quantiles, order statistics •Min/max •Variance-covariance •...	•Splines •Interpolation •Cell search

** 2011 & 2012 Evans Data N. American developer surveys*

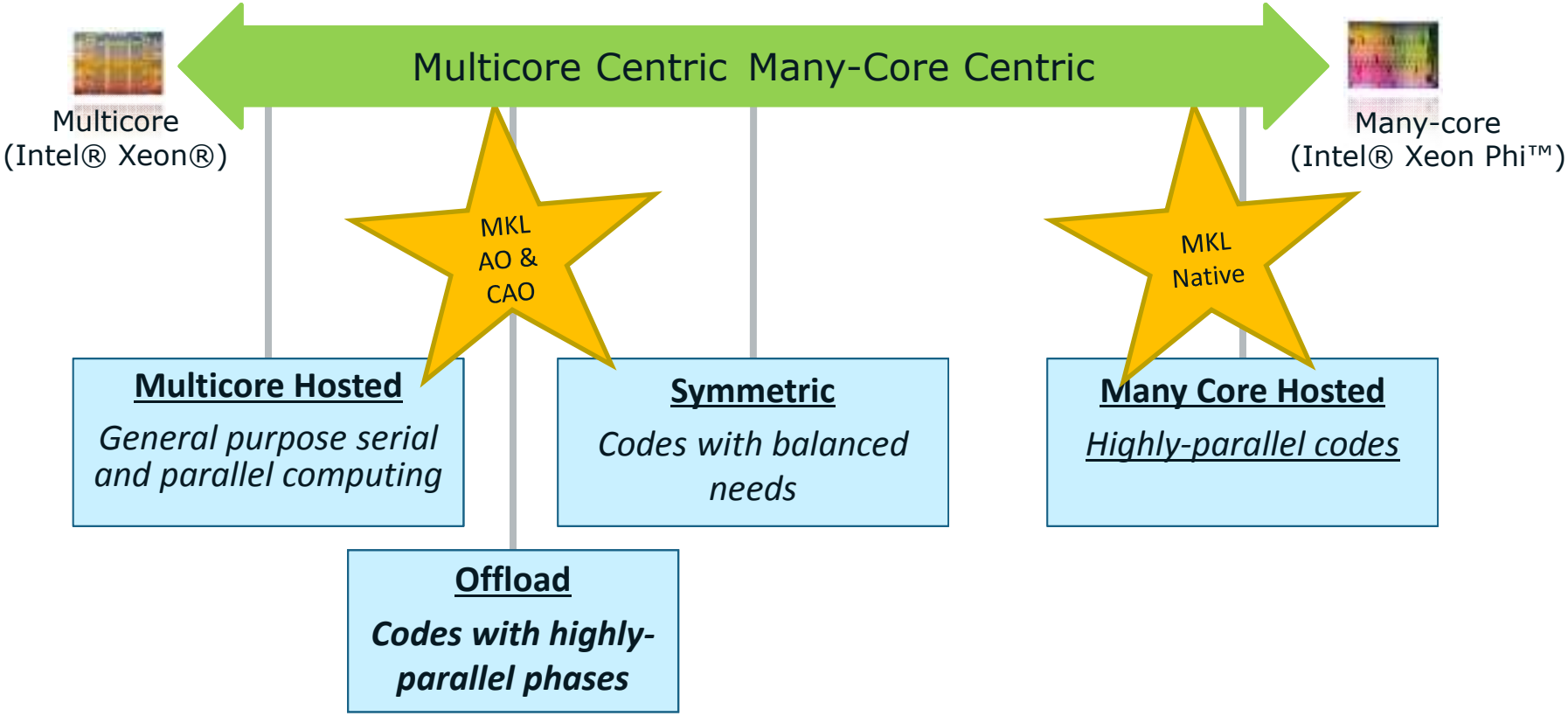


Optimization Notice

MKL Usage Models on Intel® Xeon Phi™ Coprocessor

- Automatic Offload
  - No code changes required
  - Automatically uses both host and target
  - Transparent data transfer and execution management
- Compiler Assisted Offload
  - Explicit controls of data transfer and remote execution using compiler offload pragmas/directives
  - Can be used together with Automatic Offload
- Native Execution
  - Uses the coprocessors as independent nodes
  - Input data is copied to targets in advance

MKL Execution Models



Work Division Control in MKL Automatic Offload

Examples	Notes
mkl_mic_set_Workdivision(MKL_TARGET_MIC, 0, 0.5)	Offload 50% of computation only to the 1st card.

Examples	Notes
MKL_MIC_0_WORKDIVISION=0.5	Offload 50% of computation only to the 1st card.

How to Use MKL with Compiler Assisted Offload

- The same way you would offload any function call to the coprocessor.

- An example in C:

```
#pragma offload target(mic) \
    in(transa, transb, N, alpha, beta) \
    in(A:length(matrix_elements)) \
    in(B:length(matrix_elements)) \
    in(C:length(matrix_elements)) \
    out(C:length(matrix_elements) alloc_if(0))
    {
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
              &beta, C, &N);
    }
```

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

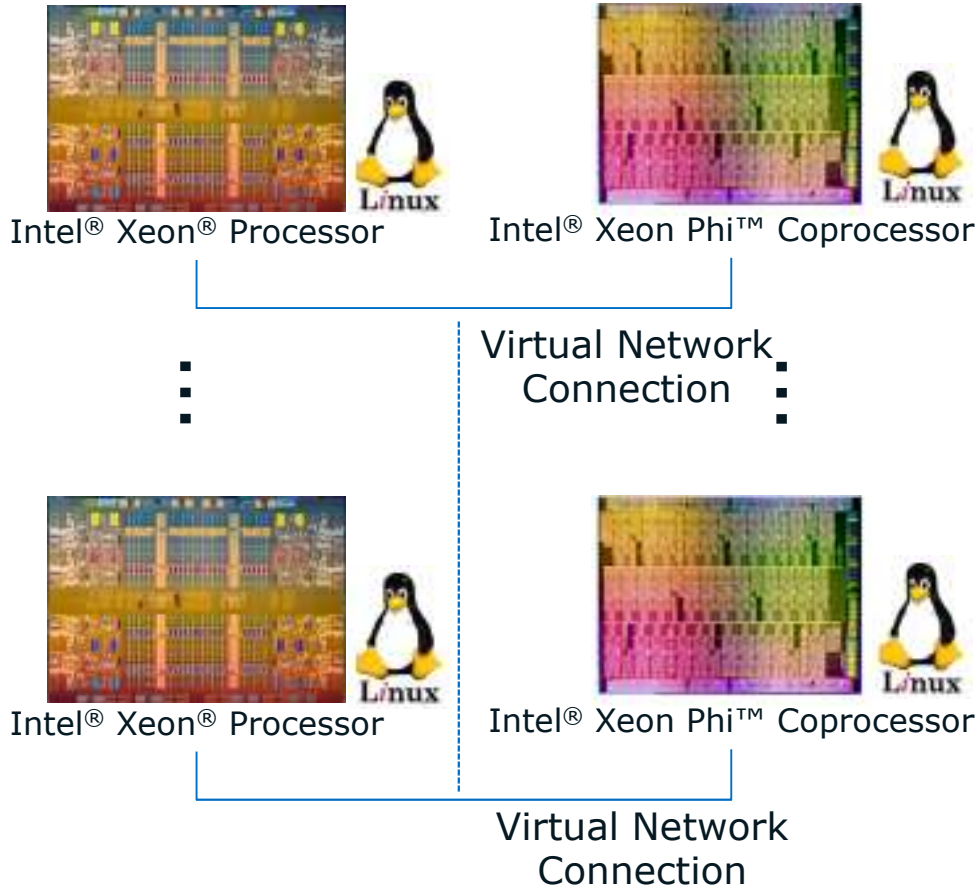
MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Intel® Xeon Phi™ Coprocessor Becomes a Network Node



Intel® Xeon Phi™ Architecture + Linux enables IP addressability

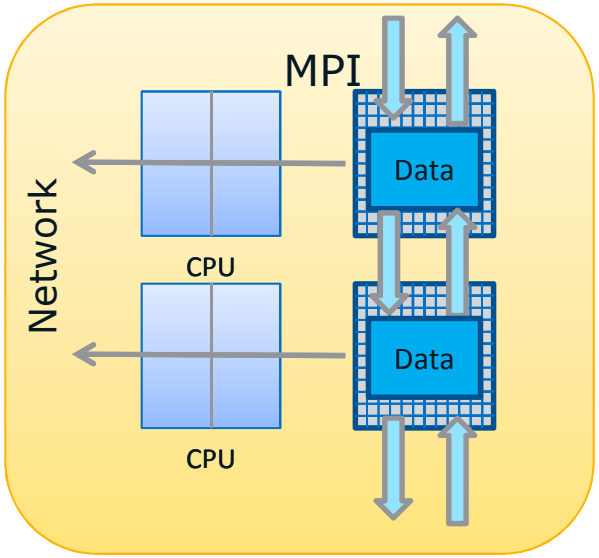
Optimization Notice

Coprocessor only Programming Model

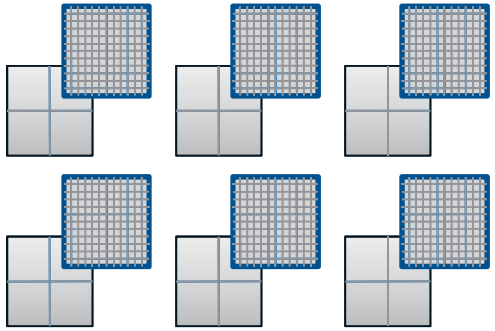
MPI ranks on Intel® Xeon Phi™ coprocessor (only)

All messages into/out of the coprocessors

Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads used directly within MPI processes



Homogenous network of many-core CPUs



Build Intel Xeon Phi coprocessor binary using the Intel® compiler

Upload the binary to the Intel Xeon Phi coprocessor

Run instances of the MPI application on Intel Xeon Phi coprocessor nodes

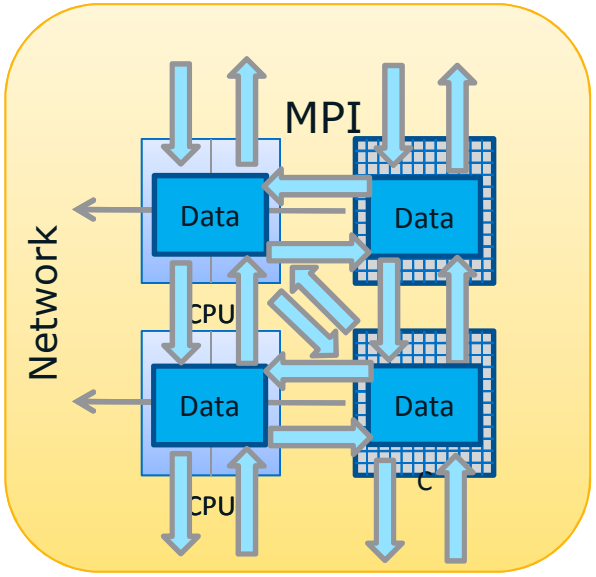
Optimization Notice

Symmetric Programming Model

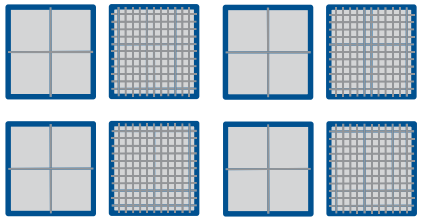
MPI ranks on Intel® Xeon Phi™ Architecture and host CPUs

Messages to/from any core

Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* used directly within MPI processes



Heterogeneous network of homogeneous CPUs



Build binaries by using the resp. compilers targeting Intel 64 and Intel Xeon Phi Architecture

Upload the binary to the Intel Xeon Phi coprocessor

Run instances of the MPI application on different mixed nodes

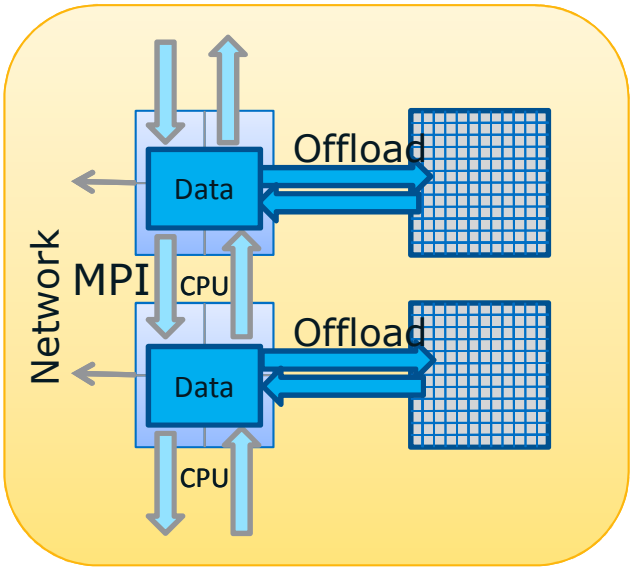
MPI+Offload Programming Model

MPI ranks on Intel® Xeon® processors (only)

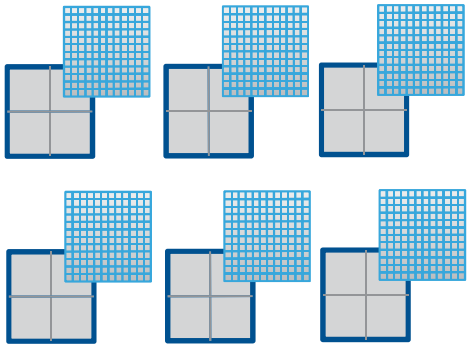
All messages into/out of host CPUs

Offload models used to accelerate MPI ranks

Intel® Cilk™ Plus, OpenMP*, Intel® Threading Building Blocks, Pthreads* within Intel® Xeon Phi™ coprocessor



Homogenous network of heterogeneous nodes



Build Intel® 64 executable with included offload by using the Intel compiler

Run instances of the MPI application on the host, offloading code onto coprocessor

Advantages of more cores and wider SIMD for certain applications

Optimization Notice

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

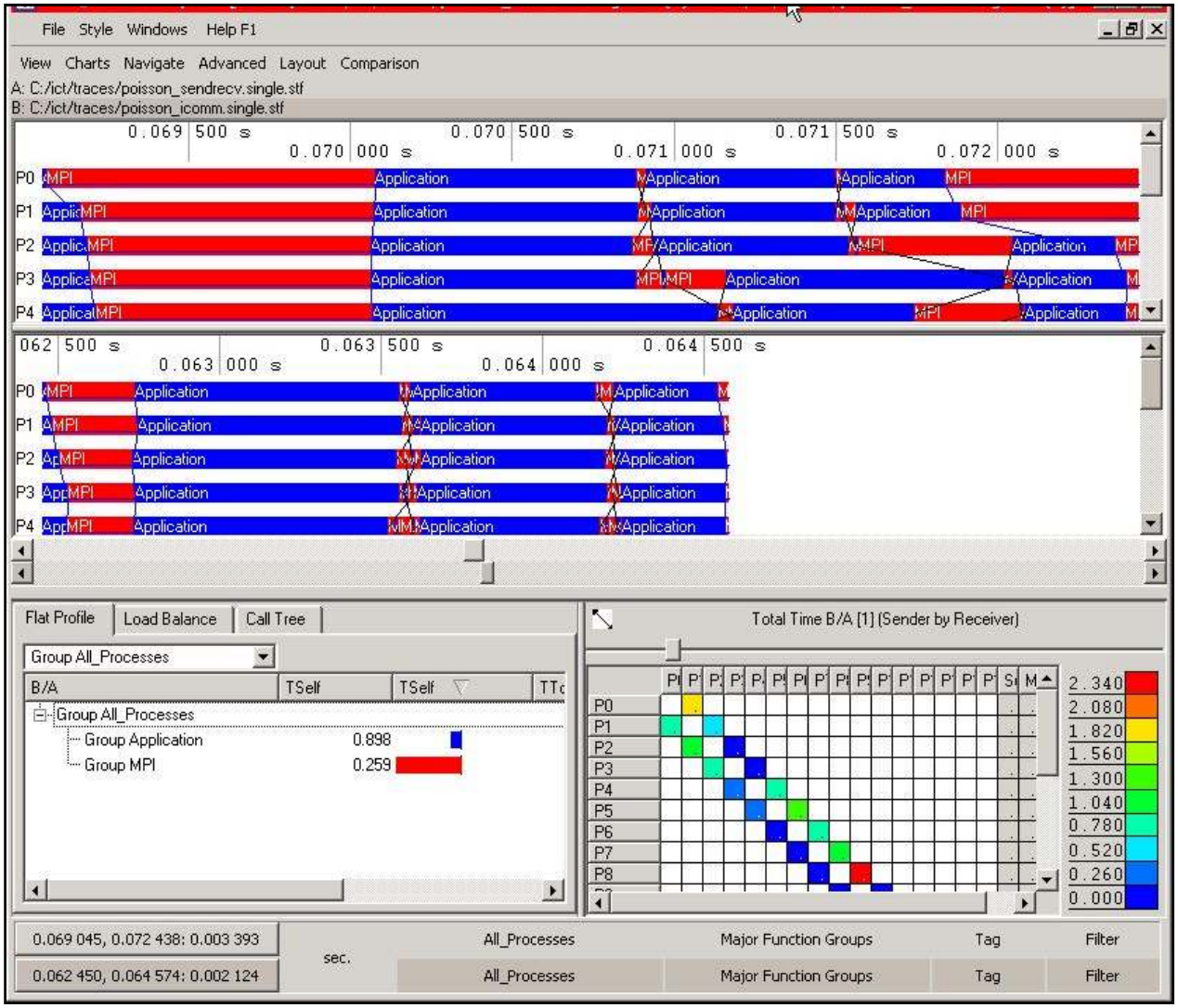
MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Intel® Trace Analyzer and Collector



Compare the event timelines of two communication profiles

Blue = computation
Red = communication

Chart showing how the MPI processes interact

Optimization Notice

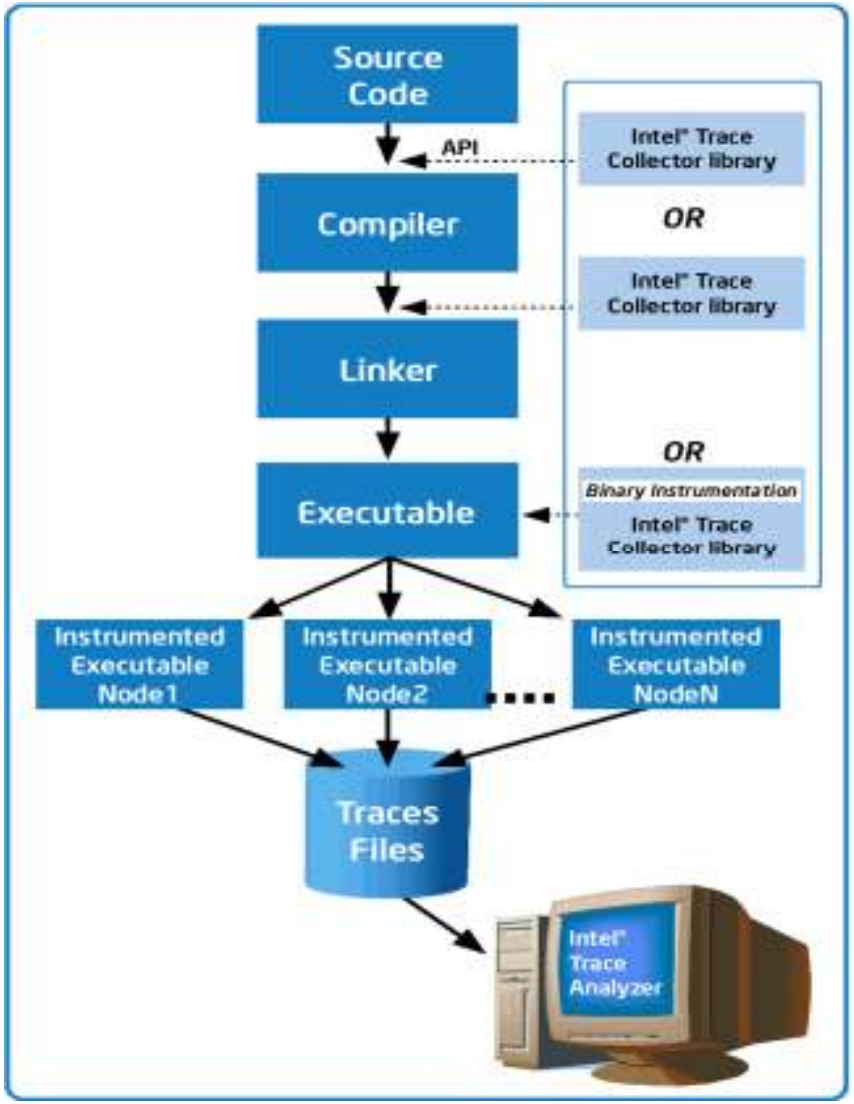
Intel® Trace Analyzer and Collector Overview

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



Optimization Notice

Full tracing functionality on Intel® Xeon Phi™ coprocessor

View Charts Navigate Advanced Layout

Flat Profile | Load Balance | Call Tree | Call Graph

Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	178.215 s		408.016 s	8	22.2768 s
MPI_Comm_size	56.995e-6 s		56.995e-6 s	8	7.12437e-6 s
MPI_Comm_rank	176.984e-6 s		176.984e-6 s	8	22.123e-6 s

20.1595 s

View Charts Navigate Advanced Layout

0.000 000, 53.010 399: 53.010 399 sec. | All_Nodes | MPI expanded in (Major Function Groups) | Tag | Filter

Optimization Notice

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

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

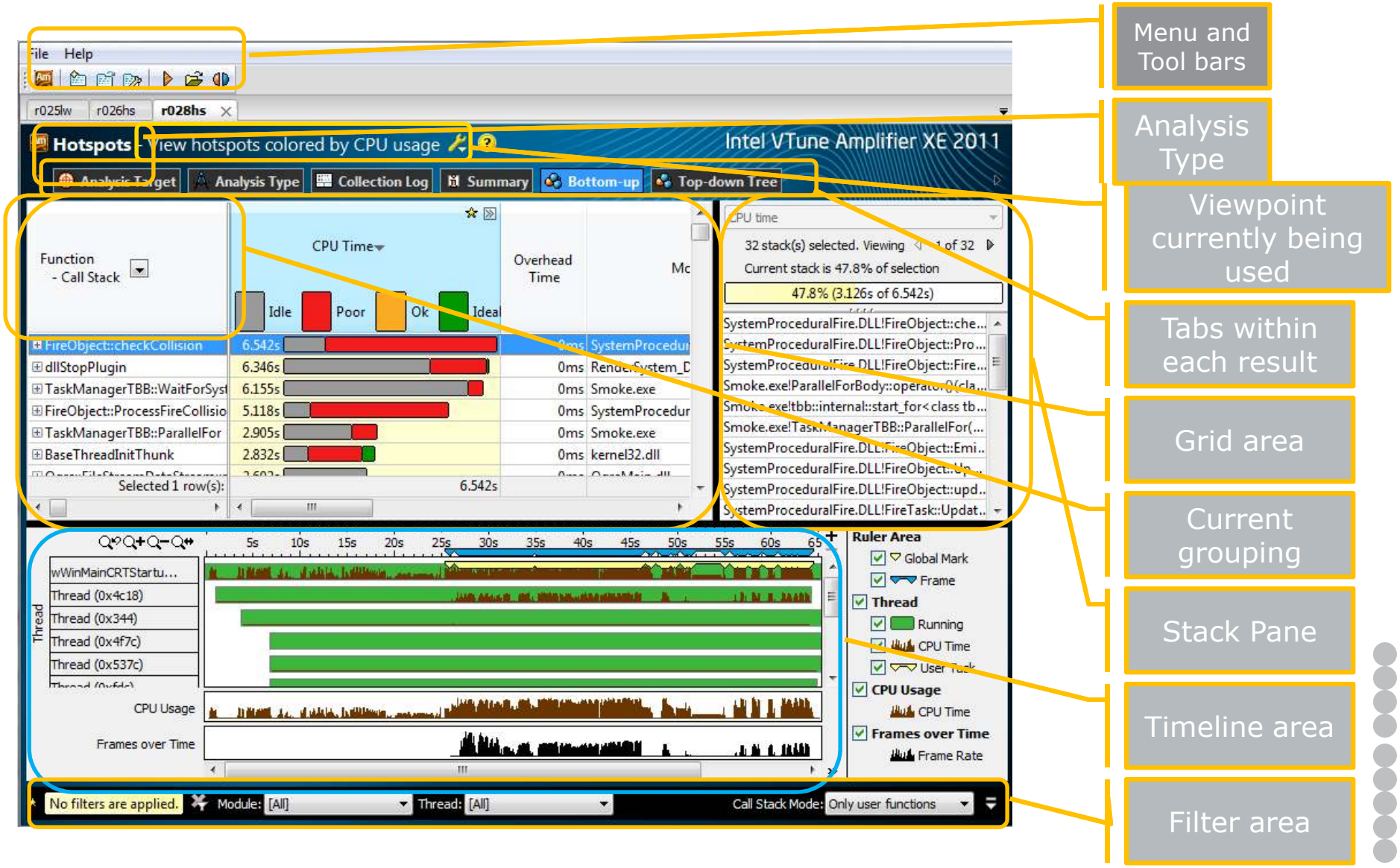
Collection

- Invoke from Intel® VTune™ Amplifier XE
- 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

- Intel VTune Amplifier XE performance database

Intel® VTune™ Amplifier XE offers a rich GUI



Menu and Tool bars

Analysis Type

Viewpoint currently being used

Tabs within each result

Grid area

Current grouping

Stack Pane

Timeline area

Filter area

Optimization Notice

Intel® VTune™ Amplifier XE on Intel® Xeon Phi™ coprocessors

Adjust Data Grouping

- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack

... (Partial list shown)

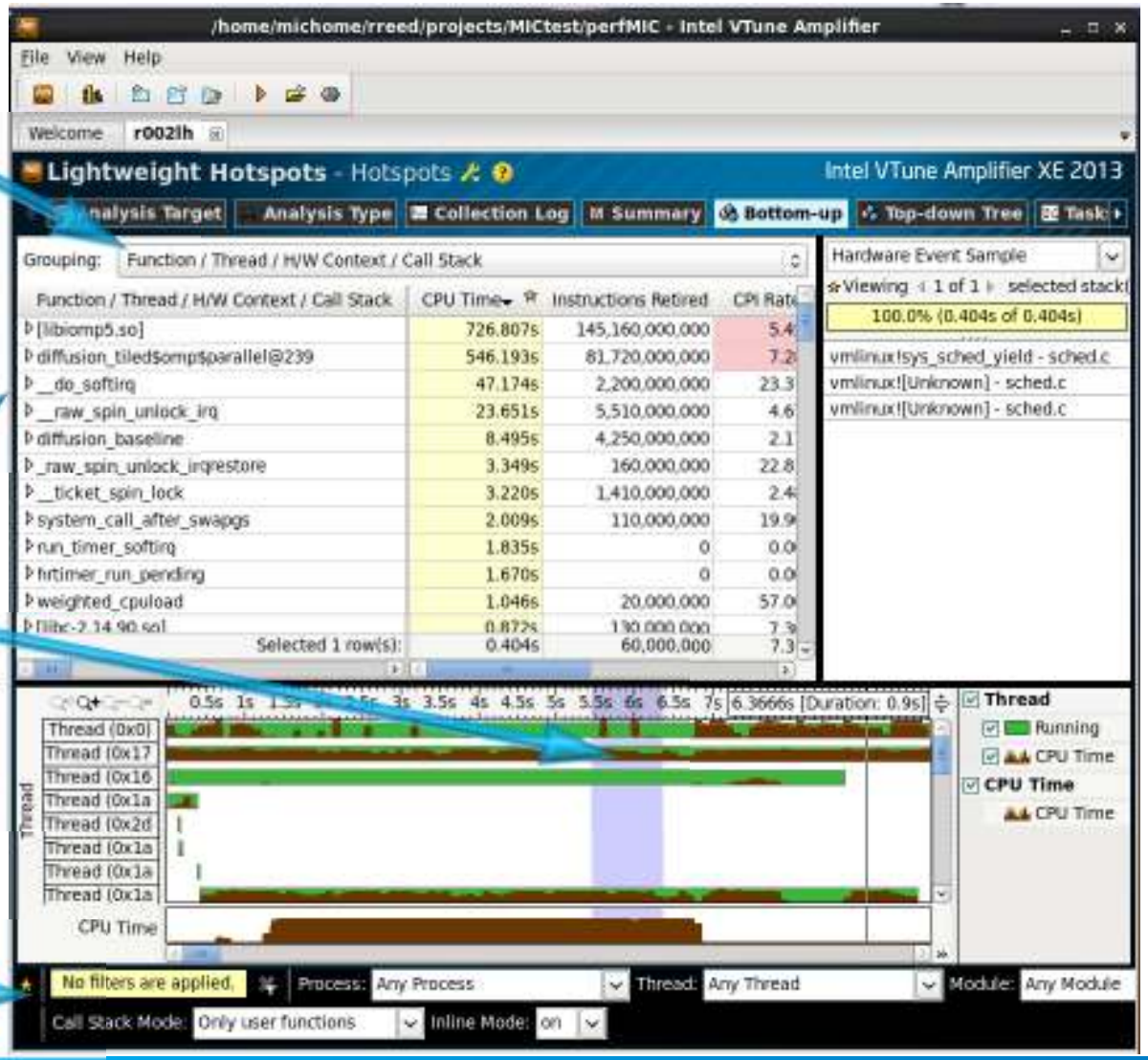
No Call Stacks Yet

Double Click Function to View Source

Filter by Timeline Selection (or by Grid Selection)

- Zoom In And Filter On Selection
- Filter In by Selection
- Remove All Filters

Filter by Module & Other Controls



Optimization Notice

Intel® VTune™ Amplifier XE displays event data at function, source & assembly levels

Lightweight Hotspots - Hotspots

Intel VTune Amplifier XE 2013

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Tasks | diffusion.c

Source | Assembly

Time on Source / Asm

Quick Asm navigation:
Select source to highlight Asm

Quickly scroll to hot spots.
Scroll Bar "Heat Map" is an overview of hot spots

Right click for instruction reference manual

Click jump to scroll Asm

		CPU Ti...	Instructions ...
269	++t;	2.110s	470,000,000
270	f2_t[c] = cc * f1_t[c] + c	146.807s	19,380,000,0...
271	+ cs * f1_t[s] + cn *	299.862s	41,250,000,0...
272	}		
273	++c;	0.624s	470,000,000
278	f2_t[c] = cc * f1_t[c] + cw	14.615s	1,890,000,000
279	+ cs * f1_t[s] + cn * f1	1.734s	620,000,000
280	} // tile ny		
281	} // tile nz		
282	} // block ny	0.009s	10,000,000
283	REAL *t = f1_t;	0.037s	
284	f1_t = f2_t;		
	Selected 1 row(s):	299.862s	41,250,000,000

Code Loc...	Sou..	Assembly	CP.	Ins...
0x407834	271	vloadunpackhd 0x44(%rax,%r14,4), %k0, %z	3.1..	540..
0x40783c	270	vprefetche1 0x204(%rdx,%r14,4)	22...	2,2..
0x407846	271	vloadunpackhd 0x44(%r12,%r14,4), %k0, %z	3.5..	650..
0x40784e	270	vprefetche0 0xc4(%rdx,%r14,4)	7.0..	950..
0x407858	270	vloadunpackhd 0x44(%rbx,%r14,4), %k0, %z	3.5..	810..
0x407860	271	vmulps %zmm15, %zmm4, %k0, %zmm19	3.1..	810..
0x407866	271	vmulps %zmm17, %zmm8, %k0, %zmm20	3.2..	840..
0x40786c	271	vmulps %zmm21, %zmm11, %k0, %zmm25	3.1..	800..
0x407872	271	vloadunpackhd 0x44(%rcx,%r14,4), %k0, %z	3.3..	1,0..
0x40787a	271	vfmadd231ps %zmm12, %zmm16, %k0, %zmm19	3.1..	950..
0x407880	271	vfmadd231ps %zmm5, %zmm18, %k0, %zmm20	3.0..	1,0..
0x407886	271	vfmadd231ps %zmm9, %zmm22, %k0, %zmm25	2.8..	1,1..
0x40788c	271	vaddps %zmm20, %zmm19, %k0, %zmm24	3.2..	1,0..
0x407892	271	vfmadd231ps %zmm10, %zmm23, %k0, %zmm25	2.8..	990..
0x407898	271	vadd		
0x40789e	270	nop		
0x40789f	270	vmovaps %zm		
0x4078aa	264	add $0x10,		
0x4078ae	264	cmp %r11, %r14	3.5..	1,0..
0x4078b1	264	jb 0x407781 <Block 59>	0.0..	20,..
		Block 60:		
0x4078b7	266	mov d, %eax	0.6..	250..
		Highlighted 172 row(s):	299..	41,..

Intel® VTune™ Amplifier XE 2013

Optimization Notice

Introduction

High-level overview of the Intel® Xeon Phi™ platform: Hardware and Software

Intel Xeon Phi Coprocessor programming considerations: Native or Offload

Performance and Thread Parallelism

MPI Programming Models

Tracing: Intel® Trace Analyzer and Collector

Profiling: Intel® Trace Analyzer and Collector

Conclusions

Conclusions: Intel® Xeon Phi™ Coprocessor supports a variety of programming models

The familiar Intel development environment is available:

- Intel® Composer: C, C++ and Fortran Compilers
- OpenMP*
- Intel® MPI Library support for the Intel® Xeon Phi™ Coprocessor
  - Use as an MPI node via TCP/IP or OFED
- Parallel Programming Models
  - Intel® Threading Building Blocks (Intel® TBB)
  - Intel® Cilk™ Plus
- Intel support for gdb on Intel Xeon Phi Coprocessor
- Intel Performance Libraries (e.g. Intel Math Kernel Library)
  - Three versions: host-only, coprocessor-only, heterogeneous
- Intel® VTune™ Amplifier XE for performance analysis
- Standard runtime libraries, including pthreads*

Intel® Xeon Phi™ Coprocessor Developer site:

http://software.intel.com/mic-developer

One Stop Shop for:

- Tools & Software Downloads
- Getting Started Development Guides
- Video Workshops, Tutorials, & Events
- Code Samples & Case Studies
- Articles, Forums, & Blogs
- Associated Product Links

http://software.intel.com/mic-developer

Resources

http://software.intel.com/mic-developer

- Developer's Quick Start Guide
- Programming Overview
- New User Forum at

  http://software.intel.com/en-us/forums/intel-many-integrated-core

http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture

http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture

Intel® Composer XE 2013 for Linux* User and Reference Guides

Intel Premier Support https://premier.intel.com

Optimization Notice


intel®

Legal Disclaimer & Optimization Notice

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

Offloaded data have some restrictions and directives to channel their transfer

Offload data are limited to scalars, arrays, and "bitwise-copyable" structs (C++) or derived types (Fortran)

- No structures with embedded pointers (or allocatable arrays)
- No C++ classes beyond the very simplest
- Fortran 2003 object constructs also off limits, mostly
- Data exclusive to the coprocessor has no restrictions

Offload data includes all scalars & named arrays in lexical scope, which are copied both directions automatically

- IN, OUT, INOUT, NOCOPY are used to limit/channel copying
- Data not automatically transferred:
  - Local buffers referenced by local pointers
  - Global variables in functions called from the offloaded code
- Use IN/OUT/INOUT to specify these copies – use LENGTH

alloc_if() and free_if() provide a means to manage coprocessor memory allocs

Both default to true: normally coprocessor variables are created/destroyed with each offload

A common convention is to use these macros:

```
#define ALLOC alloc_if(1)
#define FREE free_if(1)
#define RETAIN free_if(0)
#define REUSE alloc_if(0)
```

To allocate a variable and keep it for the next offload

```
#pragma offload target(mic)  in(p:length(n) ALLOC RETAIN)
```

To reuse that variable and keep it again:

```
#pragma offload target(mic)  in(p:length(n) REUSE  RETAIN)
```

To reuse one more time, then discard:

```
#pragma offload target(mic)  in(p:length(n) REUSE  FREE)
```