“Double Rewards” of Porting Scientific Applications to the Intel MIC Architecture

Troy A. Porter
Hansen Experimental Physics Laboratory
and
Kavli Institute for Particle Astrophysics and Cosmology Stanford University

Work in collaboration with Andrey Vladimirov (Colfax International)
Intel MIC Architecture

Intel Xeon processors (multi-core CPUs)
- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- > 512 GB of DDR3/4 RAM
- Up to 18 cores at ~ 3 GHz
- 2-way SMT
- 256-bit AVX vectors

Intel Xeon Phi coprocessors (Many Integrated Core, or MIC)
- C/C++/Fortran; OpenMP/MPI
- Special μOS Linux
- 6-16 GB of onboard GDDR5
- 57 to 61 cores at ~ 1 GHz
- 4-way SMT
- 512-bit IMCI vectors
Same Code, Better Performance

- For **highly parallel** applications
- **Same code** for CPU and MIC
- **Similar optimization** strategies
- Xeon Phi is 2x-3x faster than Xeon CPU of comparable cost and thermal design power
- Theoretical peak performance: 1 TFLOP/s in DP (75% usable); 350 GB/s on-board RAM bandwidth (50% usable)

Case study: HEATCODE
Native Model
application runs directly on coprocessor

Use Xeon Phi as an independent compute node

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Hello world! I have %ld logical cores.\n", 
    sysconf(_SC_NPROCCESSORS_ONLN ));
}
```

```
user@host% icc hello.c -mmic
user@host% scp a.out mic0:~/
user@host% ssh mic0
user@mic0% ./a.out
Hello world! I have 240 logical cores.
user@mic0%
```
Programming Models for the MIC Architecture

Offload Models
application runs on host, communicates w/coprocessor

Explicit offload (pragma-based)
Virtual-shared Memory

#include <stdio.h>
int main(int argc, char * argv[]) {
    printf("Hello World from host!\n");
    #pragma offload target(mic)
    {
        printf("Hello World from coprocessor!\n");
        fflush(0);
    }
    printf("Bye\n");
}

user@host% icpc hello_offload.cpp -o hello_offload
user@host% ./hello_offload
Hello World from host!
Bye
Hello World from coprocessor!
**Goal:** build a 3D model of the Milky Way Galaxy using a large volume of 2D data from sky surveys.

Andromeda galaxy (left) and the Milky Way (below) seen at near infrared wavelengths.
**Goal:** build a 3D model of the Milky Way Galaxy using a large volume of 2D data from sky surveys.
**Goal:** build a 3D model of the Milky Way Galaxy using a large volume of 2D data from sky surveys.

**Method:** Bayesian inference. Simulate the Galaxy, assess the fit to data, refine 3D model parameters, rinse & repeat.

**Challenge:** modeling the process of stochastic heating of cosmic dust by starlight, in each voxel of a 3D grid, is very time consuming. With unoptimized code, **hundreds of CPU-years** for each run.

One of possible realizations of 3D models of the Milky Way Galaxy (cosmic dust luminosity map calculated by the FRaNKIE code)
Software Stack for Modeling Galactic 3D Structure

- **MultiNest**
  - Bayesian analysis engine
  - Scales to $O(10)$ nodes

- **FRaNKIE**
  - Radiation transport Monte Carlo
  - Scales to multiple cores in 1 node

- **HEATCODE**
  - Cosmic dust heating library
  - Multiple Xeon Phi coprocessors in 1 node

- **P > 0.95**
- **P > 0.67**
Calculation of Stochastic Heating and Emissivity of Cosmic Dust Grains with Optimization for the Intel Many-Core Architecture

Troy A. Porter¹, Andrey E. Vladimirov¹,²

Physics Laboratory, Stanford University, 452 Lomita Mall, Stanford, CA 94305-4085, USA
Colfax International, 750 Palomar Ave, Sunnyvale, CA 94085, USA

Solution: use a computing accelerator, optimize existing code.

Result: HEATCODE (HEterogeneous Architecture library for sTochastic COsmic Dust Emissivity) (open source, code soon to be published)

http://arxiv.org/abs/1311.4627

Hundreds of CPU-years

Hundred of CPU-days
Stochastic Dust Grain Heating

- Small grains ($\leq 0.1 \, \mu m$) are important
- Absorption and re-emission is stochastic (non-thermal)
- Grains undergo “temperature” spikes, characterized by temperature distribution
- Evaluation is computationally expensive
Calculation of Stochastic Dust Emissivity

- **Input**: incident electromagnetic radiation field

- **Intermediate**: “temperature” distribution of grains of all sizes

- **Output**: spectrum of re-emitted photons

Matrix Formalism for Stochastic Dust Emissivity

- **Stage 1:** Interpolate (in log space) and convolve the incident RF with the photon absorption cross sections

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

\[ I(\lambda)\sigma(\lambda) \equiv \Omega(\lambda) \]

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

- **Stage 2:** Form and solve a quasi-triangular system of linear algebraic equations for the “temperature” distribution

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

\[ T_{ij} = 0, \text{ if } i < j - 1 \]

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

\[ X_f = \frac{1}{T_{(f-1)f}} \sum_{i=0}^{f-1} B_{fj} X_j \]

- **Stage 3:** Convolve the “temperature” distribution with the grain size distribution and emissivity function

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

\[ \Lambda(\nu, E_i) = \left\{ \begin{array}{ll}
0, & E_i < h\nu, \\
\frac{2h\nu^3}{c^2} \exp(h\nu/kT_i) \frac{P_i}{\exp(h\nu/kT_i) - 1}, & \text{otherwise}
\end{array} \right. \]

\[ \nu F(\nu) = \int_{a_{\min}}^{a_{\max}} \nu F_a(\nu) Q(a) da \]
Optimization Roadmap

- Scalar Optimization
- Vectorization
- Thread Scalability
- Memory Access
- Communication

Dual-socket Intel Xeon E5-2670 CPU (16 cores total) versus Intel Xeon Phi 5110P coprocessor (60 cores)
Scalar Optimization: Strength Reduction, Precomputation, Optimized Transcendentals

**UNOPTIMIZED IMPLEMENTATION**

```c
for (int i = 0; i < f; i++) { /* Original, unoptimized implementation */
    const double wl = grainWavelength[gI*tempBins*tempBins + f*tempBins + i];
    if (wl >= wavelength[0] && wl <= wavelength[wlBins-1]) {
        /* The usage of std::lower_bound precludes automatic vectorization */
        const float* wlVal = std::lower_bound(&wavelength[0], &wavelength[wlBins-1], wl);
        const int j = wlVal - &wavelength[0];
        const double upper = radiationField[j]*absorptionCrossSection[gI*wlBins + j];
        const double lower = radiationField[j]*absorptionCrossSection[gI*wlBins + j-1];
        if (((upper > 0) && (lower > 0)) { /* Power-law interpolation */
            weightedRadiationField[gI*tempBins*tempBins + f*tempBins + i] =
                exp(log(lower) + (log(upper) - log(lower))*/
                (log(wl) - log(wavelength[j-1]))/(log(wavelength[j]) - log(wavelength[j-1])));
    }
} }
```

- **Combinatorial (non-vectorizable)**
  - Computation of the index

- **Natural base logarithms and exponentials**

- **Loop in “i” is not vectorizable**

- **Eight transcendental functions, one division per evaluation**
Scalar Optimization: Strength Reduction, Precomputation, Optimized Transcendentals

OPTIMIZED IMPLEMENTATION

```c
/* Optimized implementation */
const float upper = radiationField[j]*absorptionCrossSection[gI*wlBins + j];
const float lower = radiationField[j]*absorptionCrossSection[gI*wlBins + j-1];
if (upper > 0.0f) && (lower > 0.0f) { /* Single precision constants */
    const double dLogUpperLower = log2f(upper/lower); /* Single precision functions */
    for (int c = 0; c < qCount; c++) { /* This loop will be partially vectorized */
        const int idx = InterpolationPatternIndex[qCtr + c]; /* Precomputed indices */
        weightedRadiationField[idx] = lower*exp2f(dLogUpperLower*interpolationOffs[qCtr+c]);
    }
} } /* Base 2 exponential and logarithm optimized for Xeon and Xeon Phi */
```

Base 2 logarithms and exponentials

Precomputed index

Loop in “c” is vectorizable

Two transcendental functions, one division per evaluation
Vector Optimization: Alignment and Hints

- In Xeon Phi, memory access works best on 64-byte aligned addresses
- By default, compiler does not assume alignment
- Hint to compiler that data is aligned improves performance
- Additional automatic vectorization hints

```c
/* Aligning data on 64-byte boundary */
float* rSum=(float*)_mm_malloc(
    tempBins*tempBins*sizeof(float), 64);
assert(tempBins%16==0);
...

/* Guarantee alignment to compiler;
 Estimate loop count for 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];
}
```
Vector Optimization: Loop Pattern

- 512 bits vector holds 16 single precision FP numbers
- HEATCODE: padded loop bounds to a multiple of 16 iterations
Vector Optimization: Loop Pattern

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

```c
/* Unoptimized: traversing matrix below the main diagonal */
for (int f = fMax; f >= 1; --f) {
    /* Compiler will implement checks for value of f, and peel the i-loop if f is not a multiple of 16 */
    for (int i = 0; i < f; ++i) {
        rSum[i] += bMatrix[f*tempBins + i];
        bMatrix[f*tempBins + i] = rSum[i];
    }
}

/* Optimized: inner loop always has a multiple of 16 iterations */
for (int f = fMax; f >= 1; --f) {
    const int uB = (f-1)+(16-(f-1)%16)-1;
    const int iMax =
        (uB<=tempBins ? uB : tempBins-1);
    for (int i = 0; i <=iMax; ++i) {
        rSum[i] += bMatrix[f*tempBins + i];
        bMatrix[f*tempBins + i] = rSum[i];
    }
}
```
Threading Optimization: Exposing Parallelism

• Using an OpenMP parallel region inside of #pragma offload
• Distribute independent incident spectra across threads
• Modified the library interface to accept an array of spectra instead of a single spectrum

```
#pragma offload target(mic)...
{
    #pragma omp parallel for schedule(dynamic)
    for (int iRF = 0; i < nSpectra; i++) {
        InterpolateWeightedRF(wlBins, iRF, ...);
        CalculateTemperatureDistribution(...);
        ComputeEmissivity (...);
    }
}
```
Threading Optimization: Reducing Per-Thread Memory Footprint

- Problem: 240 threads do not fit in onboard Xeon Phi memory
- Not an issue on the CPU host!
- Solution: reduce per-thread memory footprint
- How: **inter-procedural fusion** to eliminate unnecessary scratch data passed between functions

```c
InterpolateWeightedRadiationField() {
    for (int i = 0; i < gIMax; i++) {
        /* ... */
    }
}

CalculateMatrices() {
    for (int i = 0; i < gIMax; i++) {
        /* ... */
    }
}

RadiationFieldToTemperatureDistribution() {
    for (int i = 0; i < gIMax; i++) {
        /* ... InterpolateWeightedRadiationField ... */
        weightedRadiationField
    }
    /* ... CalculateMatrices ... */
    transientMatrix
}
```
Memory Traffic Optimization: Loop Tiling

/* 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;
}

/* 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];

        }

    ...

↑ “Before”
“After” →
Communication Optimization: Data Persistence

↑ **Unoptimized:**
For every offload,
- Send/receive input & output
- Send model data
- Allocate/deallocate memory

↑ **Optimized:**
For every offload,
- Send/receive input & output
- Re-use previously sent model data
- Retain memory for use in next offload

```c
/* 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))
{ ... }
```

```c
/* 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))
{ ... }
```
Optimization: Heterogeneous Computing with the Offload Model

- Use **all available** compute devices: CPU + two Xeon Phi
- **Same offloaded code** in C language for both platforms
- For *load balancing*, split work into chunks (\(\sim 10^4\) spectra in each), use “boss-worker” model to dynamically distribute chunks

```c
#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)
    {
        ... 
    }
```
Guided Optimization: VTune

- Intel Vtune Amplifier XE – performance analysis for thread-parallel applications on Intel CPUs and Xeon Phi coprocessors

- Finds **bottlenecks** down to a single line of code

- Diagnoses **performance issues**: cache misses, bandwidth utilization, vectorization intensity

- Uses **hardware event-based** data collection: does not slow down application
“Double Rewards” of Optimization for MIC

- **After optimization**, performance on Xeon Phi 620x better
- But the **same code** is also 125x faster on the Xeon CPU
- Acceleration factor **1.9x**
- **One code** for both platforms, **same methods** of optimization

Dual-socket Intel Xeon E5-2670 CPU (16 cores total) versus Intel Xeon Phi 5110P coprocessor (60 cores)
Compute Density and Efficiency

- **Multiple coprocessors and heterogeneous computing with only one optimized code**

- **Improvement of compute density and power efficiency**

Dual-socket Intel Xeon E5-2670 CPU (16 cores total) versus Intel Xeon Phi 5110P coprocessor (60 cores)
Incremental Porting and Optimization

Unoptimized with Offload

Thread Parallelism: Fit All Threads in Memory

Scalar Optimizations: Precomputation, Precision Control

Vectorization: Alignment, Padding, Hints

Heterogeneous: Using Host + Two Coprocessors

Performance Relative to Baseline

Optimized, ICPC, Xeon Phi

Optimized, ICPC, Host

Optimized, GCC

Baseline: unoptimized, compiled with GCC, running on host (59 ms per spectrum)

Algorithm Optimization: Pruning, Recurrence

Improved Interpolation Method: Packed Operations

Memory Access: Packed Data, Loop Tiling

Offload Traffic: Data Persistence on Coprocessor

Optimization Step
Future-Proofing Applications for Knights Landing

- Future MIC product: codename **Knights Landing**

- **14nm** Tri-Gate technology. In the past, smaller transistors led to more cores in CPUs.

- Available as **stand-alone** chip and as PCIe-endpoint coprocessor

- Instruction set **AVX-512** published
Summary

- **Intel MIC** – accelerator architecture for highly parallel application with support for C/C++/Fortran, OpenMP/MPI

- **Same code and same optimization strategies** for MIC and for multi-core CPU architectures – “double rewards”

- **Optimization areas** include: scalar math, vectorization, thread scalability, memory traffic and communication

- Porting for Xeon Phi prepares application for future product **Knights Landing (KNL)** – MIC platform, 14 nm technology, possibility of usage as a **stand-alone processor**
Memory Traffic Optimization: Loop Tiling

/* 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 jj = 0; jj < jMax; jj += T)
for (int i = 0; i < iMax; ++i)
for (int j = jj; j < jj+T; ++j)
PerformWork(A[i], B[jj]);

Example:
- tile size $T=2$
- cache size = 3

Cache Hit Rate = 6/16  SLOWER

Cache Hit Rate = 10/16  FASTER

Without Tiling

With Tiling