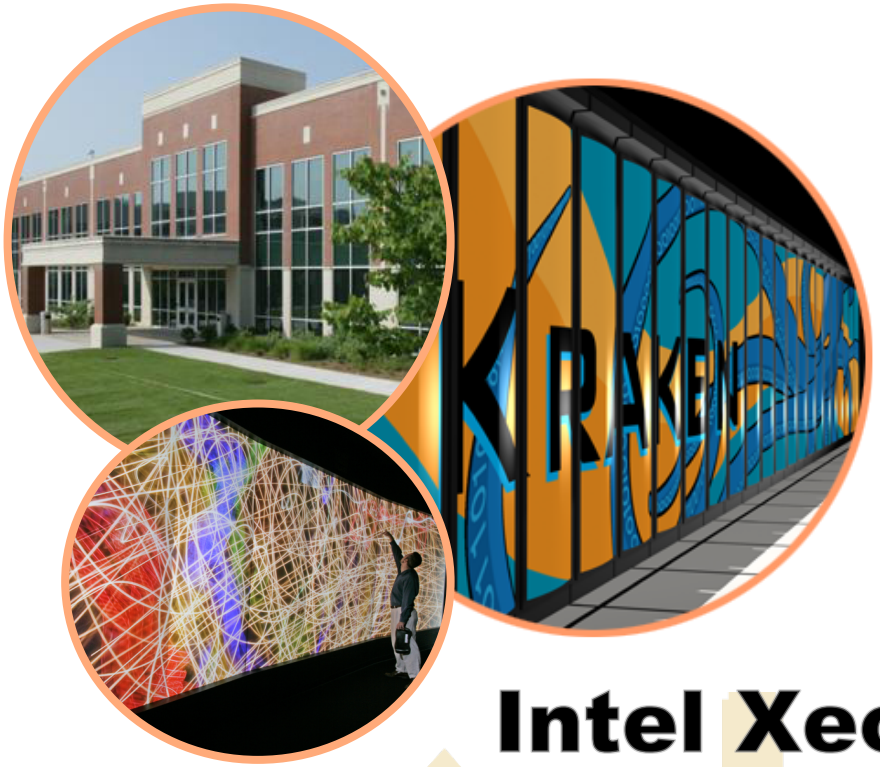


A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

Intro to Beacon and Intel Xeon Phi Coprocessors

Ryan Hulguin

ryan-hulguin@tennessee.edu

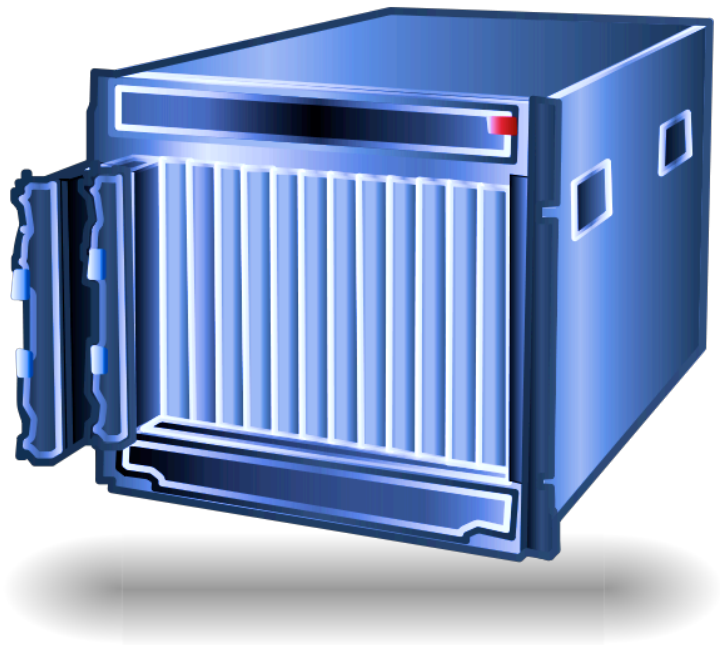
Outline

- Beacon
  - The Beacon project
  - The Beacon cluster
  - TOP500 ranking
  - System specs
- Xeon Phi Coprocessor
  - Technical specs
  - Many core trend
  - Programming models
  - Applications and performance

The Beacon Project

- This material is based upon work supported by the National Science Foundation (NSF) under Grant #1137097
- NSF funding is used to port and optimize scientific codes to the Intel® Xeon Phi™ coprocessor
- The state of Tennessee has funded an expansion focusing on energy efficiency, big data applications, and industry
- The pre-production Intel® Xeon Phi™ coprocessors in the original Beacon cluster have been upgraded to the commercial Intel® Xeon Phi™ 5110P coprocessors
- Currently, there are 60 projects that may request time on the Beacon cluster

Beacon System Specs



Beacon Cray CS300-AC Cluster	
CPU cores	768
Coprocessor cores	11520
Total system RAM	12 TB
Total coprocessor RAM	1.5 TB
Total SSD storage	73 TB
I/O nodes	6
Interconnect	FDR InfiniBand

Node configuration
Two 2.6 GHz eight-core Intel® Xeon® E5-2670
256 GB memory, 960 GB SSD storage
Four Intel® Xeon Phi™ 5110P coprocessors

top500.org list as of June 2013

Rank	Site	System	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National University of Defense Technology China	Tianhe-2 (Milky Way-2) *Xeon Phi*	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan *Nvidia K20x*	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	Sequoia	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	Mira	8586.6	10066.3	3945
397	**National Institute for Computational Sciences/ University of Tennessee United States**	**Beacon** ***Xeon Phi***	**110.5**	**157.5**	**45**

New World Record

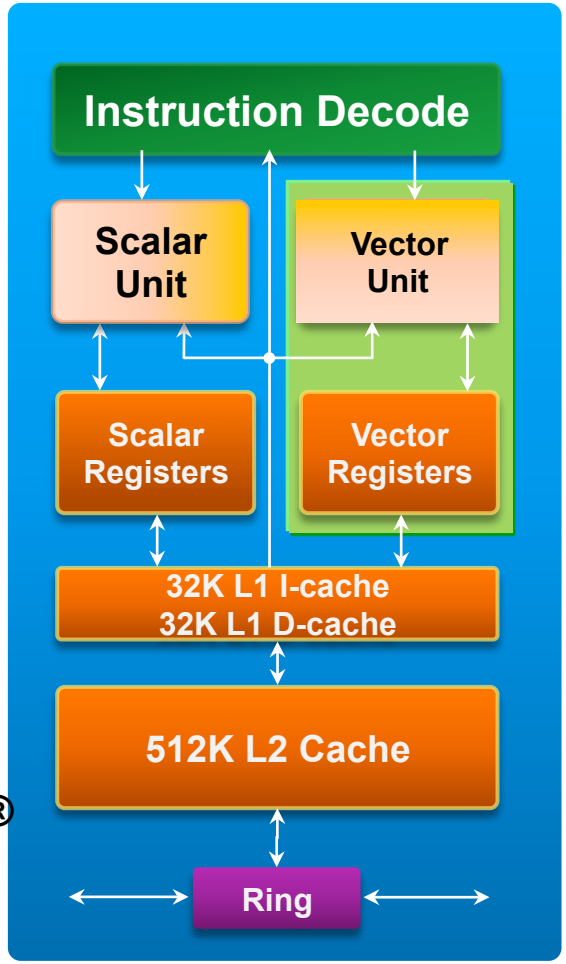
Green500 as of November 2012

Intel® Xeon Phi™ Coprocessor

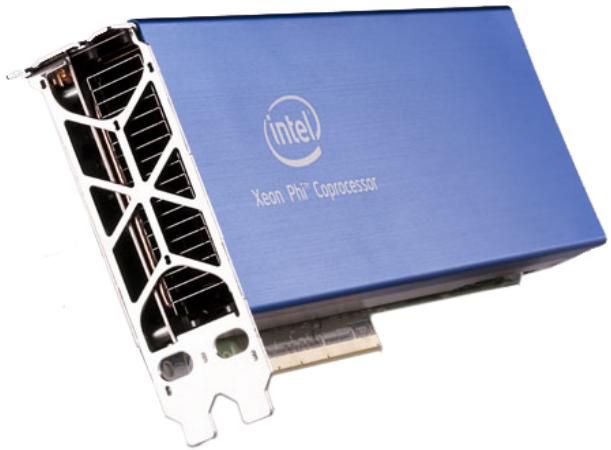
- Xeon Phi is the brand name that Intel uses for all their products based on the Many Integrated Core (MIC) architecture
- The cores are based on the x86 instruction set
- Xeon Phi can be programmed in familiar languages (C/C++ and Fortran) with familiar parallel programming models (OpenMP and/or MPI)
- Xeon Phi was initially referred to as Knights Corner (KNC)
- Knights Landing (KNL) is the codename for the next generation MIC product

Intel® Xeon Phi™ Coprocessor Overview

- Up to 61 in-order cores
  - Ring interconnect
- 64-bit addressing
- Two pipelines
  - Intel® Pentium® processor family-based scalar units
    - Dual issue with scalar instructions
  - Pipelined one-per-clock scalar throughput
    - 4 clock latency, hidden by round-robin scheduling of threads
- 4 hardware threads per core
  - Cannot issue back to back instruction in same thread
- All new vector unit
  - 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
  - 32 512-bit wide vector registers
    - 16 singles or 8 doubles per register
- Fully-coherent L1 and L2 caches



Intel® Xeon Phi™ coprocessor 5110P (codenamed Knights Corner)



The Intel® Xeon Phi™ coprocessor (codenamed Knights Corner) is the first commercial product employing the Intel® Many Integrated Core (MIC) architecture.

The Intel® Xeon Phi™ coprocessor 5110P shown here employs passive cooling.

SKU #	5110P
Form factor	PCIe card
Thermal solution	passively cooled
Peak double precision	1011 GF
Max number of cores	60
Core clock speed	1.053 GHz
Memory capacity	8 GB
GDDR5 memory speeds	5.0 GT/s
Peak memory BW	320
Total cache	30 MB
Board TDP	225 Watts
Fabrication process	22 nm

x86 SMP-on-a-chip running Linux

- SMP = symmetric multiprocessor
  - shared memory running a single operating system
- Each coprocessor has its own ip address
- Can ssh to individual coporocessors
- Feels like an independent compute node
- Currently uses a custom Linux operating system (Not Ubuntu, Red Hat, etc.)

Many core trend

- In the early 2000s, CPU core speeds plateaued at ~3 GHz
- Further advances to increase computing power are achieved using parallel programming
- As seen in the previous Top500 slide, supercomputers are now turning to accelerators/coprocessors to increase computing power
- This new hardware is requiring new ways of programming
- There is a push for hybrid distributed/threaded programming using these accelerators/coprocessors

Why Use Xeon Phi?

- Code tuned to run on Xeon Phi is guaranteed to run well on normal Xeon CPUs
- Popular programming models such as MPI, OpenMP, and Intel TBB are fully supported
- Support for newer models exists as well: Coarray Fortran, Intel Cilk Plus, and OpenCL
- Can run programs that GPUs cannot
  - Example: NWChem

Considerations for Good Performance on the Intel® Xeon Phi™ Coprocessor

- Some portion of the code must be highly parallel and highly vectorizable
  - Not all code can be written this way
  - Serial code run on the Intel Xeon Phi will take a huge performance hit
- Very short (low-latency) tasks are not optimal to offload to the coprocessor
  - There will be thread setup and communication overhead

Programming Models – Brief Overview

- **Native Mode**
  - Everything runs on the MIC
  - May have issues with libraries not existing, needing copied over (e.g., MKL, MPI with debug symbols)
  - Especially useful when the majority of the code can run in parallel
- **Offload Mode**
  - Serial portion runs on host
  - Parallel portions are offloaded and run on the MIC
  - Especially useful when there are plenty of data dependent serial calculations, and only small sections of code that can run in parallel
- **Automatic Offload Mode**
  - select MKL functions only

Native mode

- Native mode libraries and binaries are created by simply compiling with the –mmic compile flag
- After they are created, they need to be either copied to the coprocessors directly (scp file mic0), or be placed in a directory on a file system that is mounted by the coprocessors (i.e. lustre/medusa scratch filesystem)
- This does ***not*** make a serial code parallel
- The native mode binaries can be launched by either connecting to the coprocessor via ssh or by using MPI to launch it remotely from the host node

Offload Mode

- Code starts running on host and regions designated to be offloaded via pragmas are run on the MIC card when encountered
- The host CPU and the MIC cards do not share memory in hardware
- Data is passed to and from the MIC card explicitly or implicitly (C/C++ only)

C/C++ Syntax	Fortran Syntax
#pragma offload <clauses> <statement>	!dir$ offload <clauses> <statement>

- The statement <u>immediately following the offload pragma/directive</u> will be run on a coprocessor

Marking Variables/Functions for use on MIC

- In offload mode, the compiler needs to know ahead of time which functions will run on the MIC
- Also any variables that are to exist on both the host and the MIC need to be known by the compiler as well
- This is done for both functions and variables using the following keyword

C/C++ Syntax	Fortran Syntax
__attribute__((target(mic)))	!dir$ attributes offload:<MIC> :: <routine-name>

An alternative keyword is __declspec(target (mic))

Explicit Copy

- Programmer identifies the variables that need copying to and from the card ***in the offload directive***
- C/C++ Example:
  - `#pragma offload target(mic) in(data:length(size))`
- Fortran Example:
  - `!dir$ offload target(mic) in(data:length(size))`
- Variables and pointers to be copied are restricted to scalars, structs of scalars, and arrays of scalars
  - i.e. `double *var` is allowed, but not `double **var`.

Explicit Copy Clauses and Modifiers

Clauses / Modifiers	Syntax	Semantics
Target specification	`target( name[:card_number] )`	Where to run construct
Conditional offload	`if (condition)`	Boolean expression
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 pointer length	`length(element-count-expr)`	Copy N elements of the pointer's type
Control pointer memory allocation	`alloc_if ( condition )`	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	`free_if ( condition )`	Free memory used by pointer if condition is TRUE
Control target data alignment	`align ( expression )`	Specify minimum memory alignment on target

Implicit Copy

- This method is available only in C/C++
- Sections of memory are maintained at the same virtual address on both the host and the MIC
- This enables sharing of complex data structures that contain pointers
- This “shared” memory is synchronized when entering and exiting an offload call
- Only modified data is transferred between CPU and MIC

Dynamic Memory Allocation Using Implicit Copies

- Special functions are needed in order to allocate and free dynamic memory for implicit copies

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

The _Cilk_shared keyword for Data and Functions

What	Syntax	Semantics
Function	`int _Cilk_shared f(int x) { return x+1; }`	Versions generated for both CPU and card; may be called from either side
Global	`Cilk_shared int x = 0;`	Visible on both sides
File/Function static	`static _Cilk_shared int x;`	Visible on both sides, only to code within the file/function
Class	`class _Cilk_shared x {…};`	Class methods, members, and and operators are 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 large blocks of code _Cilk_shared using this pragma

Offloading using Implicit Copy

- Rather than using a pragma directive, the keyword "`_Cilk_offload`" is used when calling a function to be run on the MIC
  - Examples:
    - `x = _Cilk_offload function(y)`
    - `x = _Cilk_offload_to (card_number) function(y)`
  - Note: function needs to be defined using the `_Cilk_shared` keyword

Explicit/Implicit Copy Comparison

	Offload via Explicit Data Copying	Offload via Implicit Data Copying
Language Support	Fortran, C, C++ (*C++ functions may be called, but C++ classes cannot be transferred*)	C, C++
Syntax	Pragmas/Directives: • `#pragma offload` in C/C++ • `!dir$ omp offload` directive in Fortran	Keywords: `_Cilk_shared` and `_Cilk_offload`
Used for…	Offloads that transfer contiguous blocks of data	Offloads that transfer all or parts of complex data structures, or many small pieces of data

Select Offload Examples

- The offload mode allows select portions of a code to run on the Intel MIC, while the rest of it runs on the host.
- Ideally, the offload regions are highly parallel
- What follows is select offload examples, provided by Intel, that demonstrate how to move data to and from the Intel MIC cards
- Intel has many offload examples located in the following directory
  - `/global/opt/intel/composerxe_mic/Samples/en_US/C++/mic_samples/intro_sampleC/`
- They can be copied to a directory of your choice and then compiled with `make mic`

SampleC01

- **This code computes Pi on the MIC using `#pragma offload`**

```
float pi = 0.0f;
int count = 10000;
int i;

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

- **`#pragma offload target (mic)` runs the very next line (or block of code if braces are used) on the Intel MIC**
  - **In this case the whole for loop is run on the Intel MIC**
- **Note that pi was declared outside of the offload region, and it did not need to be explicitly copied to the MIC since it is a scalar**

SampleC02

- This code initializes 2 arrays on the host, and then has the Intel MIC add the arrays together, and store the result in a third array

```
typedef double T;

#define SIZE 1000

#pragma offload_attribute(push, target(mic))
static T in1_02[SIZE];
static T in2_02[SIZE];
static T res_02[SIZE];
#pragma offload_attribute(pop)

static void populate_02(T* a, int s);
```

- The `#pragma offload_attribute(push/pop)` pair marks the block of code between them to be used on both the host and the Intel MIC
- They could have been marked individually with `__attribute__((target(mic)))`
- Without those statements, the Intel MIC would not be able to see/use the 3 arrays

SampleC02 Continued

- The sum of the 2 arrays is done by the Intel MIC
- Note that only a single Intel MIC core is used

```
void sample02()
{
    int i;
    populate_02(in1_02, SIZE);
    populate_02(in2_02, SIZE);

    #pragma offload target(mic)
    {
    for (i=0; i<SIZE; i++)
        {
            res_02[i] = in1_02[i] + in2_02[i];
        }
    }
}
```

SampleC03

- **This program is similar to SampleC02, except that it avoids unnecessary data transfer**

```
void sample03()
{
    int i;
    populate_03(in1_03, SIZE);
    populate_03(in2_03, SIZE);

    #pragma offload target(mic) in(in1_03, in2_03) out(res_03)
    {
        for (i=0; i<SIZE; i++)
        {
            res_03[i] = in1_03[i] + in2_03[i];
        }
    }
}
```

- **Previously, all 3 arrays were copied to the card at the start of the offload call, and then copied back at the end of the offload call**
- **Now, only the `in1_03` and `in2_03` arrays are copied to the card, and only the `res_03` array is copied back**

SampleC04

- **This program is similar to the previous two samples, but now we are dealing with pointers instead of the static arrays directly**

```
void sample04()
{
    T* p1, *p2;
    int i, s;
    populate_04(in1_04, SIZE);
    populate_04(in2_04, SIZE);

    p1 = in1_04;
    p2 = in2_04;
    s = SIZE;

    #pragma offload target(mic) in(p1, p2:length(s)) out(res_04)
    {
        for (i=0; i<s; i++)
        {
                res_04[i] = p1[i] + p2[i];
        }
    }
}
```

- **Since the length of the pointer is not known, it must be explicitly passed as an argument**
- **`res_04` is still a static array in this sample**

SampleC05

- This program is like the last except the sum of the arrays, via pointers, is now stored in a pointer to the result array
- This pointer needs to have its length specified as well
- Also, the summation now happens in the function `get_result()`
- `get_result()` did not need to be marked with `__attribute__ ((target(mic)))` because it was called by the host and not by the Intel MIC

```
void sample05()
{
    T my_result[SIZE];
    populate_05(in1_05, SIZE);
    populate_05(in2_05, SIZE);

    get_result(in1_05, in2_05, my_result, SIZE);
}
```

```
static void get_result(T* pin1, T* pin2,
                       T* res, int s)
{
    int i;

    #pragma offload target(mic)          \
            in(pin1, pin2 : length(s))   \
            out(res : length(s))
    {
        for (i=0; i<s; i++)
        {
            res[i] = pin1[i] + pin2[i];
        }
    }
}
```

SampleC07

- In this program, an array of data is sent from the host to the Intel MIC in one offload call
- The array values are then doubled on the MIC in a separate offload call, as long as a MIC card exists

```
#define SIZE 1000

__attribute__((target(mic))) int array1[SIZE];
__attribute__((target(mic))) int send_array(int* p, int s);
__attribute__((target(mic))) void compute07(int* out, int size);

void sample07()
{
    int in_data[16] = { 1, 2, 3, 4, 5, 6, 7, 8,
                        9, 10, 11, 12, 13, 14, 15, 16 };
    int out_data[16];
    int array_sent = 0;
    int num_devices;

    // Check if coprocessor(s) are installed and available
    num_devices = _Offload_number_of_devices();

    #pragma offload target(mic : 0)
    array_sent = send_array(in_data, 16);

    #pragma offload target(mic : 0) if(array_sent) out(out_data)
    compute07(out_data, 16);
}
```

SampleC07 Continued

- Reminder, `__attribute__((target(mic)))` makes it so both the host and the Intel MIC can see/use the variable/function
- The function `_Offload_number_of_devices()` returns how many Intel MIC cards are available
- The macro `__MIC__` lets you know if the MIC (value of 1) or host (value of 0) is currently evaluating the statements

```
__attribute__((target(mic))) int send_array(int* p, int s)
{
    int retval;
    int i;

    for (i=0; i<s; i++)
    {
        array1[i] = p[i];
    }

#ifdef __MIC__
    retval = 1;
#else
    retval = 0;
#endif

    // Return 1 if array initialization
    // was done on target
    return retval;
}
```

```
__attribute__((target(mic))) void compute07(int* out, int size)
{
    int i;
    for (i=0; i<size; i++)
    {
        out[i] = array1[i]*2;
    }
}
```

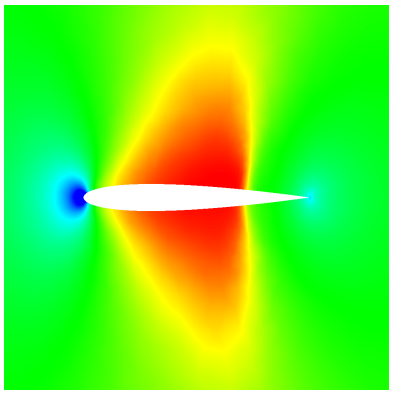
SampleC08

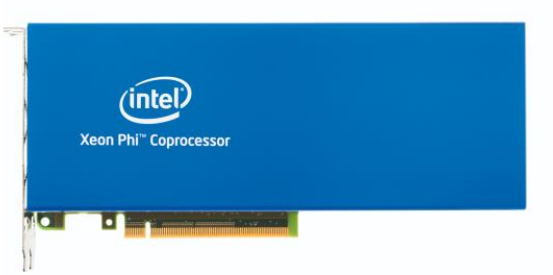
- **This program is like SampleC01, except now the Pi calculation is done using an OpenMP for loop on the Intel MIC to utilize the many cores**

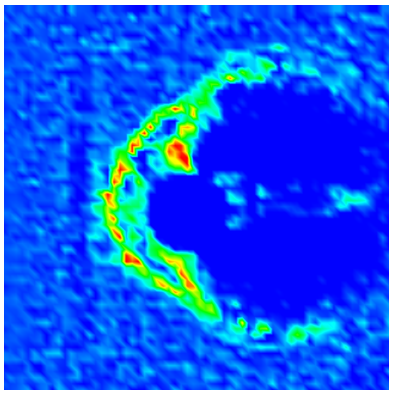
```
float pi = 0.0f;
int count = 10000;
int i;

#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;
```

Applications run on Xeon Phi

- **Science codes ported and/or optimized through the Beacon Project**
  - Chemistry – NWChem (ported)
  - Astrophysics – Enzo (ported and optimized)
  - Magnetospheric Physics – H3D (ported and optimized)
- **Other codes of interest**
  - Electronic Structures – Elk FP-LAPW (ported)
  - Computational Fluid Dynamics (CFD) – Euler and BGK Boltzmann Solver (ported and optimized)

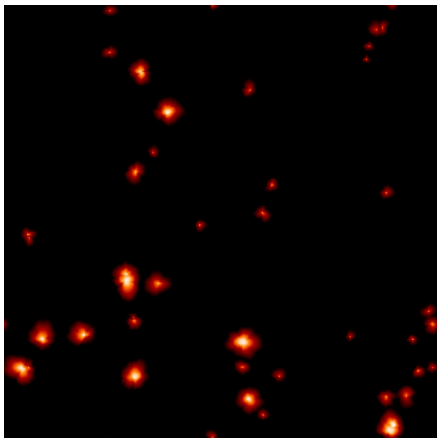
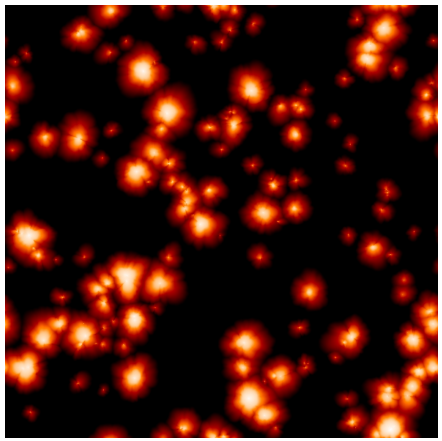
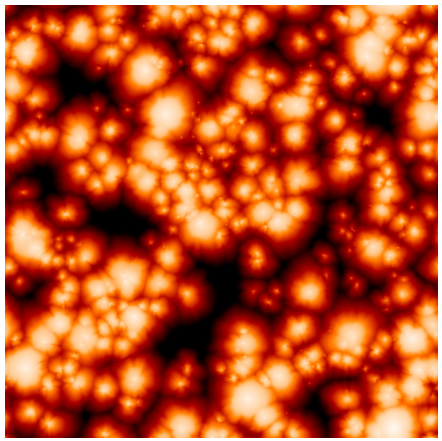






Enzo

- Community code for computational astrophysics and cosmology
- More than 1 million lines of code
- Uses powerful adaptive mesh refinement
- Highly vectorized with a hybrid MPI + OpenMP programming model
- Utilizes HDF5 and HYPRE libraries

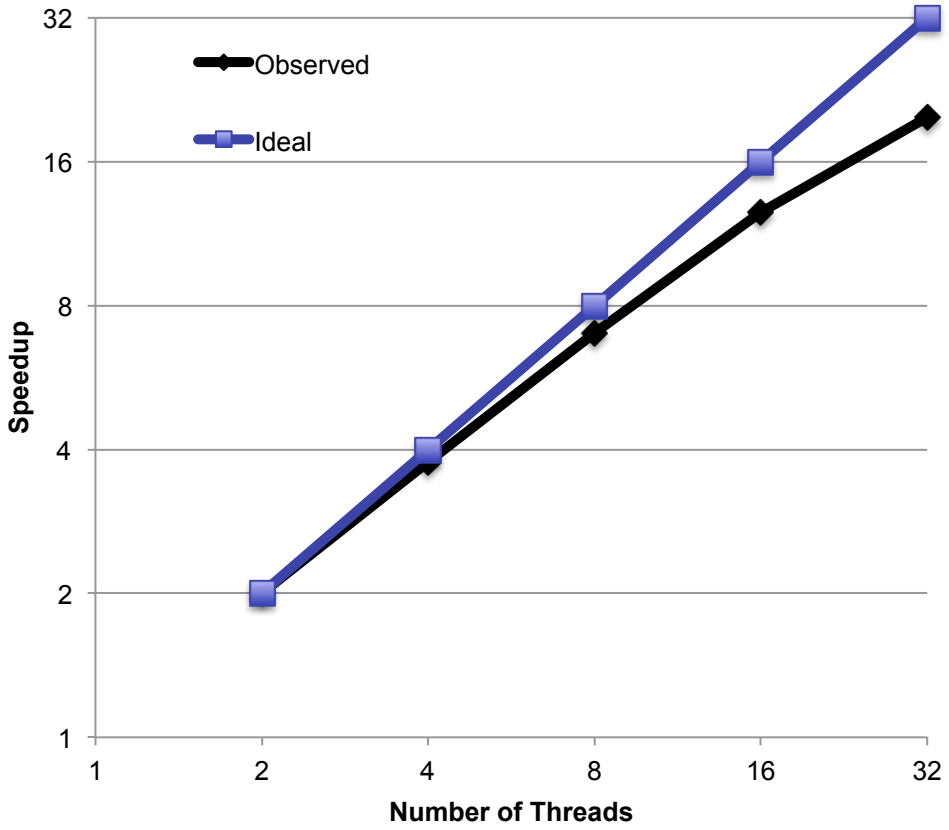




Enzo was ported and optimized for the the Intel® Xeon Phi™ Coprocessor by

Dr. Robert Harkness
harkness@sdsc.edu

Preliminary Scaling Study: Native

- ENZO-C
- 128^3 mesh (non-AMR)
- pure MPI
- native mode



Results were generated on the Intel® Knights Ferry software development platform

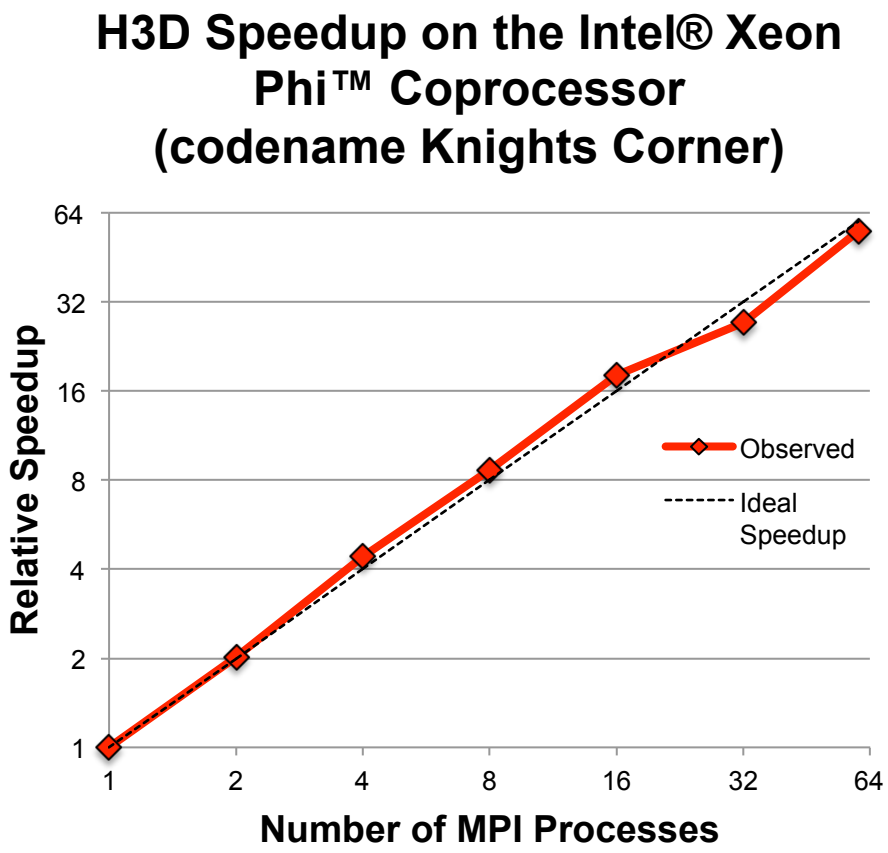
Hybrid3d (H3D)

- Provides breakthrough kinetic simulations of the Earth's magnetosphere
- Models the complex solar wind-magnetosphere interaction using both electron fluid and kinetic ions
  - **This is unlike magnetohydrodynamics (MHD), which completely ignores ion kinetic effects**
- Contains the following **HPC innovations:**
  1. multi-zone (asynchronous) algorithm
  2. **dynamic load balancing**
  3. code adaptation and optimization to large number of cores

Hybrid3d (H3D) was provided for porting to the the Intel® Xeon Phi™ Coprocessor by
Dr. Homa Karimabadi
hkarimabadi@ucsd.edu

Hybrid3d (H3D) Performance

Optimizations were provided by Intel senior software engineer Rob Van der Wjingaart

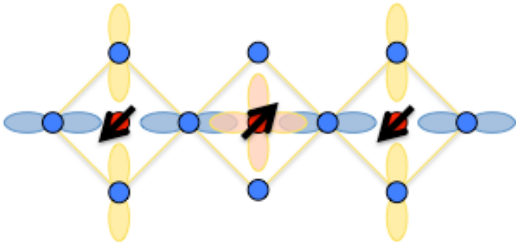


Results were generated on a Pre-Production Intel® Xeon Phi™ coprocessor with B0 HW and Beta SW
61 cores @ 1.09 GHz and 8 GB of GDDR5 RAM @ 2.75 GHz

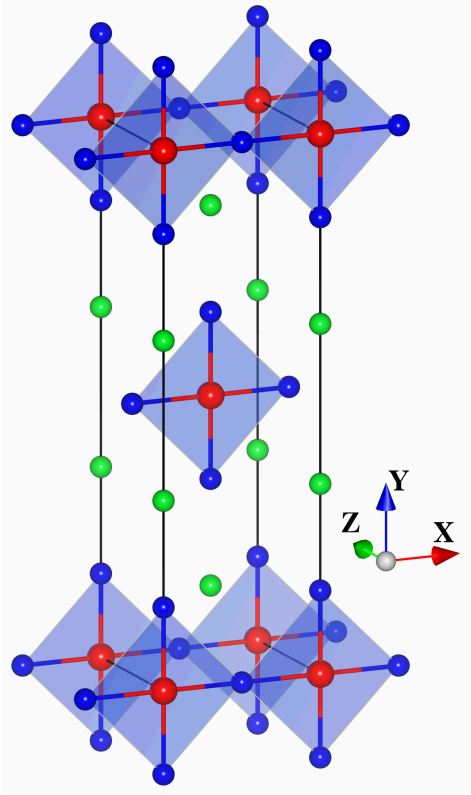
Elk FP-LAPW
http://elk.sourceforge.net/

Paramount to extracting functionality from these advanced materials is having a detailed understanding of their electronic, magnetic, vibrational, and optical properties.

Elk is a software platform which allows for the understanding of these properties from a first principles approach. It employs electronic structure techniques such as density functional theory, Hartree-Fock theory, and Green's function theory for the calculation of relevant properties from first principles.



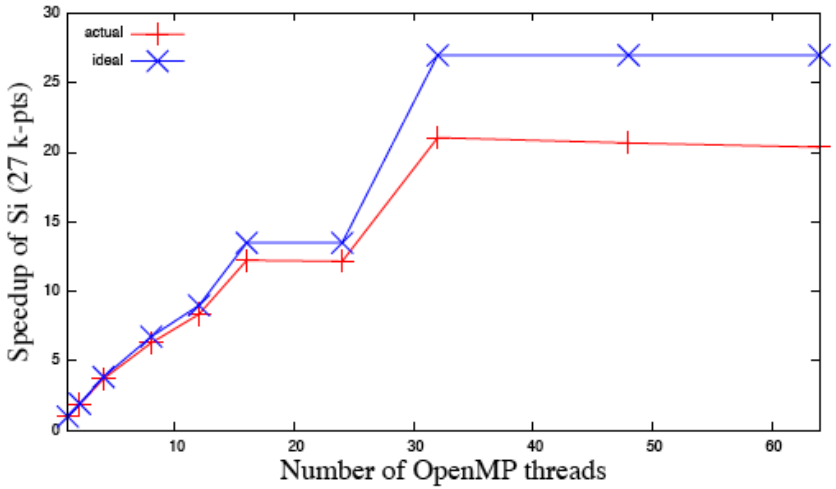
Antiferromagnetic structure of Sr_2CuO_3

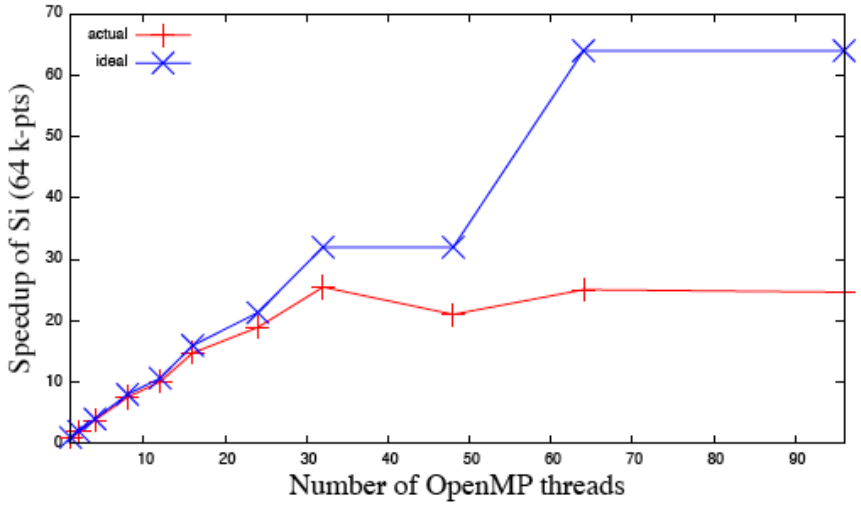


Elk was ported to the the Intel® Xeon Phi™ Coprocessor by
W. Scott Thornton
wsttiger@gmail.com

Elk FP-LAPW Performance

Elk uses master-slave parallelism where orbitals for different momenta are computed semi-independently. In this test 27 and 64 different crystal momenta were used. Test case was bulk silicon.

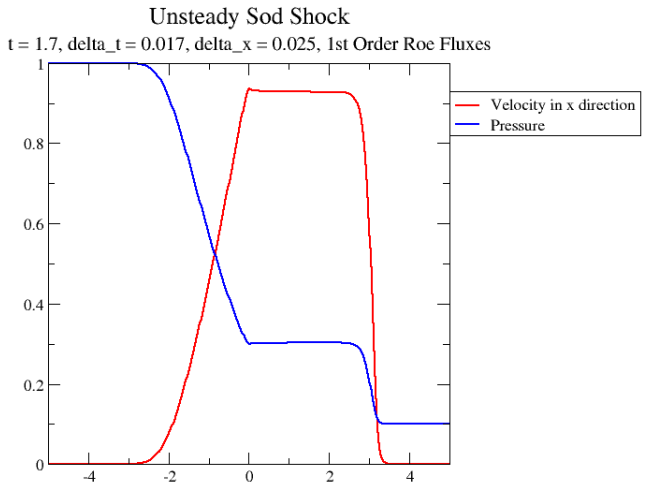




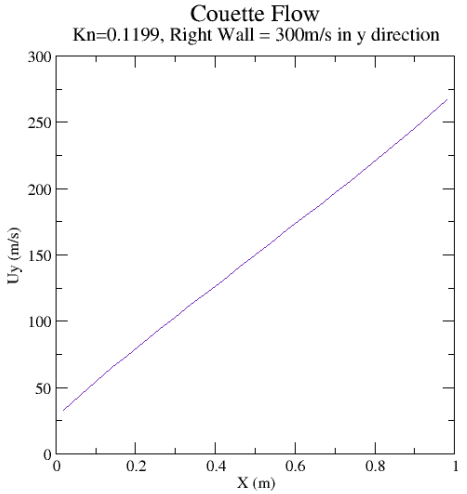
Results were generated on a Pre-Production Intel® Xeon Phi™ coprocessor with A0 HW and Beta SW
52 cores @ 1.00 GHz and 8 GB of GDDR5 RAM @ 2.25 GHz

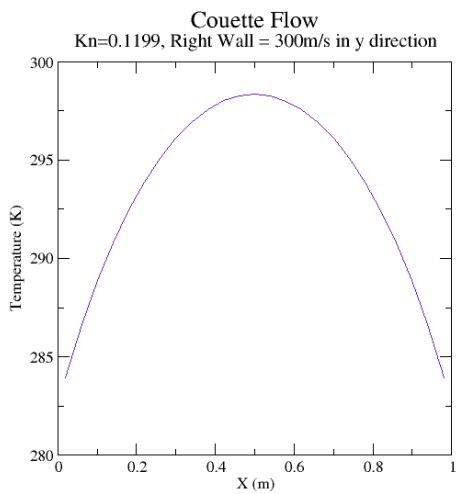
Computational Fluid Dynamics (CFD)

- **2 CFD solvers were developed in house at NICS**
- **1st solver is based on the Euler equations**
- **2nd solver is based on Model Boltzmann equations**



Unsteady solution of a Sod Shock using the Euler equations





Steady-state solution of a Couette flow using the Boltzmann equation with BGK collision approximation

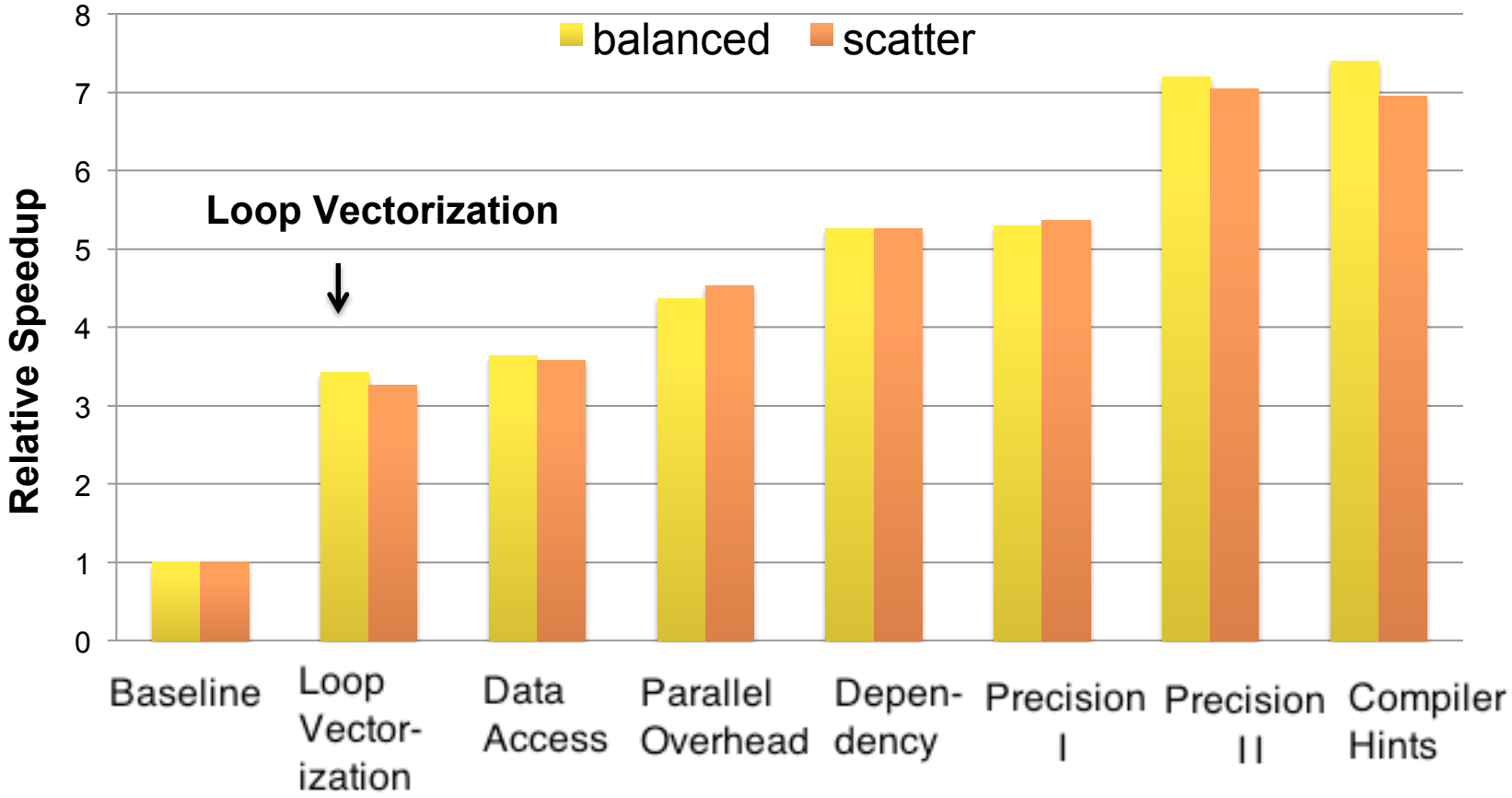
The above CFD solvers were developed for the Intel® Xeon Phi™ Coprocessor by
Ryan C. Hulguin
ryan-hulguin@tennessee.edu

Impact of Various Optimizations on the Model Boltzmann Equation Solver

- The Model Boltzmann Equation solver was optimized by Intel software engineer Rob Van der Wjingaart
- He took a baseline solver where all loops were vectorized except for one, and applied the following optimizations to get the most performance out of the Intel® Xeon Phi™ Coprocessor (codename Knights Corner)

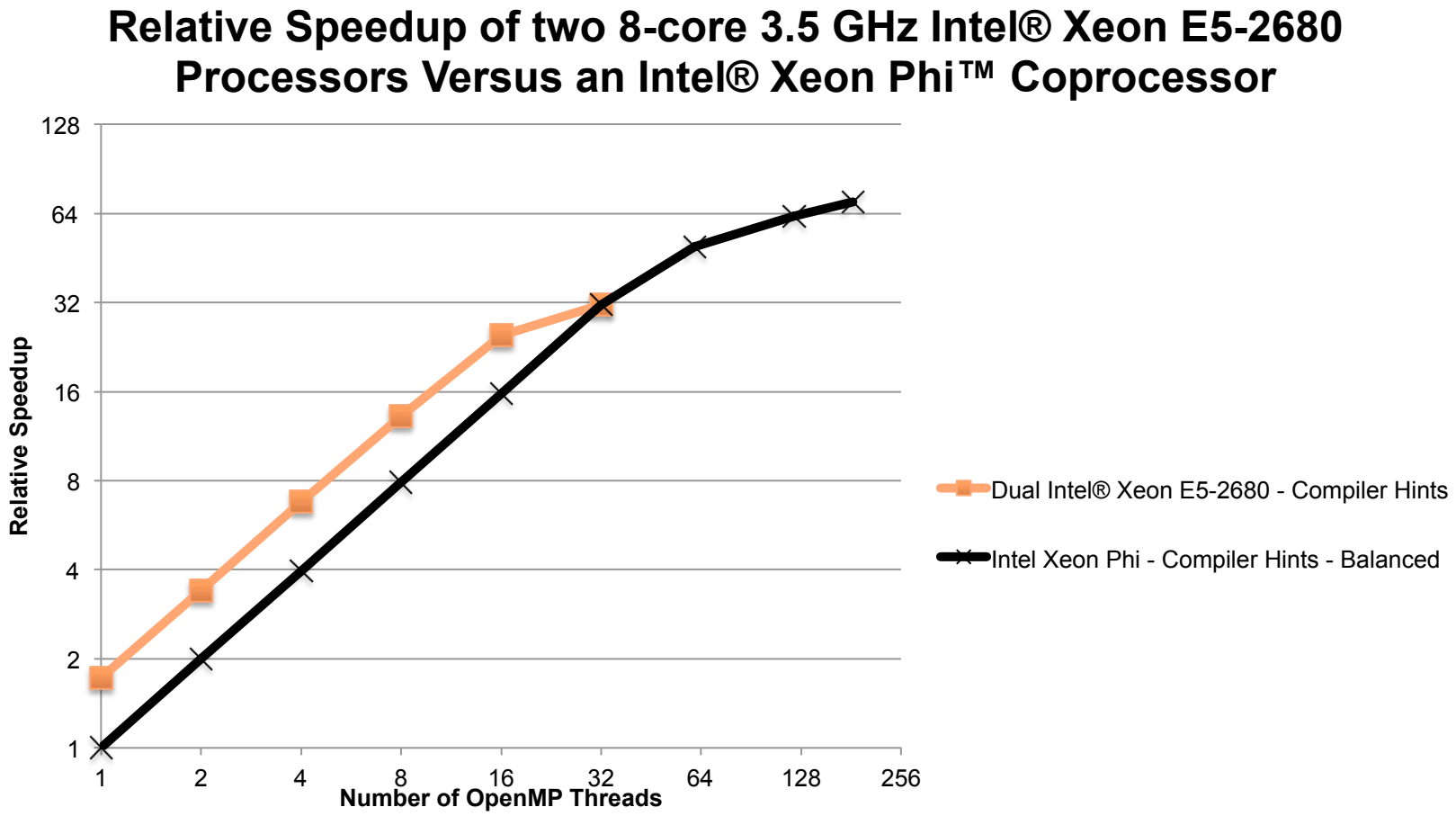
- **Set I — Loop Vectorization**
  - Stack variable pulled out of the loop
  - Class member turned into a regular structure
- **Set II — Data Access**
  - Arrays linearized using macros
  - Align data for more efficient access
- **Set III — Parallel Overhead**
  - Reduce the number of parallel sections
- **Set IV — Dependency**
  - Remove reduction from computational loop by saving value into a private variable
- **Set V — Precision**
  - Use medium precision for math function calls (-fimf-precision=medium)
- **Set VI — Precision**
  - Use single precision constants and intrinsics
- **Set VII — Compiler Hints**
  - Use #pragma SIMD instead of #pragma IVDEP

Optimization Results from the Model Boltzmann Equation Solver



Results were generated on a Pre-Production Intel® Xeon Phi™ coprocessor
with B0 HW and Beta SW
61 cores @ 1.09 GHz and 8 GB of GDDR5 RAM @ 2.75 GHz

Model Boltzmann Equation Solver Performance



Results were generated on a Pre-Production Intel® Xeon Phi™ coprocessor with B0 HW and Beta SW
61 cores @ 1.09 GHz and 8 GB of GDDR5 RAM @ 2.75 GHz

Requesting time on Beacon

- Fill out form at https://portal.nics.tennessee.edu/accounts/request for a director's discretionary account
  - Students should have their advisor make the request
- An abstract and justification as to why time should be granted is needed

Links/Contact Information

- More information about beacon can found at: http://www.jics.tennessee.edu/aace/beacon/
- More information about using/programming for Intel Xeon Phi can be found at: http://software.intel.com/en-us/mic-developer

Ryan Hulguin

ryan-hulguin@tennessee.edu