Accelerated Simulations of Cosmic Dust Heating Using the Intel Many Integrated Core Architecture

Andrey Vladimirov (avladim@stanford.edu)
Hansen Experimental Physics Laboratory, Stanford University

work with: Troy Porter, HEPL, Stanford

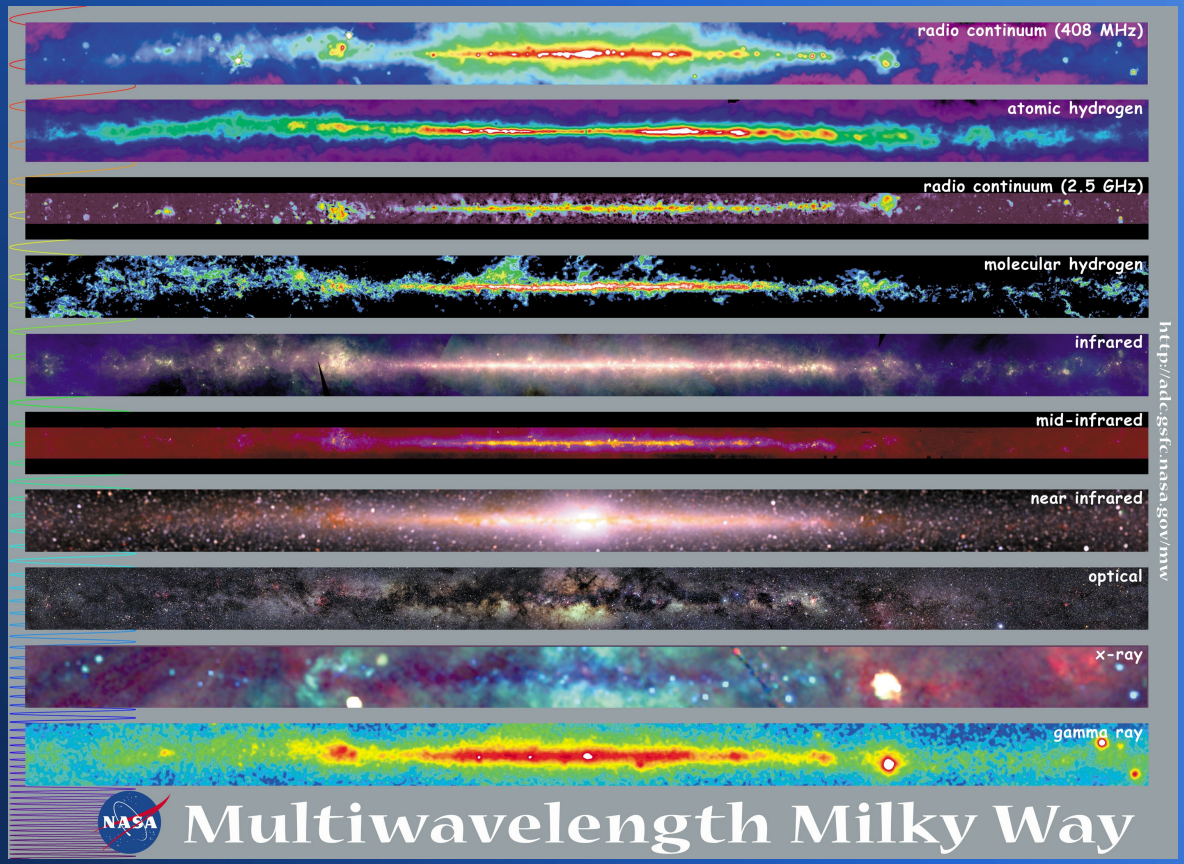
Talk given at UC Santa Cruz, Applied Mathematics and Statistics, June 6, 2013

Interstellar Radiation Field (ISRF)

- Most energy in the infrared (IR), optical and ultraviolet (UV) ranges.
- Sources: stars, dust.
- Processes: elastic scattering, absorption by dust, re-emission

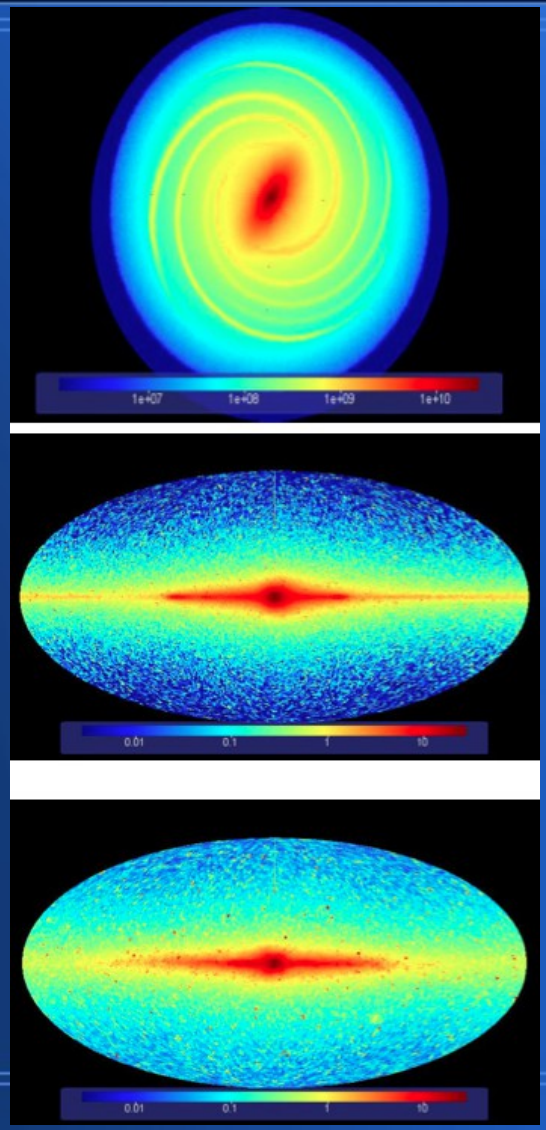
Precise ISRF modeling is necessary for the studies of:

- The interstellar medium (ISM)
- Cosmic ray (CR) propagation
- Extragalactic background light (EBL)
- Dust-obscured objects (for background elimination)



FRaNKIE Code

- "Fast RAdiative transfer Numerical Kode for Interstellar Emission"
- Monte Carlo transport of photons (broadband) in the Galaxy
- Physical input: distribution of stars and dust, microscopic processes
- Output: 3D ISRF density, sky maps + wavelength dependence
- References: Porter et al. (2008) ApJ, 682, 400, Ackermann et al. (2012) ApJ 750, 3
- Results used in the GALPROP code for cosmic ray transport



2.2 μm

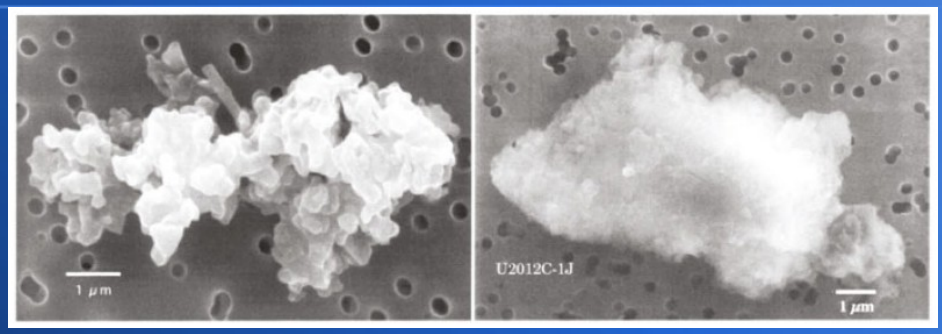
Simulation:
x=0
y=0
z=-20 kpc

Simulation:
x=8.5 kpc
y=0
z=0

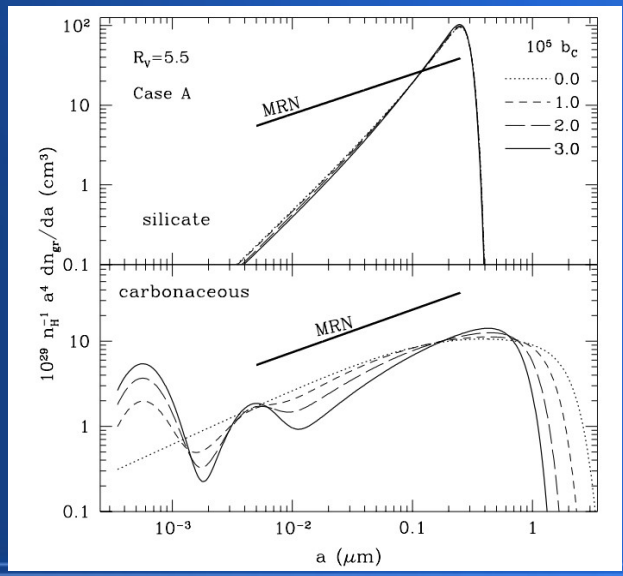
DIRBE data

Interstellar Dust

- Conglomerates of atoms (H, C, O, Si, etc.) and molecules, 0.5–1000 nm = 10^1-10^{10} particles
- Absorbs and scatters optical & UV light
- Optically heated dust re-emits energy in IR
- Important for H_2 production, star formation



SEM images of interplanetary dust
Jessberger et al. (2001)

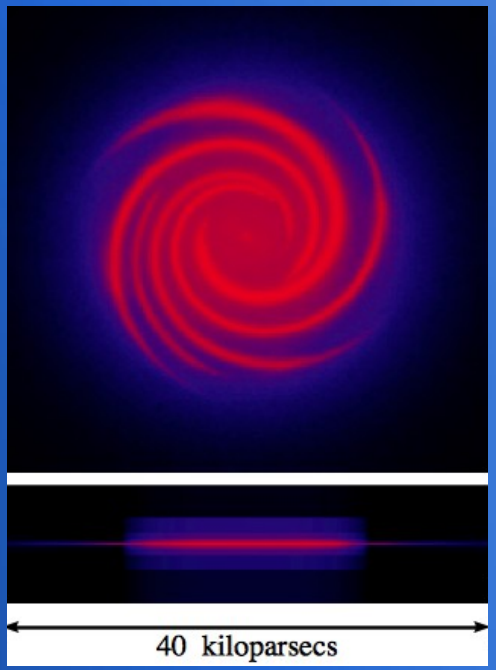


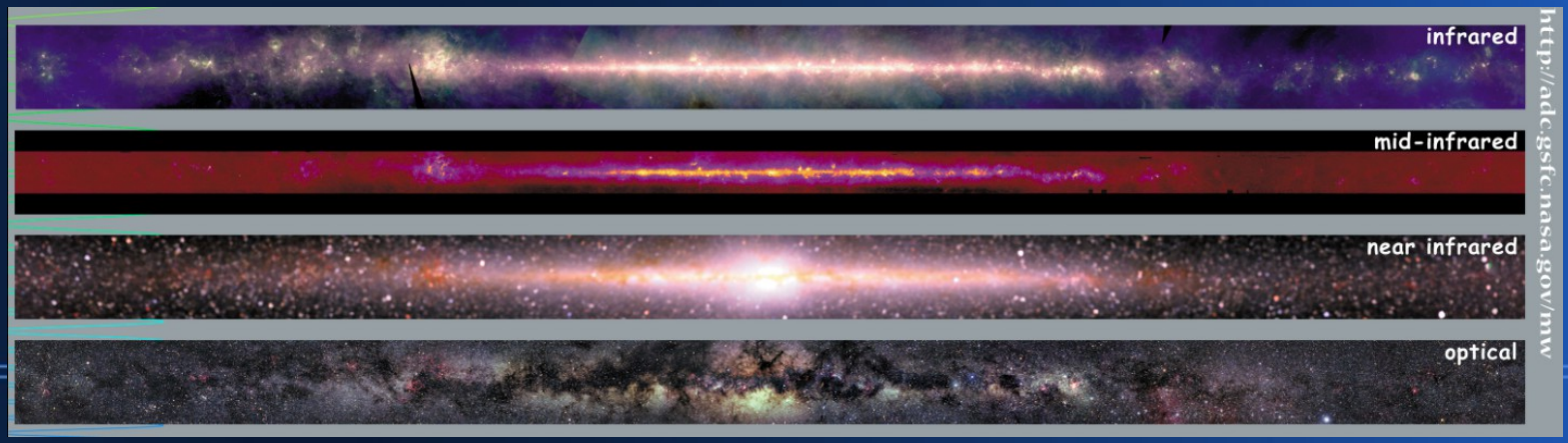
Size distribution of interstellar dust
Weingartner & Draine (2001)

Interstellar Dust

- Optical depth toward Galactic center >> 1 (significant attenuation, wavelength-dependent)
- Must be included in radiative transport (RT) simulations
- Dust heating and RT must be treated self-consistently (local dust heating by propagated photons + propagation of re-emitted IR photons)

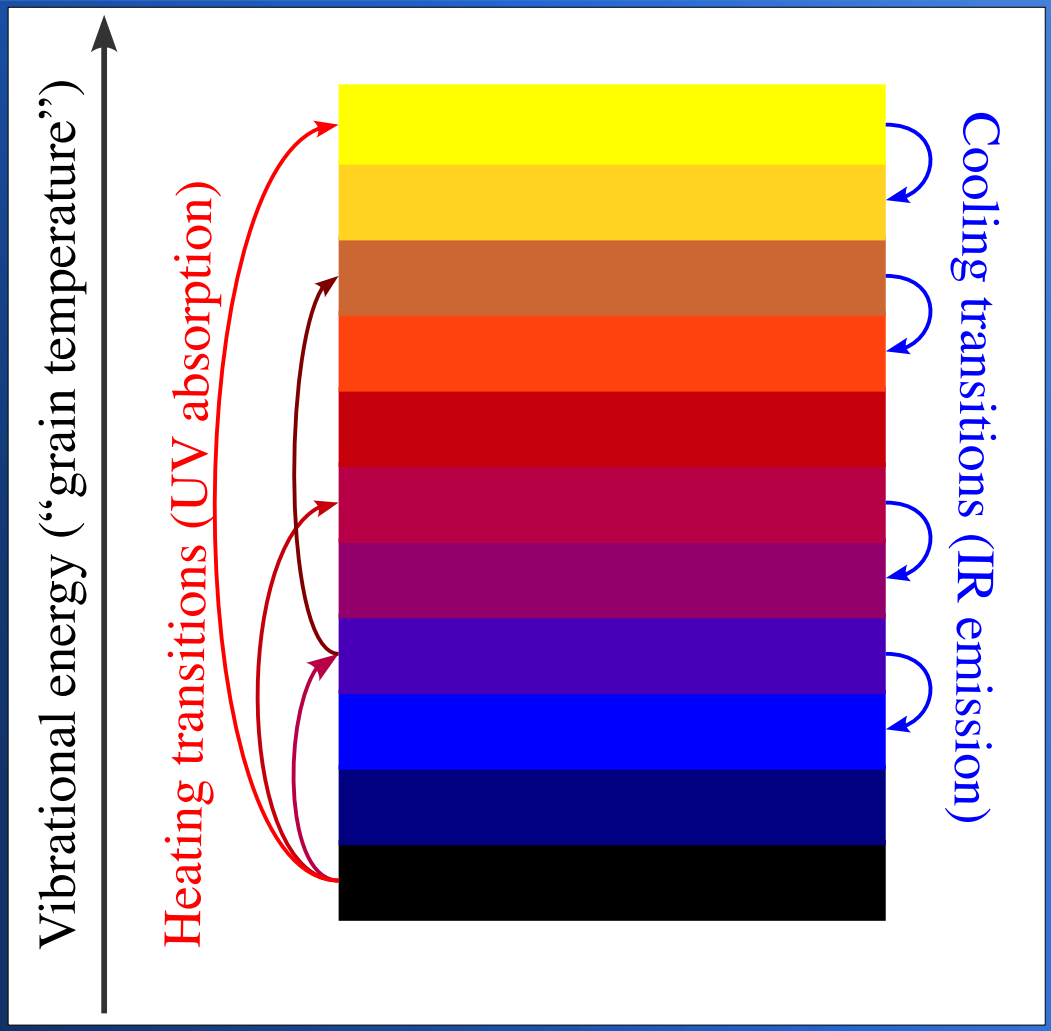
Dust luminosity computed by the FRaNKIE code





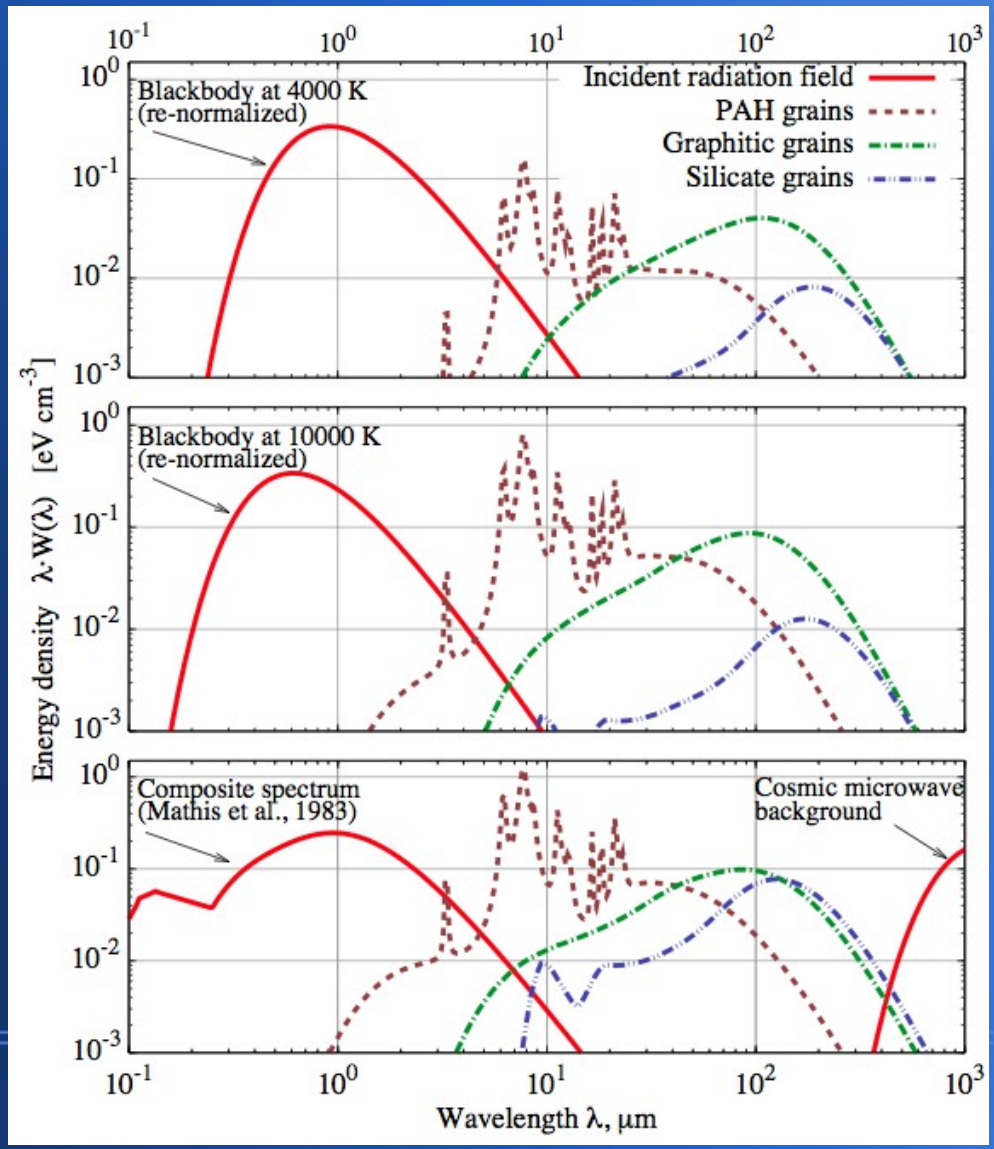
Stochastic Dust Grain Heating

- Large grains are heated by photon absorption attaining thermal equilibrium with the heating radiation field (characterizable by a single temperature, easy to model)
- For very small grains (≤0.1 μm), absorption and re-emission is stochastic (grains undergo "temperature" spikes, characterized by temperature distribution — evaluation computationally expensive)



Matrix Formalism for Stochastic Dust Emissivity Calculation

- Input data: incident electromagnetic radiation field
- Intermediate: “temperature” distribution of grains of all sizes
- Output: spectrum of re-emitted photons
- Carbonaceous, silicate and PAH grains (polycyclic aromatic hydrocarbons)
- Method and absorption cross sections: Draine et al. (2001), ApJ, 551, 807



Matrix Formalism for Stochastic Dust Emissivity Calculation

- Stage 1:

Interpolate (in log space) and convolve the incident RF with the photon absorption cross sections

$$T_{ul} = I(\lambda)\sigma(\lambda)\frac{\lambda^3 \Delta E_{ul}}{hc^2} \quad \text{for} \quad u > l.$$

$$I(\lambda)\sigma(\lambda) \equiv \Omega(\lambda)$$

$$\log\left[\frac{\Omega(\lambda)}{\Omega(\lambda_{j-1})}\right] = \frac{\log\left(\lambda/\lambda_{j-1}\right)}{\log\left(\lambda_j/\lambda_{j-1}\right)} \log\left[\frac{\Omega(\lambda_j)}{\Omega(\lambda_{j-1})}\right]$$

transcendental operations

- Stage 2:

form and solve a quasi-triangular system of linear algebraic equations for the "temperature" distribution

$$\sum_{j\neq i} T_{ij}P_j - \sum_{j\neq i} T_{ji}P_i = 0$$

$$T_{ij} = 0, \quad \text{if} \quad i < j-1$$

$$B_{fj} = \sum_{k=f}^{M} T_{kj} \quad (f > j)$$

$$X_f = \frac{1}{T_{(f-1)f}} \sum_{j=0}^{f-1} B_{fj}X_j$$

sparse memory access

- Stage 3:

convolve the "temperature" distribution with the grain size distribution and emissivity function

$$\nu F_a(\nu) = \sigma(\nu) \sum_{i=0}^{M} P_i(a)\Lambda(\nu, E_i)$$

$$\Lambda(\nu, E_i) = \begin{cases} 0, & \text{if} \quad E_i < h\nu, \\ \dfrac{2h\nu^4}{c^2} \dfrac{P_i}{\exp(h\nu/kT_i) - 1} \end{cases}$$

$$\nu F(\nu) = \int_{a_{\min}}^{a_{\max}} \nu F_a(\nu) Q(a) da$$

dense linear algebra

Computational Challenge

- What: Constrain the geometrical/compositional parameters of the distribution of light sources (stars) in the Galaxy.
  How: Bayesian analysis of sky survey data. The analysis fits the results of the FRaNKIE code within a parameter space to the observational data.
- Need of order 10^5 FRaNKIE evaluations with 10^5 cells
- Difficulty: stochastic dust heating must be computed for every simulation cell in the Galaxy consistently with the RF.
- Bottlenecked by stochastic emissivity calculation: 60 ms per cell on a modern 16-core Intel Xeon E5 server.
- Translates to 20 machine-years for the calculation — too much.

HEATCODE (HEterogeneous Architecture library for sTochastic COsmic Dust Emissivity)

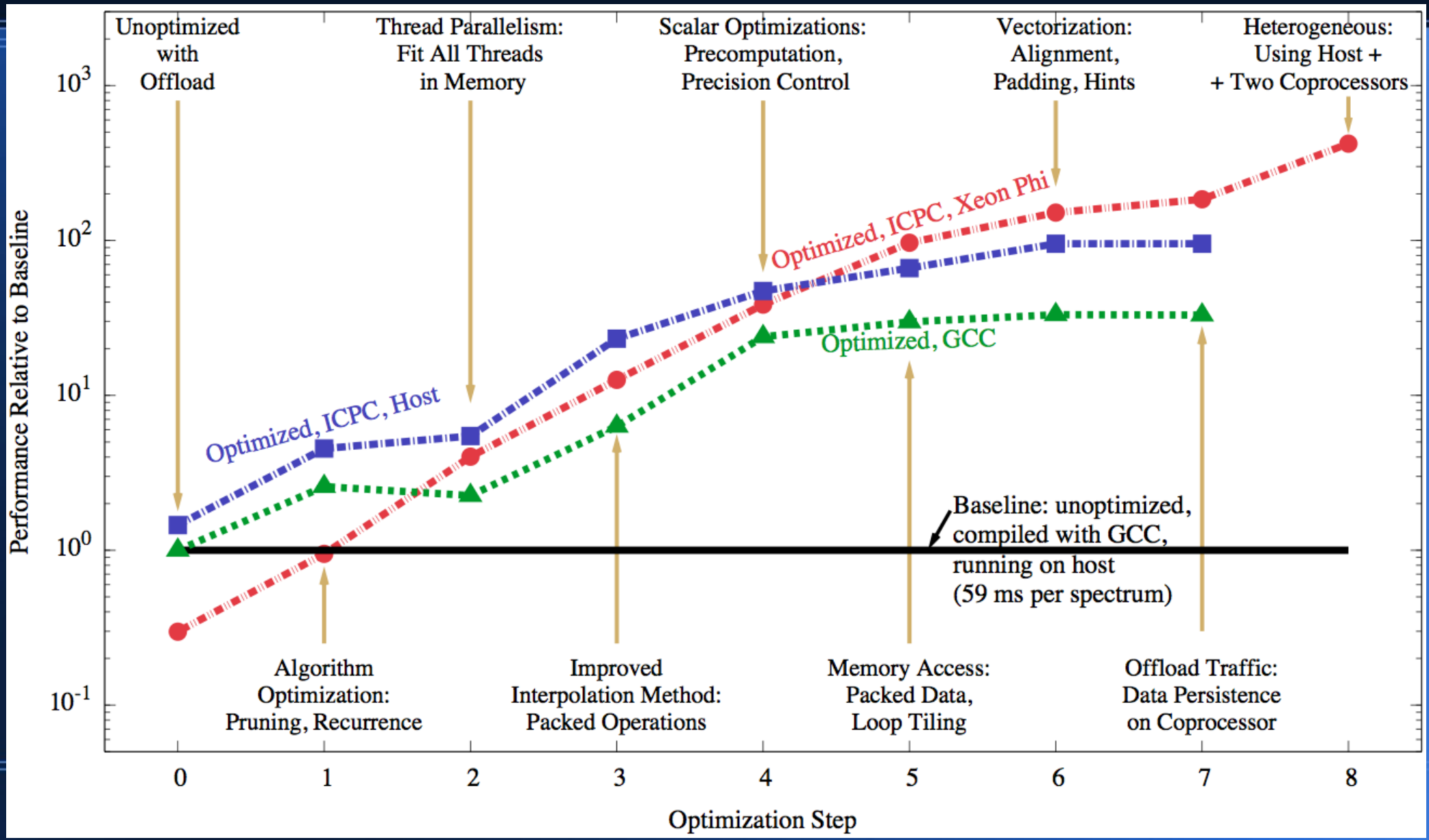
What we need:

- Optimize the stochastic emissivity calculation to employ available compute resources more efficiently
- Use high performance computing accelerators if available
- Operate on a wide range of computing platforms for public distribution

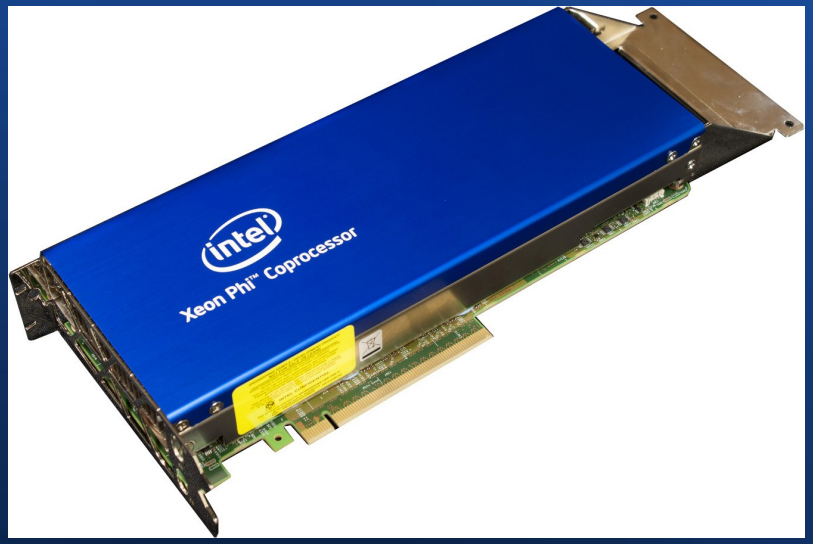
Our solution:

- A new library called HEATCODE (started from unoptimized implementation of the Draine et al. matrix formalism)
- Optimized for Intel Xeon multi-core architecture, suitable for any CPU
- After optimization, 100x more efficient on the same hardware (2x Intel Xeon E5-2680 CPUs)
- Additional 4.5x with two Intel Xeon Phi 5110P coprocessors
- Support for GPGPUs can easily be added

Preview of Results



Intel MIC Architecture



- Intel Xeon Phi coprocessor: to accelerate applications that have reached the parallel scaling limits of Intel Xeon processors
- PCIe v 2.0 device
- Nominal: 245-300 W, ~1 TFLOP/s in double precision, 354 GB/s bandwidth
- 60 dual-issue in-order cores @1 GHz with 4-way hyper-threading (240 logical cores)
- 8 GB onboard GDDR5
- 512-bit SIMD instructions

- Runs its own Linux OS
- Hosts a virtual file system
- IP-addressable
- The same languages (C, C++, Fortran), tools (compilers, profilers) and optimization methods as general-purpose Intel CPUs

Programming Models for the MIC Architecture

Native Model
application runs directly on coprocessor

- Use coprocessor as a compute node
or
- MIC-only MPI
or
- Heterogeneous MPI

Offload Models
application runs on host, communicates w/coprocessor

Explicit offload (pragma-based)

Virtual-shared Memory

- Move parallel tasks to coprocessor
or
- Share work between host and MIC
and/or
- MPI with offload

OpenCL
Launched May'13

Explicit Offload Model

- HEATCODE uses the explicit offload model
- Simplicity, compatibility with the CUDA approach, fall-back to host

```
#pragma  offload_attribute (push, target(mic))
    void InterpolateWeightedRF(const int wlBins, float* RF, ...) {
      /* ... one implementation for both the host and the MIC */
    }
#pragma  offload_attribute(pop)

void CalculateTransientEmissivity( ... ) {
  RF = (float*)malloc(wlBins*nSpectra*sizeof(float));
#pragma offload target(mic) inout(RF: length(wlBins*nSpectra))
  { /* run on the coprocessor if available, otherwise on host */
    InterpolateWeightedRF(wlBins,  RF, ...);
    ...
  }
}
```

Threading Models for the MIC Architecture

OpenMP
v 4.0

MPI
Processes directly on MIC *or*
Hybrid MPI+OpenMP *or*
Offload to MIC + OpenMP

Intel Cilk Plus, TBB
"tasks, not threads"

Pthreads
For the fearless

- HEATCODE uses OpenMP
- Parallelization across multiple incident radiation spectra
- Each spectrum processed serially
- Same code for host and coprocessor
- Must process >>240 spectra to be efficient on coprocessor

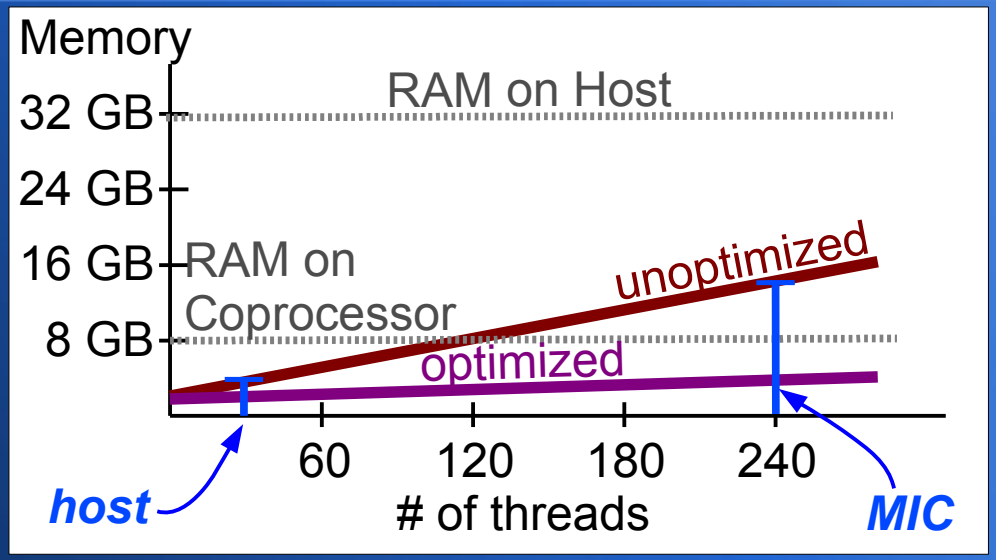
```
#pragma offload target(mic)…
{
  #pragma omp parallel for schedule(dynamic)
  for (int iRF = 0; i < nSpectra; i++) {
    InterpolateWeightedRF(wlBins,  iRF, ...);
    CalculateTemperatureDistribution(...);
    ComputeEmissivity (...);
  }
}
```

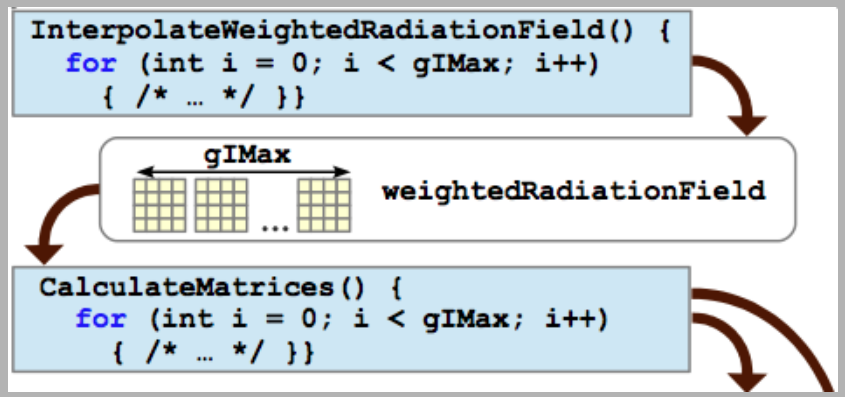
Threading Optimization for the MIC Architecture

Qualitatively, MIC requires the same optimizations as multi-core CPUs:	HOW TO ACHIEVE:
• Avoid synchronization	➔ Use reduction instead of mutexes
• Eliminate false sharing	➔ Padding, thread-private containers
• Choose optimal scheduling mode	➔ Load balance vs sched. overhead
• Avoid serial operations	➔ Parallelize algorithm, use static memory allocation: malloc() is serial
Quantitatively, more parallelism: MIC applications must scale to 240 threads	
• Increase the the number of parallel tasks to keep all threads busy	➔ Collapse nested loops or rethink parallelization strategy
• Reduce per-thread memory overhead if problem does not fit in memory	➔ Change algorithm or use nested parallelism within tasks
• Set appropriate thread affinity	➔ In OpenMP, use KMP_AFFINITY

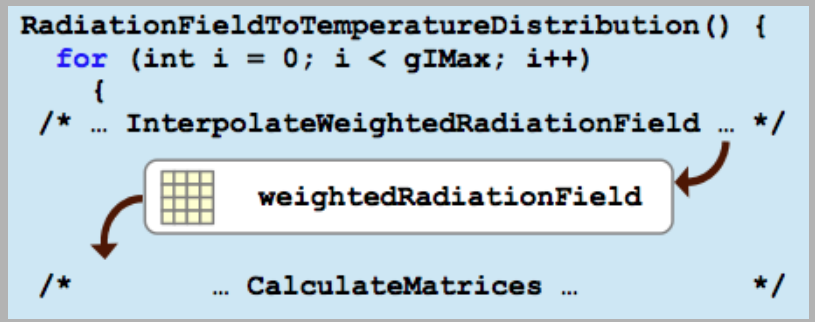
Threading Issues in HEATCODE

- Unoptimized code used scratch space liberally. This is OK on host, but a limitation on MIC.
- Before optimization, we had to reduce the # of threads on the MIC to fit in 8 GB memory
- Optimization: reducing per-thread memory overhead by loop fusion





Unoptimized



Optimized (loop fusion)

Why Care about Vectorization

- In older architectures, SIMD registers were narrow (64-, 128-bit), and scientists often must use double precision (64-bit) =>
  Without vectorization, one loses up to 2x — often not significant enough
- On the MIC architecture, 512-bit vectors =>
  Without vectorization, one pays a 8x penalty in double precision (16x in single precision)

Instruction Set	Year and Intel Processor	Vector registers
MMX	1997, Pentium	64-bit
SSE	1999, Pentium III	128-bit
SSE2	2001, Pentium 4	128-bit
SSE3–SSE4.2	2004 – 2009	128-bit
AVX	2011, Sandy Bridge	256-bit
AVX2	2013, (future) Haswell	256-bit
IMCI	2012, Knights Corner	512-bit

Table credit: “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors”, Colfax

Optimization for the Intel MIC Architecture: (Automatic) Vectorization

- Intel Xeon E5 (Sandy Bridge) architecture: 256-bit AVX vector instructions, legacy SSE, MMX...
- Intel Xeon Phi (Knights Corner) architecture: 512-bit IMCI instructions
- Xeon Phi does not understand AVX or SSE
- Explicit SIMD coding is possible, but automatic vectorization is more portable *and* flexible & efficient

```
/* Fragment of the solution for
temperature distribution P_i
from B_ij in the HEATCODE library
with automatic vectorization */
#pragma vector aligned
for (int i = 0; i < tempBins; ++i)
  sum += bMatrix[f*tempBins + i]*x[i];
```

```
avladim@dublin$ # Auto-vectorization report
avladim@dublin$ icpc -c -vec-report3 \
        TransientHeatingFunctionsXeonPhi.cc
...
TransientHeatingFunctionsXeonPhi.cc(199):
     (col. 6) remark: LOOP WAS VECTORIZED.
...
TransientHeatingFunctionsXeonPhi.cc(199):
     (col. 6) remark: *MIC* LOOP WAS
                                VECTORIZED.
TransientHeatingFunctionsXeonPhi.cc(199):
     (col. 6) remark: *MIC* REMAINDER LOOP
                               WAS VECTORIZED.
...
```

Optimizing Vectorization

- Contiguous memory access works best
- Align arrays on 64-byte boundary
- The compiler may need hints (pragma directives)
- Avoid type conversions
- Index notation better than pointer references
- No need to precompute array indices

```
/* Recurrent calculation of B_ij
   From T_ij in the HEATCODE library

Programmer guarantees data alignment,
so the compiler does not have to
implement runtime alignment checks.

Loop count estimate helps the compiler
to pick the optimal vectorization
strategy. */
#pragma vector aligned
#pragma loop count min(16)
for (int i = 0; i < iMax; ++i) {
  rSum[i] += bMatrix[f*tempBins + i];
  bMatrix[f*tempBins + i] = rSum[i];
}
```

Optimizing Vectorization

- 512 bits vector fits 16 single precision FP numbers
- HEATCODE: padded loop bounds to a multiple of 16 iterations

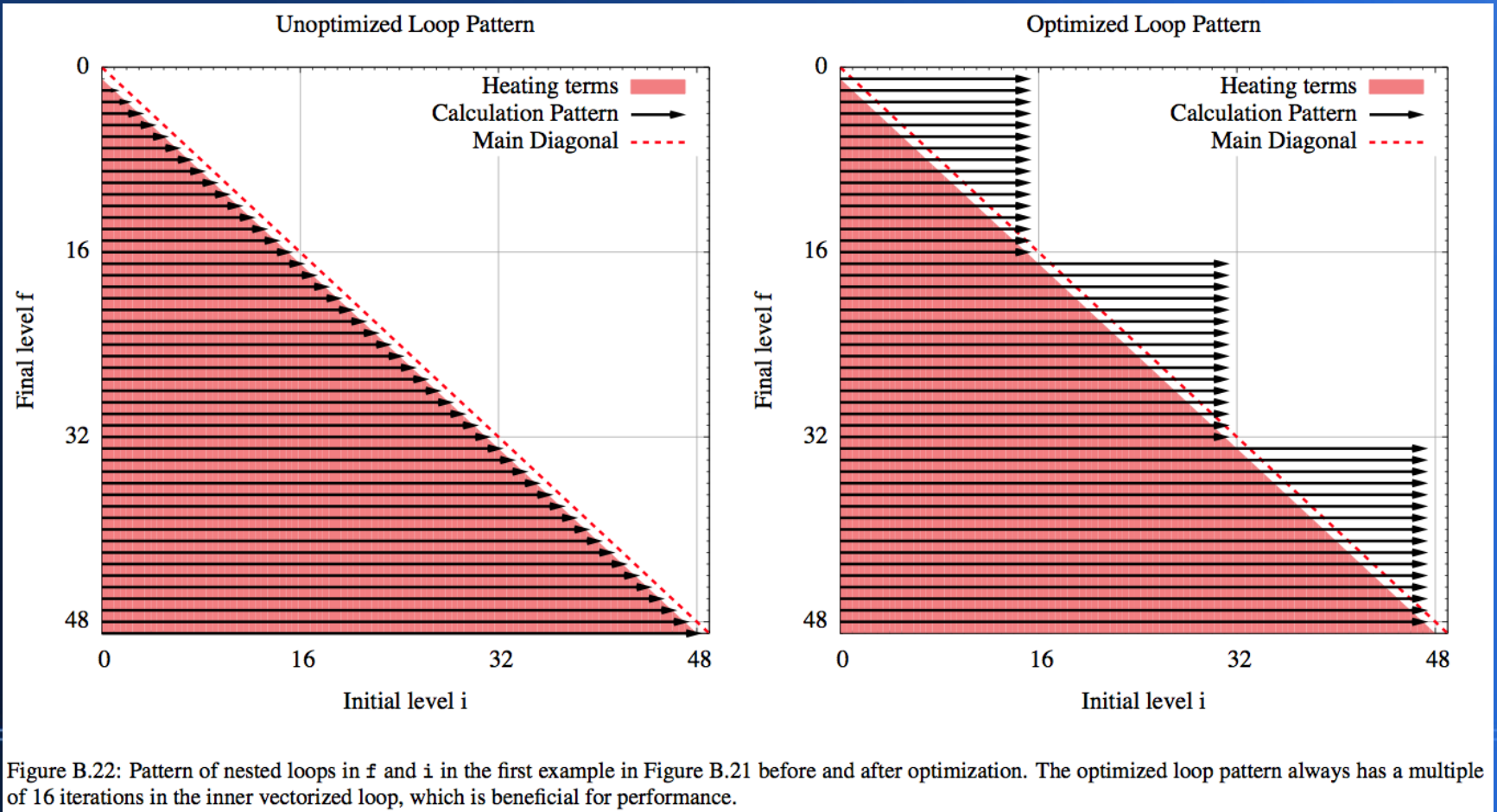


Figure B.22: Pattern of nested loops in `f` and `i` in the first example in Figure B.21 before and after optimization. The optimized loop pattern always has a multiple of 16 iterations in the inner vectorized loop, which is beneficial for performance.

Optimization for the Intel MIC Architecture: Cache Traffic

- MIC architecture has a similar cache structure to a multi-core CPU
- To minimize cache misses, maximize data locality and re-use
- This is usually done by changing the order of memory accesses:
  - Fusion of loops
  - Nested loop interchange (permutation)
  - Loop tiling (blocking)
  - Cache-oblivious recursion

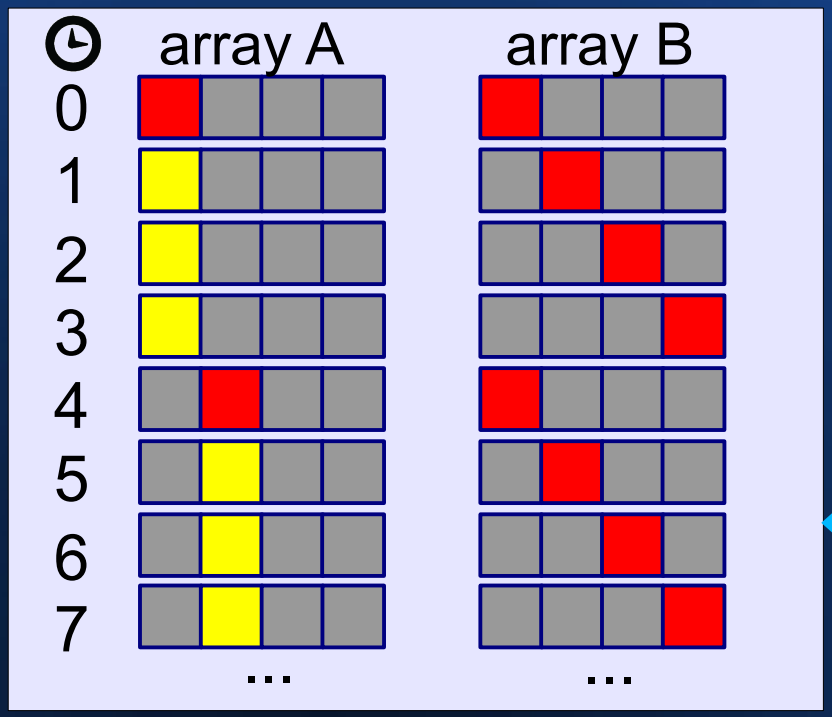
Cache line size	64B
L1 size	32KB data, 32KB code
L1 latency	1 cycle
L2 size	512KB
L2 ways	8
L2 latency	11 cycles
Memory → L2 prefetching	hardware and software
L2 → L1 prefetching	software only
Translation Lookaside Buffer(TLB)	64 pages of size 4KB (256KB coverage)
coverage options (L1, data)	8 pages of size 2MB (16MB coverage)

Table credit: Colfax

Loop Tiling Explained

```
/* Nested loops without tiling.
Array B[] does not fit into cache */
for (int i = 0; i < iMax; ++i)
 for (int j = 0; j < jMax; ++j)
  PerformWork(A[i], B[j]);
```

```
/* Tiled nested loops */
for (int ii = 0; ii < iMax; ii += T)
 for (int j = 0; j < jMax; ++j)
  for (int i = ii; i < ii+T; ++i)
   PerformWork(A[i], B[j]);
```



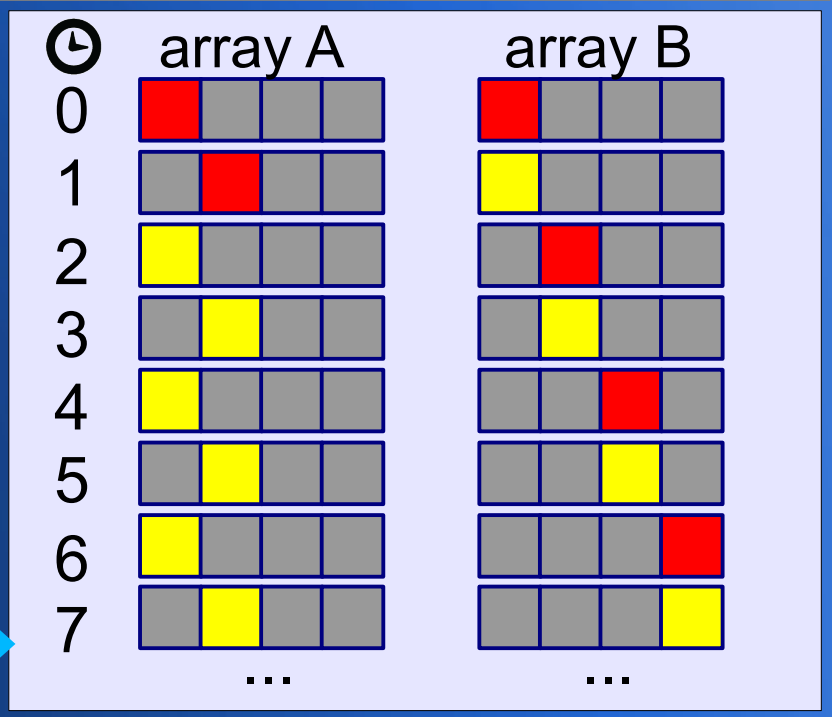
Example:
tile size T=2
cache size=3

Cache Misses

Cache Hits

Without Tiling

With Tiling



Cache Hit Rate = 6/16

Cache Hit Rate = 10/16

SLOWER

FASTER

Cache Traffic Optimization in HEATCODE

```
/* Convolution of temperature distr.
with emissivity function in the
HEATCODE library (UNOPTIMIZED) */
for (int i = 0; i < wlBins; ++i) {
 float sum = 0.0f;
 for (int j = 0; j < gIMax; ++j) {
  const float scaling = ...[i,j];

  float result = 0.0f;
  for (int k = 0; k < tempBins; ++k)
   result +=
            planck[i*tempBins + k]*
       distribution[j*tempBins + k];

  sum += result*scaling;
 }
 trans[i] = sum*wavelength[i]*units;
}
```

↑ "Before"

"After" →

```
/* OPTIMIZED w/double loop tiling */
for (int jj=0; jj<gIMax; jj+=jTile) {
for (int ii=0; ii<wlBins; ii+=iTile){
 float  result[iTile*jTile];
 for (int c = 0; c<iTile*jTile; c++)
  result[c] = 0.0f;

#pragma simd
 for (int k = 0; k < tempBins; ++k)
  for (int c = 0; c < iTile; c++) {

  result[(0)*iTile + c] +=
      distribution[(jj+0)*tempBins+k]*
          planck[(ii+c)*tempBins+k];
    result[(1)*iTile + c] +=
      distribution[(jj+1)*tempBins+k]*
         planck[(ii+c)*tempBins+k];
    result[(2)*iTile + c] +=
      distribution[(jj+2)*tempBins+k]*
         planck[(ii+c)*tempBins+k];
    result[(3)*iTile + c] +=
      distribution[(jj+3)*tempBins+k]*
             planck[(ii+c)*tempBins+k];
   }
...
```

Memory Traffic Optimization in HEATCODE

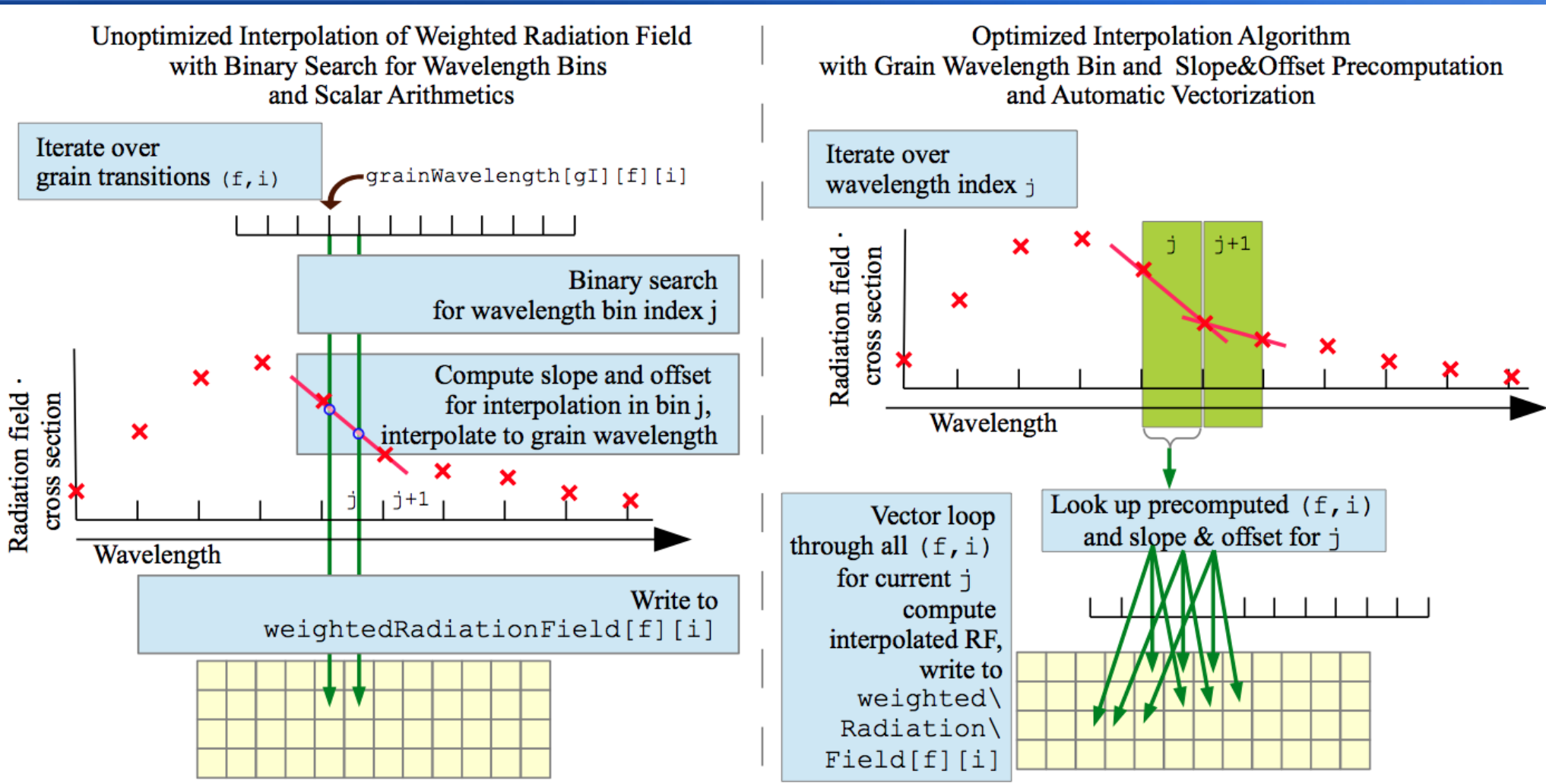
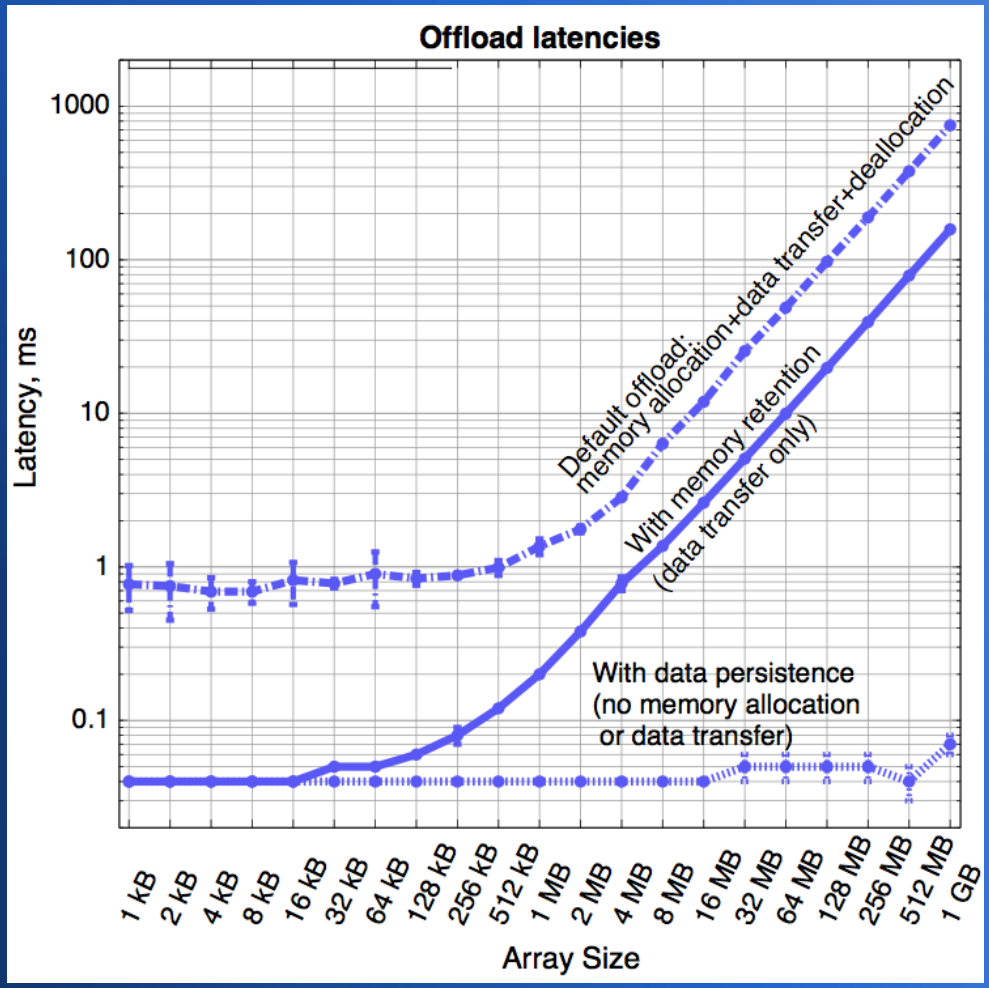


Figure B.12: Schematic interpolation algorithm before and after optimization.

Optimization for the Intel MIC Architecture: Offload Data Traffic

- Upon offload, data are transferred across the PCIe bus: ~6 GB/s
- Whenever possible, retain data on coprocessor between offloads
- Memory allocation on coprocessor is slow (a serial operation): ~1 GB/s
- Whenever possible, retain allocated memory on coprocessor between offloads



Optimization for the Intel MIC Architecture: Offload Data Traffic

```
/* Offload pragma in HEATCODE,
data marshaling directives */
#pragma offload target(mic)
…
in(rfArray :                    \
     length(n*rfBins))          \
out(emissivityArray :           \
     length(n*rfBins))          \
…
in(absorptionCrossSection : \
     length(gIMax*wlBins))
}
```

```
/* Offload pragma in HEATCODE, optimized
using data and memory persistence */
#pragma offload target(mic:iDevice)
…
in(rfArray :                      \
 length(n*rfBins) alloc_if(0) free_if(0)) \
out(emissivityArray :             \
 length(n*rfBins) alloc_if(0) free_if(0)) \
…
in(absorptionCrossSection : \
 length(0) alloc_if(0) free_if(0))
}
```

__Unoptimized:__

For every offload,

- Sending/receiving input & output
- Sending/receiving model data
- Allocating/deallocating memory

__Optimized:__

For every offload,

- Sending/receiving input & output
- Re-using offloaded model data
- Re-using allocated memory
- Requires initial offload and cleanup (not shown here)

Optimization for the Intel MIC Architecture: Heterogeneous Work Sharing

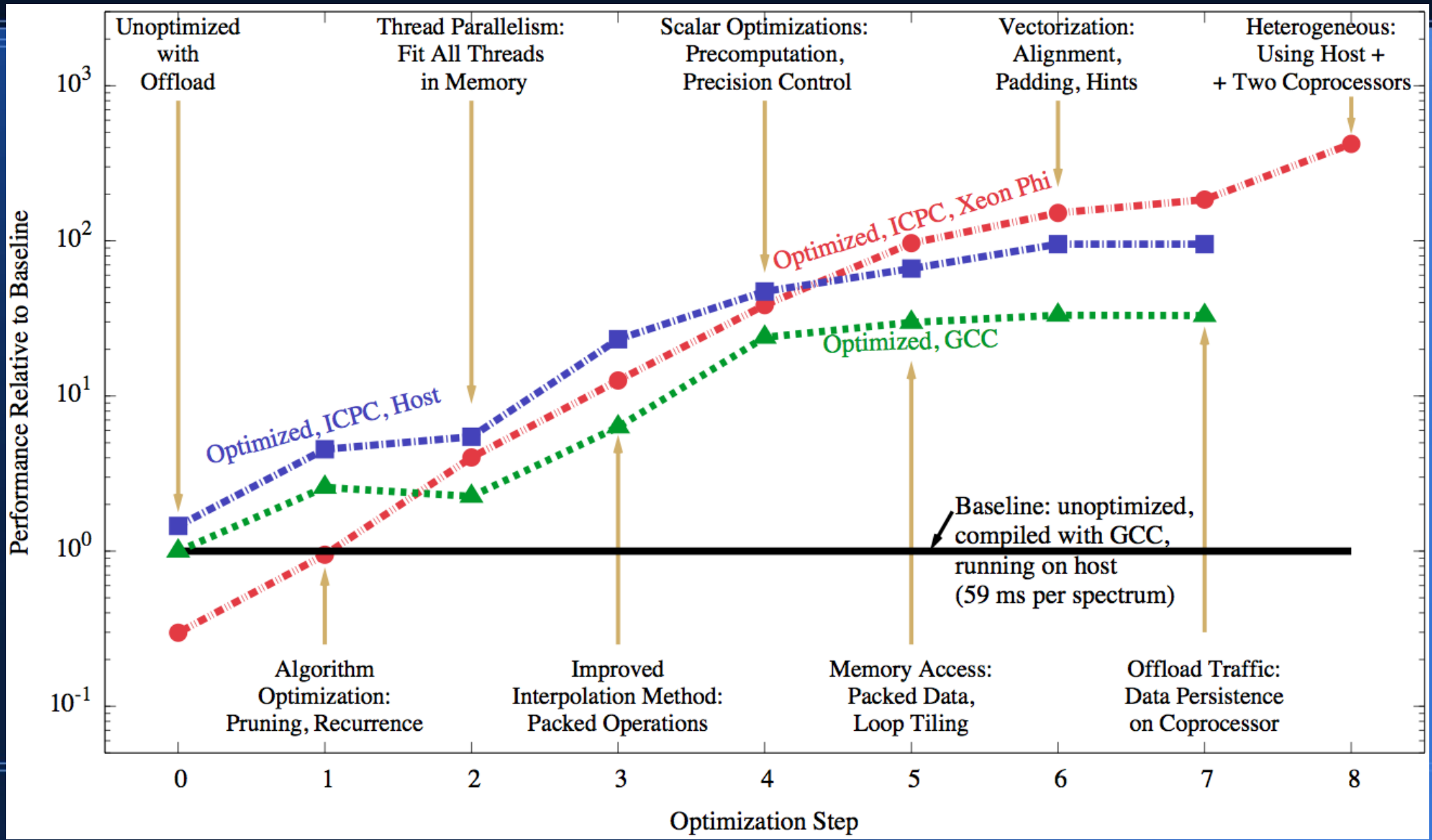
- In our compute nodes: a 2-socket Xeon CPU + two Xeon Phi
- We would like to use all available compute power
- One Xeon Phi is ~2x faster than CPU → can't split work evenly
- Must split up work into chunks and use “boss-worker” scheduling
- Easy solution using the OpenMP scheduler:

```
#pragma omp parallel for n_threads(3) schedule(dynamic,1)
for (int i = 0; i < nChunks; i++) {
  int iDevice = omp_get_thread_num();
  #pragma offload target(mic: iDevice) if (iDevice > 0)
  ...
```

Other Optimization Considerations for HEATCODE

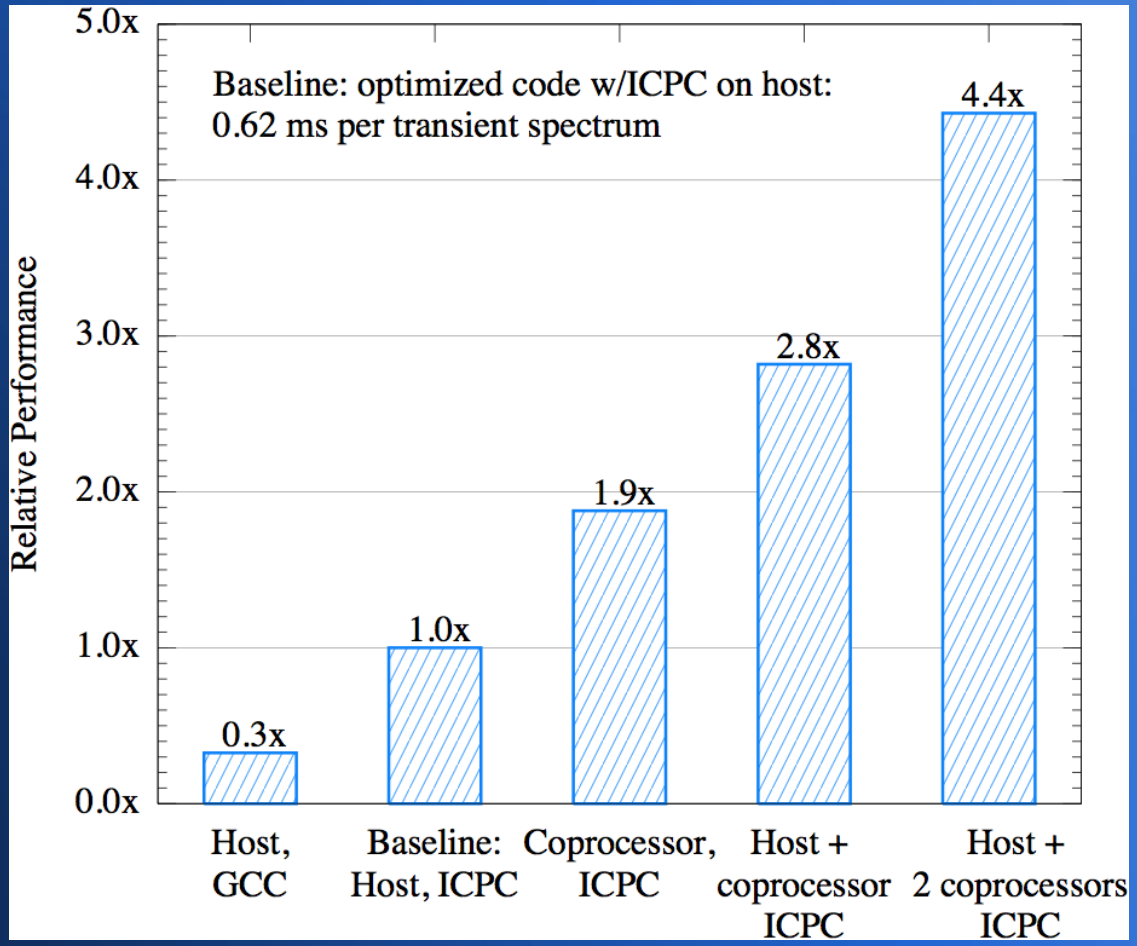
- Precomputation often helps; however, sometimes reading a precomputed value is more expensive than computing on the fly
- Avoiding type conversion:
  - consistently use single or double precision variables
  - specify single precision constants as “0.0f”, “1.0f”, etc.
  - use single-precision math functions: sinf(), expf(), fabsf()...
- Use base 2 logarithms and exponentials: exp2f(), log2f()
- Use -fimf-domain-exclusion=15 if do not need denormals, NaNs...
- Set MIC_USE_2MB_BUFFERS to improve TLB traffic
- Potentially: use the Intel Math Kernel Library (MKL) for matrix multiplication. MKL has a number of standard routines (xGEMM, FFT, random numbers, etc) optimized for Intel Xeon Phi coprocessors.

Performance Benchmarks: Optimization



Performance Benchmarks: Platforms

- HEATCODE on one Xeon Phi vs two Xeon E5-2680s: 1.9x speedup
- Why against *two* CPUs? Same power ~ 250 W
- Synthetic benchmarks: SGEMM 2.9x, LINPACK 2.6x, STREAM 2.2x
- Before optimization, the (parallel) code was 100x slower on host and 400x slower on MIC



Developer Experience: Programming

- Initial porting is trivial with native execution: “-mmic” compiler argument. Works for open source packages and autotools as well. Example:

```
./configure  --prefix=~/mic/xerces-c --without-curl --enable-transcoder-iconv \
             CC="icc" CXX="icpc" CFLAGS="-mmic" CXXFLAGS="-mmic" --host=x86_64
```

- Explicit offload model is straightforward, but must pack all data into arrays
- Initially, the ported code was miserable on the coprocessor. However, it meant that it was not doing very well on the host, either.
- Optimizations for coprocessor lead to better performance on the host, and vice-versa. Coprocessor/host performance ratio is a measure of efficiency.
- Areas of optimization: thread scalability, vectorization, scalar efficiency, cache traffic, communication with coprocessor.
- If don't know where to optimize, use VTune.

Developer Experience: Development Tools

- Must use Intel compilers ($500-1500 single user academic license)
- Intel C++ compiler beats GCC by 3x (HEATCODE on the host)
- VTune: performance analysis with hardware event sampling. Works on Intel CPUs and Xeon Phi. The greatest thing since sliced bread. Can find hotspots down to a single line of code.
- Debugger is available, but “printf debugging” works, too: console output from the coprocessor is piped to host
- Intel MKL has a lot of optimized routines for Xeon Phi. Binaries are redistributable

Intel Vtune Parallel Amplifier XE

General Exploration - Knights Corner Platform

Identify where microarchitectural issues affect the performance of your application. Press F1 for more details.

- ☑ Analyze general cache usage
- ☑ Analyze vectorization usage
- ☑ Analyze TLB misses
- ☑ Analyze additional L2 cache events

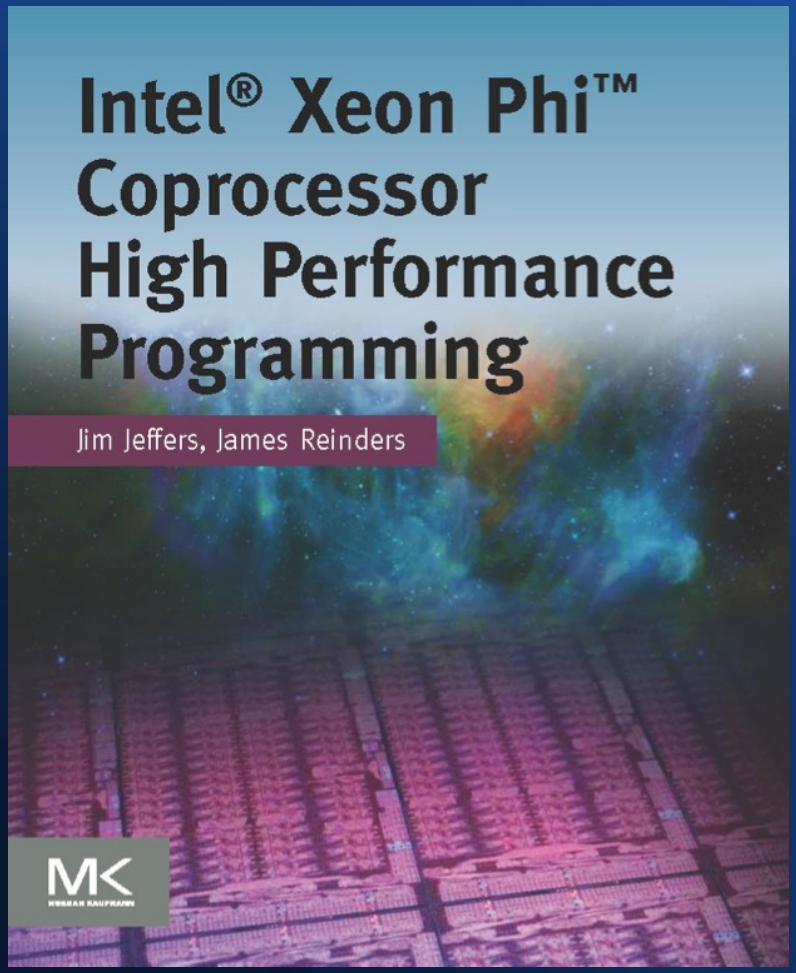
Function / Call Stack	CPU Time
thXeonPhi::RadiationFieldToTemperatureDistr	659.011s
thXeonPhi::CalculateEmissivity	202.414s
__intel_lrb_memset	124.030s
__kmp_wait_sleep	79.249s
__kmp_static_yield	46.179s
__kmp_yield	5.722s

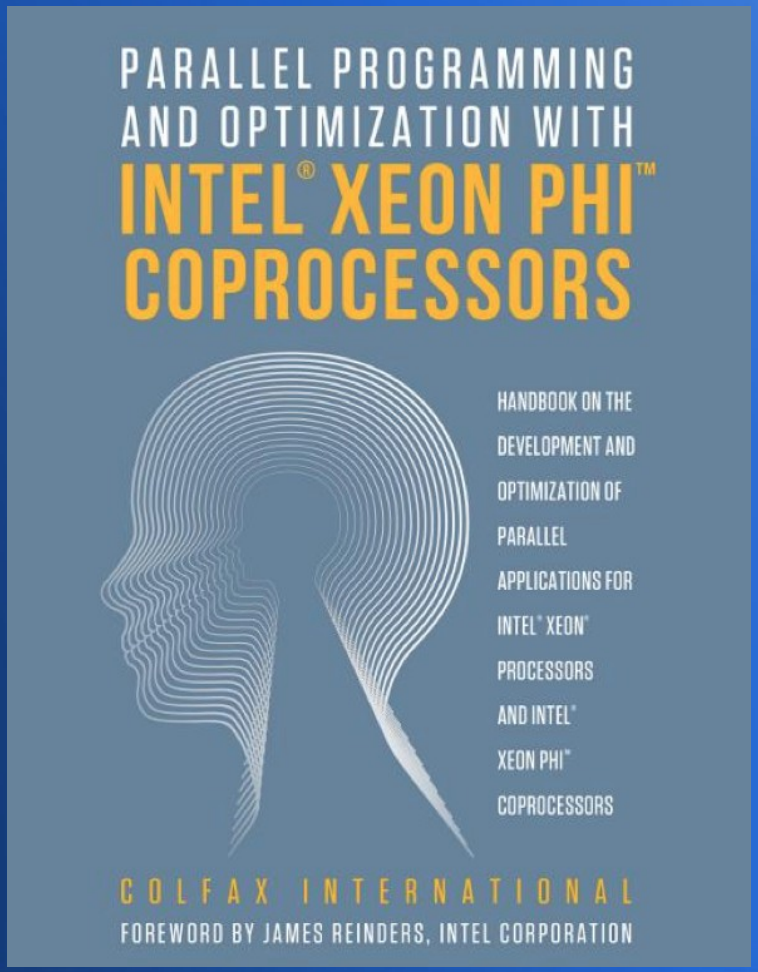
```
239   #ifdef HAVE_ICC                                          
240   #pragma simd reduction(+: sum)                           
241   #pragma vector aligned                                   
242   #endif                                                   
243         for (int i = 0; i < tempBins; ++i)          8.850s
244           sum += bMatrix[f*tempBins + i]*x[i];     70.599s
245                                                            
246         // rTransientMatrixOverDiagonal contains         
247         // (or zeroes if enthalpyDelta == 0, whi         
248         x[f] = sum*rTransientMatrixOverDiagonal[     4.325s
```

Developer Experience: Optimization

- Is it easy to port applications to the MIC architecture? Yes
  Do I get accelerated performance out of the box? Likely, No
- Same code on the host and the MIC? True
  Same optimization for the host and the MIC? In many cases, true
- For HEATCODE, optimization involved:
  - Loop fusion to reduce memory footprint, improve data locality
  - Floating point precision consistency (constants, variables, functions)
  - Strength reduction & precomputation in common expressions
  - Nested loop permutation and tiling to improve cache traffic
  - SIMD loop bounds and data alignment: pad to a multiple of 64 bytes
  - Vectorization tuning w/pragmas (loop count, aligned notice, …)
  - Data & memory persistence on the coprocessor (PCIe traffic)

Learning Resources on the Intel MIC Architecture Programming

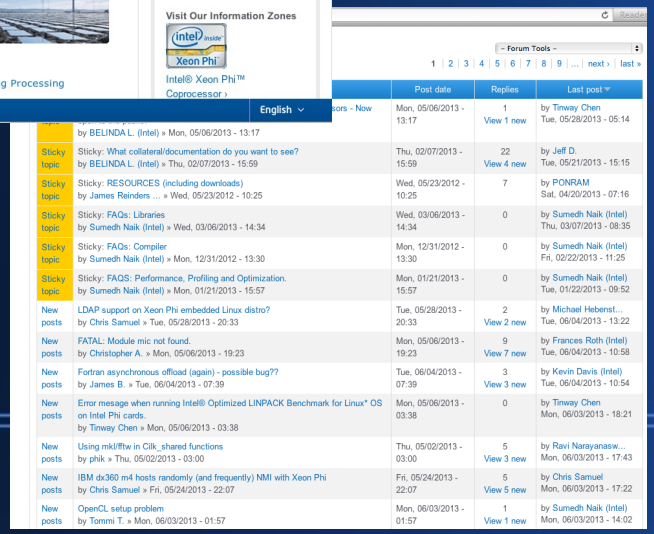




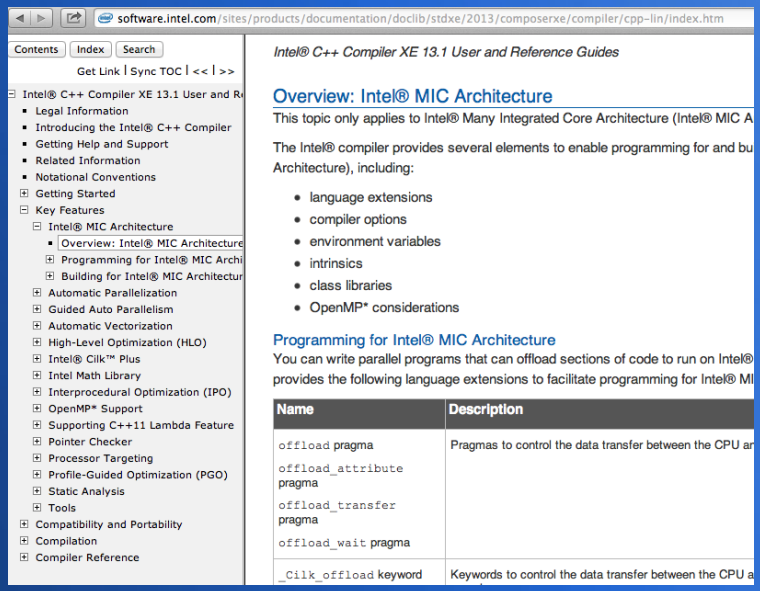
on Amazon.com: Jeffers & Reinders http://colfax-intl.com/xeonphi/

Learning Resources on the Intel MIC Architecture Programming

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



Intel MIC forum



Intel C/C++ Compiler XE 13.1 User and Reference Guide

Summary

- HEATCODE — a new library for fast calculation of stochastic cosmic dust grain heating and emissivity, with support for the Intel MIC architecture and heterogeneous multi/many-core systems
- Optimization for the MIC architecture leads to significant performance benefits on the host multi-core CPUs, and vice-versa
- One code for CPUs and Intel Xeon Phi coprocessors
- Publication for Computer Physics Communications in preparation
- Source codes will be publicly available via the CPC Program Library

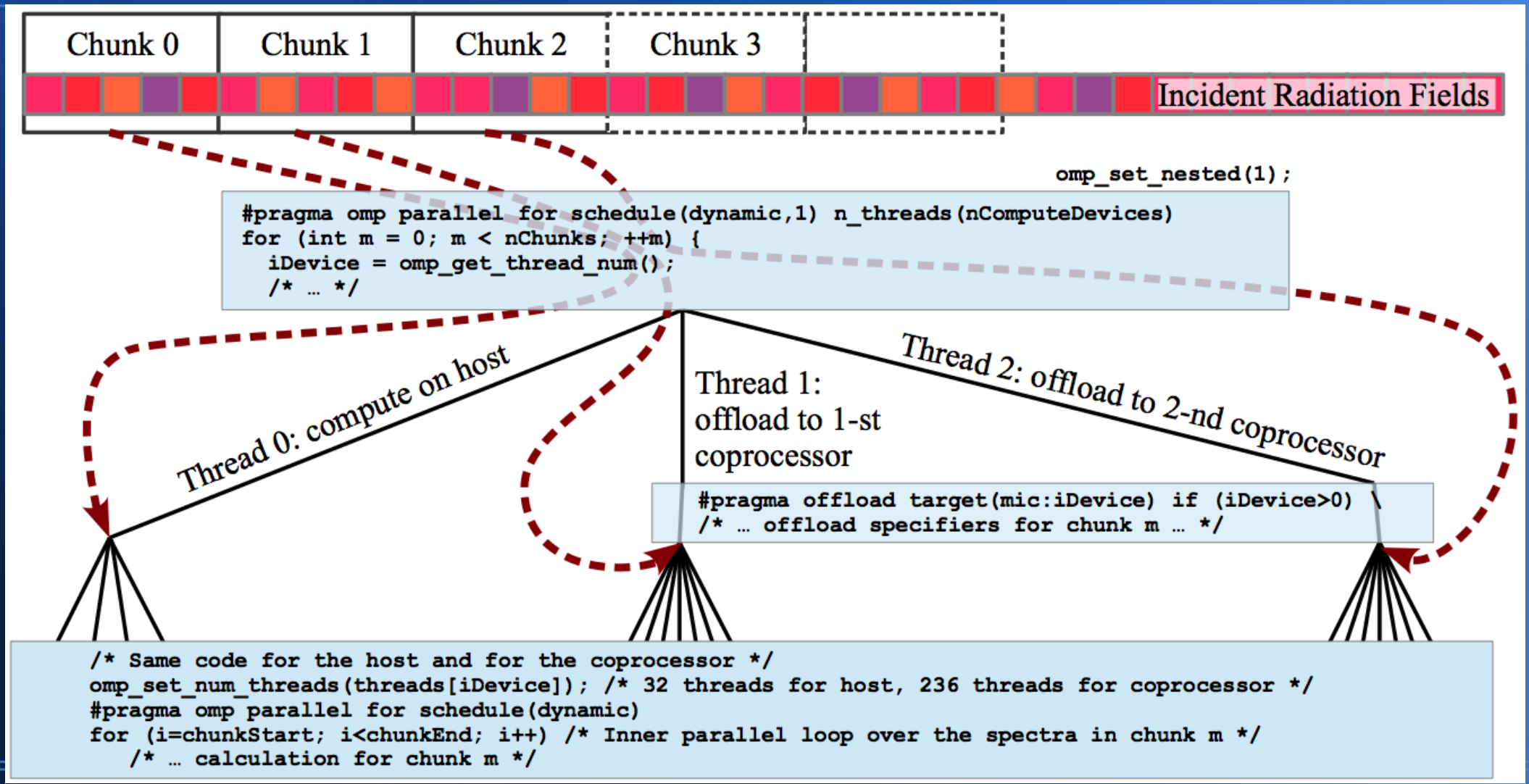
ACKNOWLEDGEMENT

We thank Colfax International and Intel
for early access to Intel Xeon Phi coprocessors
and optimization guides

http://colfax-intl.com/

Backup Slides

Optimization for the Intel MIC Architecture: Heterogeneous Work Sharing



Intel MIC Architecture

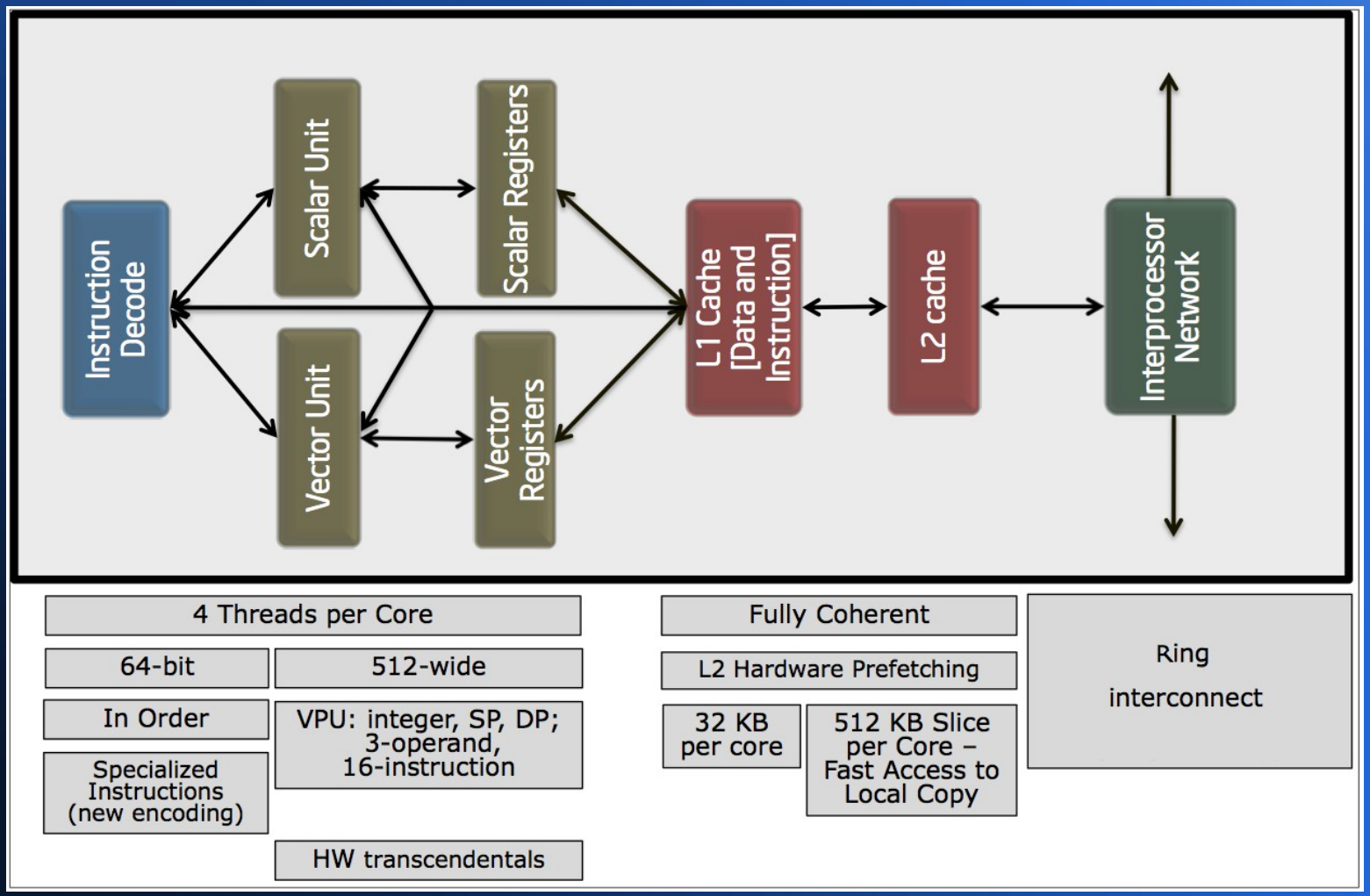


Diagram credit: Intel