Accelerating Public Domain Applications: Lessons from Models of Radiation Transport in the Milky Way Galaxy

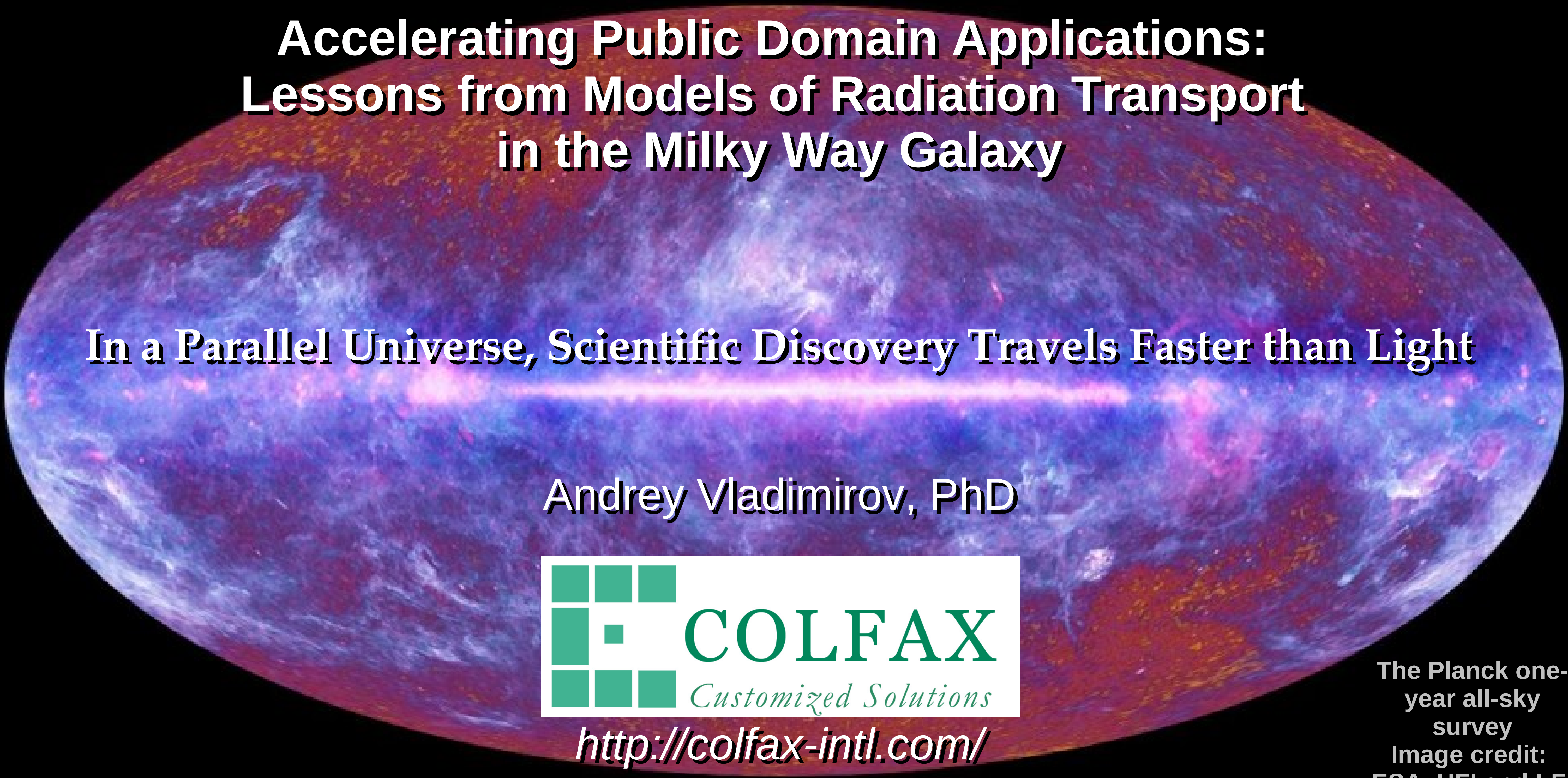
In a Parallel Universe, Scientific Discovery Travels Faster than Light

Andrey Vladimirov, PhD

COLFAX
Customized Solutions

http://colfax-intl.com/

Intel Theater Presentation SC'13, Denver, CO, USA

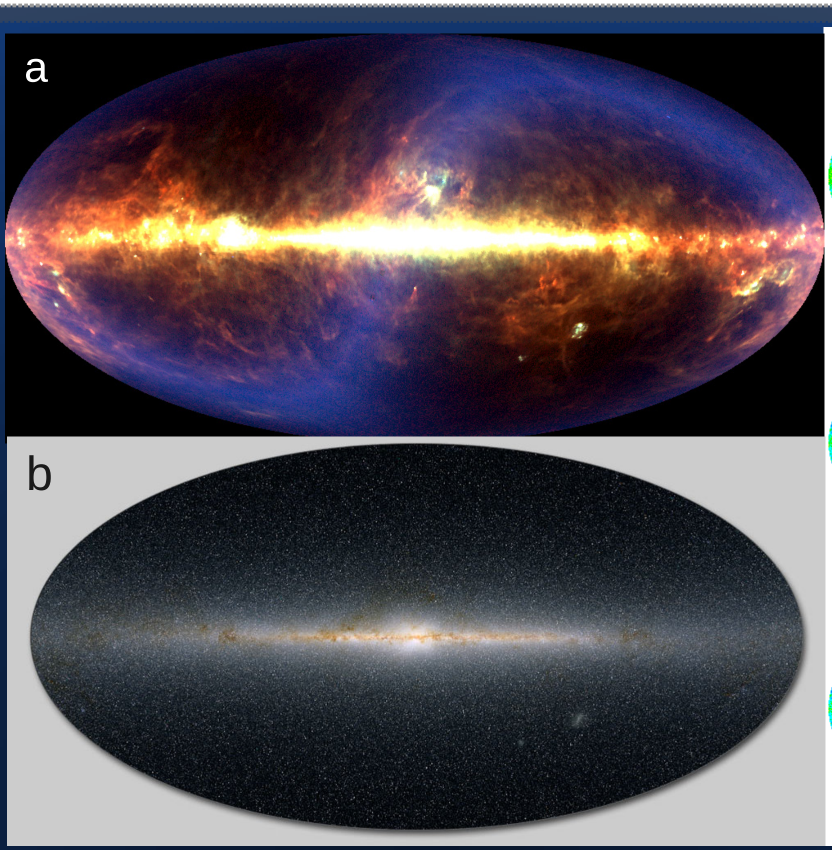


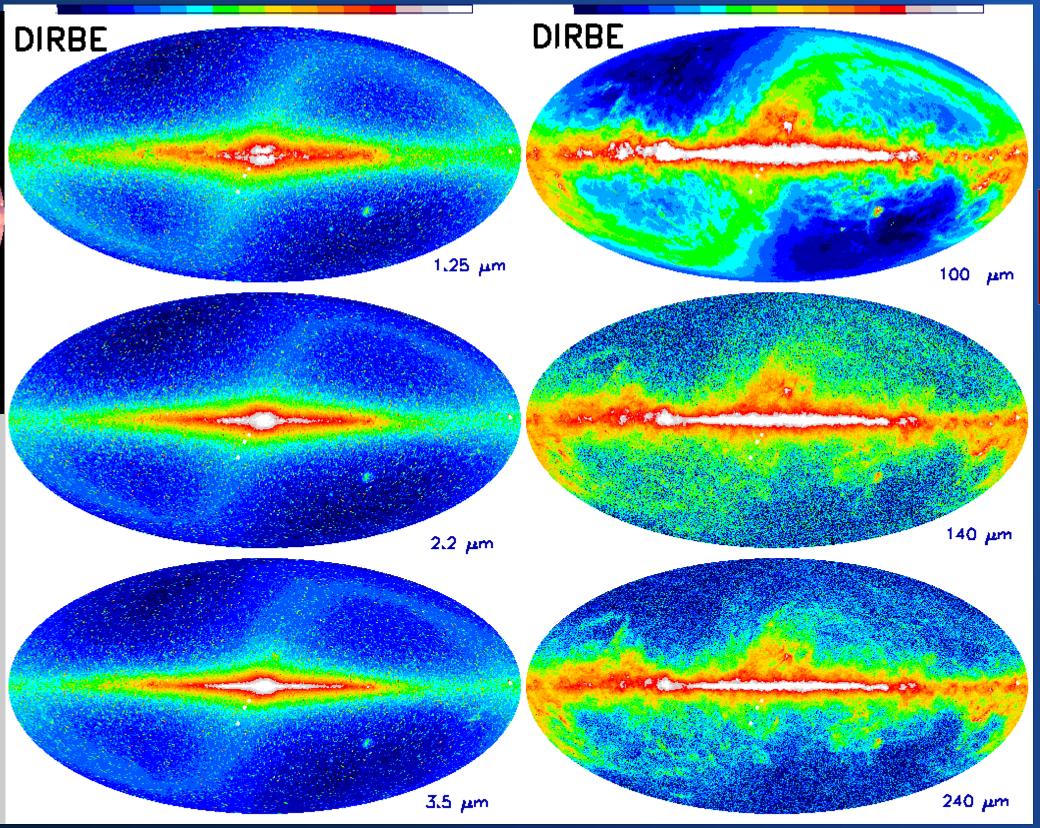
The Planck one-year all-sky survey
Image credit: ESA, HFI and LFI consortia.

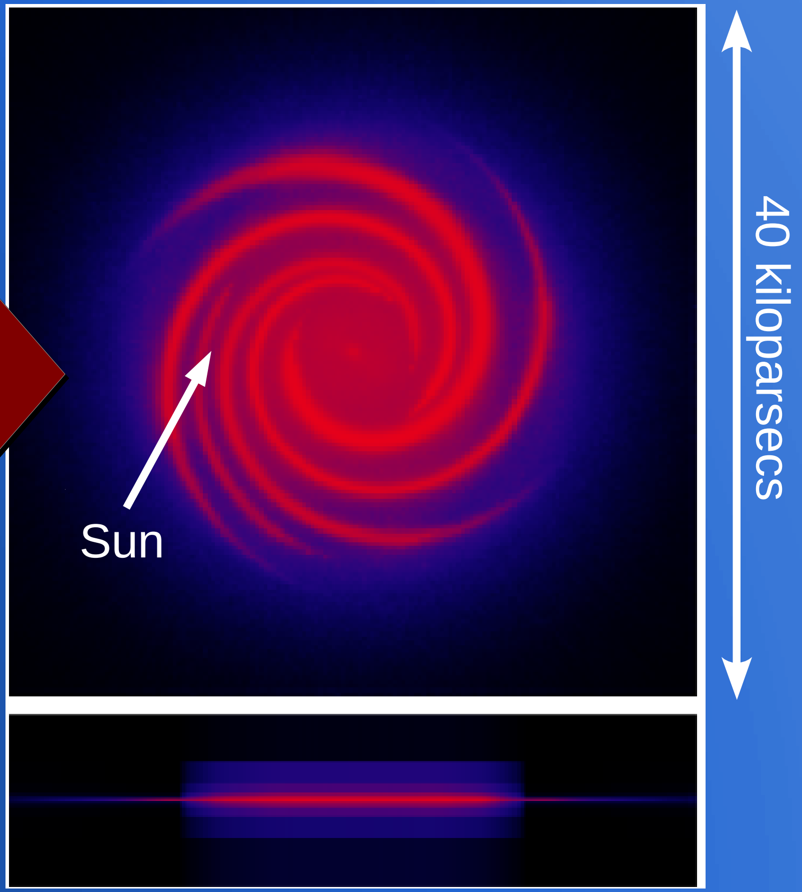
Building a 3D Model of the Milky Way Galaxy

Goal: build a 3D model of the Milky Way Galaxy using a large volume of 2D data from sky surveys.

An instance of a
generic data analysis problem







One of possible realizations of 3D models of the Milky Way Galaxy (cosmic dust luminosity map calculated by the FRaNKIE code)

Image credits: a) DIRBE Team, COBE, NASA; b) 2MASS/J. Carpenter, T. H. Jarrett, & R. Hurt

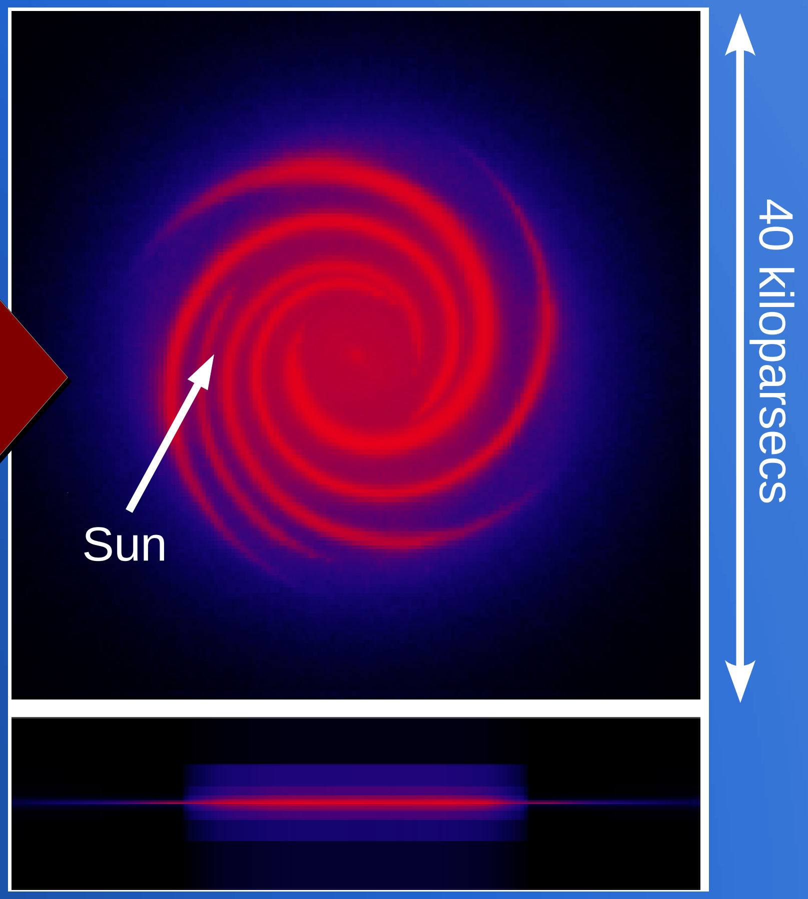
Building a 3D Model of the Milky Way Galaxy

An instance of a
generic data analysis problem

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.

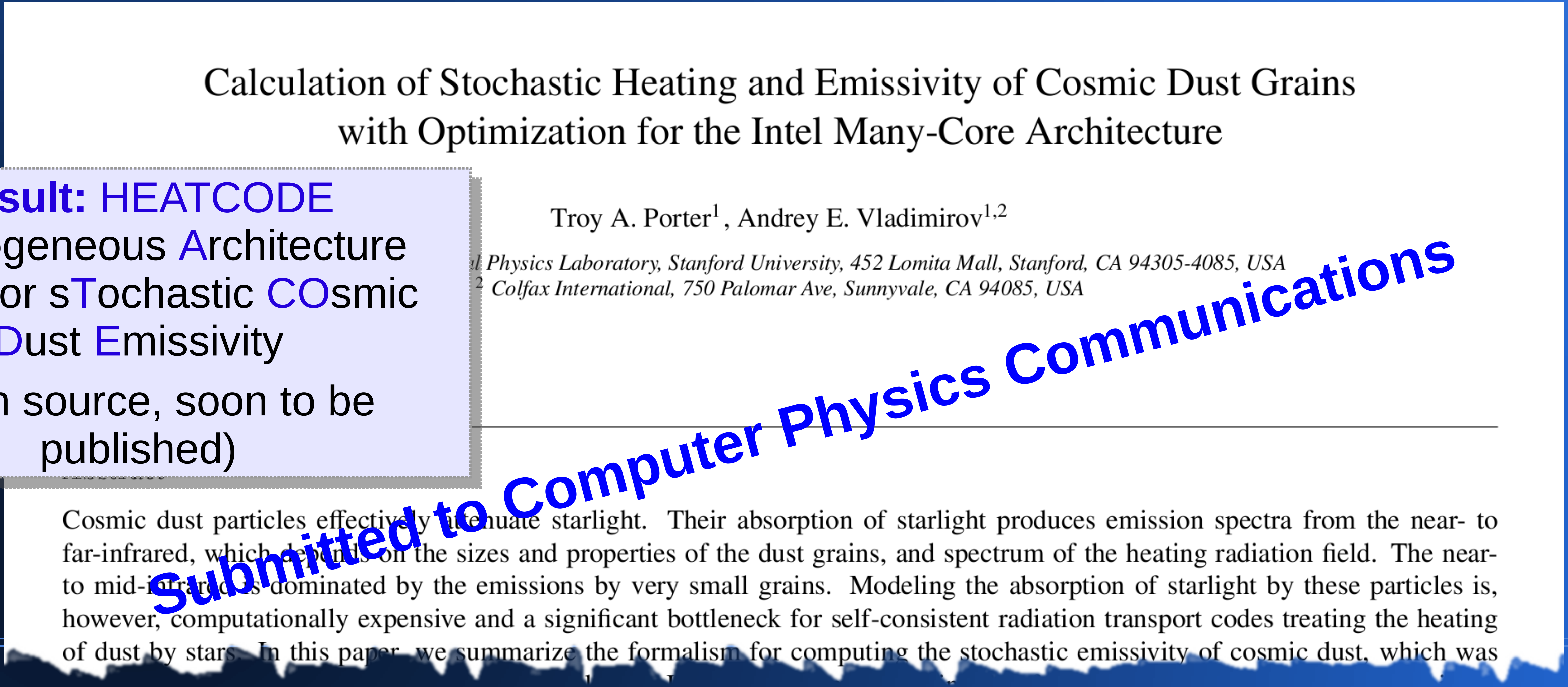
Challenge: modeling the process of stochastic heating of cosmic dust by starlight, in each voxel of a 3D grid, is very time consuming. With unoptimized code, **hundreds of CPU-years** for each run.



One of possible realizations of 3D models of the Milky Way Galaxy (cosmic dust luminosity map calculated by the FRaNKIE code)

Accelerating Radiation Transport Models for the Milky Way

Solution: use a computing accelerator, optimize existing code.



Calculation of Stochastic Heating and Emissivity of Cosmic Dust Grains with Optimization for the Intel Many-Core Architecture

Troy A. Porter[1], Andrey E. Vladimirov[1,2]

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

Cosmic dust particles effectively attenuate starlight. Their absorption of starlight produces emission spectra from the near- to far-infrared, which depends on the sizes and properties of the dust grains, and spectrum of the heating radiation field. The near- to mid-infrared is dominated by the emissions by very small grains. Modeling the absorption of starlight by these particles is, however, computationally expensive and a significant bottleneck for self-consistent radiation transport codes treating the heating of dust by stars. In this paper, we summarize the formalism for computing the stochastic emissivity of cosmic dust, which was

Submitted to Computer Physics Communications

Result: HEATCODE
HEterogeneous Architecture library for sTochastic COsmic Dust Emissivity

(open source, soon to be published)

Hundreds of CPU-**years**

Hundreds of CPU-**days**

Three Mainstream Routes

Framework	GPGPU + CUDA	GPGPU or MIC + OpenCL or Template Libraries	MIC (Xeon Phi) + C/C++, OpenMP
Maturity	Established	Young	Brand new
Target Architecture	GPU only	CPU, GPU and MIC	CPU and MIC
Development	Re-write in CUDA	Re-write in OpenCL	Orchestrate Offload
Optimization	for GPU	for each platform	common arch.
End-Users Must Know	CUDA	OpenCL	C/C++ and OpenMP

⇧ **Our choice** for HEATCODE

Offload to Xeon Phi and Out-of-Box Performance

Easy to offload the calculation to Xeon Phi:

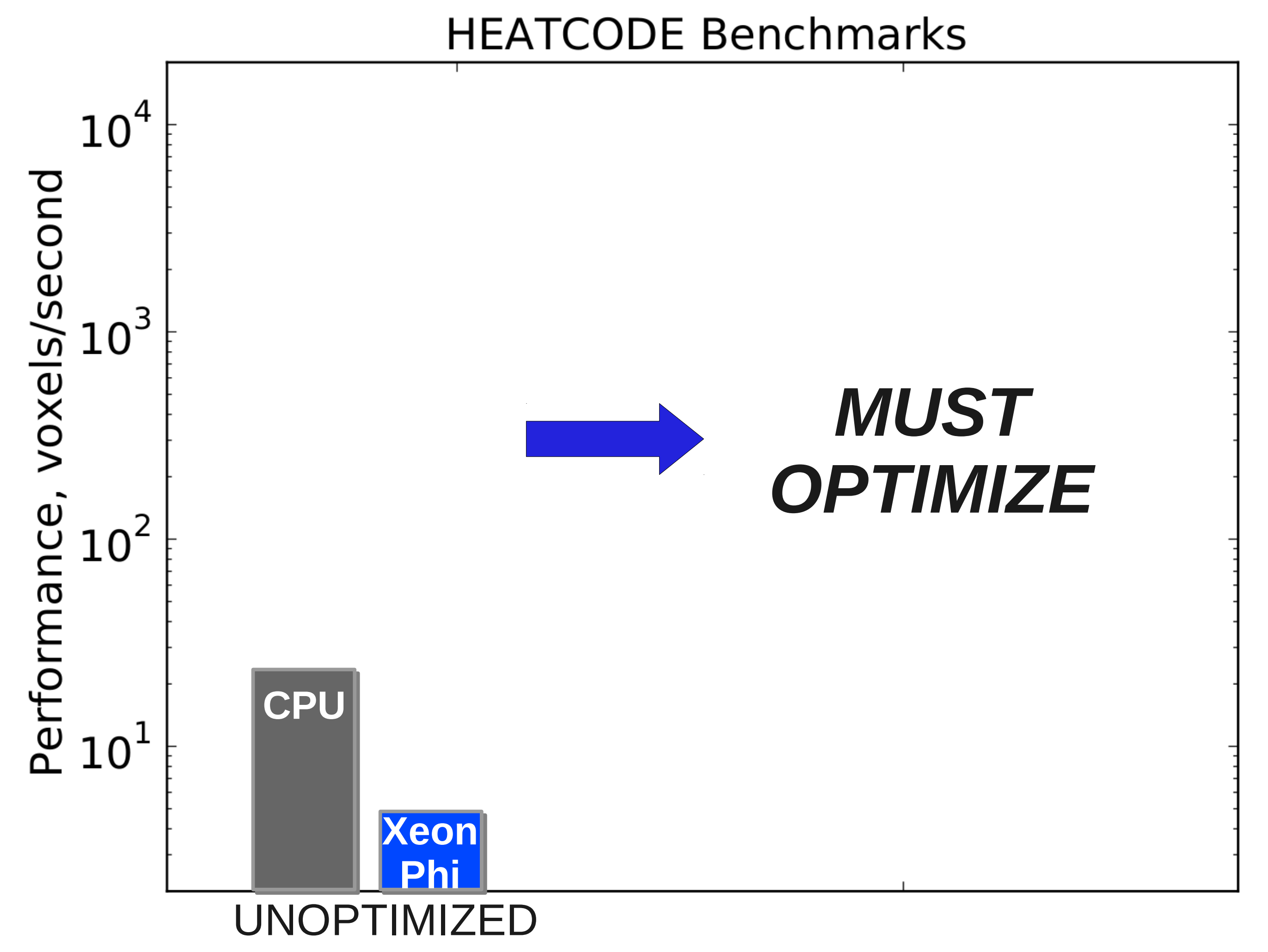
- **Represent data as bitwise-copyable arrays**
- **Insert #pragma offload into the code**

```
void CalculateTransientEmissivity(
    const vector< const valarray<double>* > & inData,
    const vector< valarray<double>* > & outData) {

const int M = inData.size();
double* inDataBC = (double*) malloc(sizeof(double)*M*N);
for (int r = 0; r < M; r++)
  inDataBC[r*N:N] = (*inData[r])[0:N]

#pragma offload target(mic) \
   in(inDataBC : length(M*N)) out(outDataBC : length(M*N))
  {
      #pragma omp parallel for
      for (int r = 0; r < M; r++) {
        /* ... proceed with calculation ... */
      }
  }
```

Performance out of the box? No free lunch! →



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

Optimization Strategies: “Without Your Space Helmet, Dave, You Are Going to Find That Rather Difficult”

1) Scale to 240 threads:

- Reduce memory footprint
- Increase iteration space, collapse nested loops
- Avoid synchronization

2) Vectorization:

- Rely on Intel compiler for auto-vectorization
- Guide compiler with pragma hints
- Pad/modify loop bounds

3) Memory traffic:

- Change order of operations for data locality
- Avoid dynamic memory allocation

4) General:

- Single precision everywhere
- Optimized math functions
- Precompute but not too much

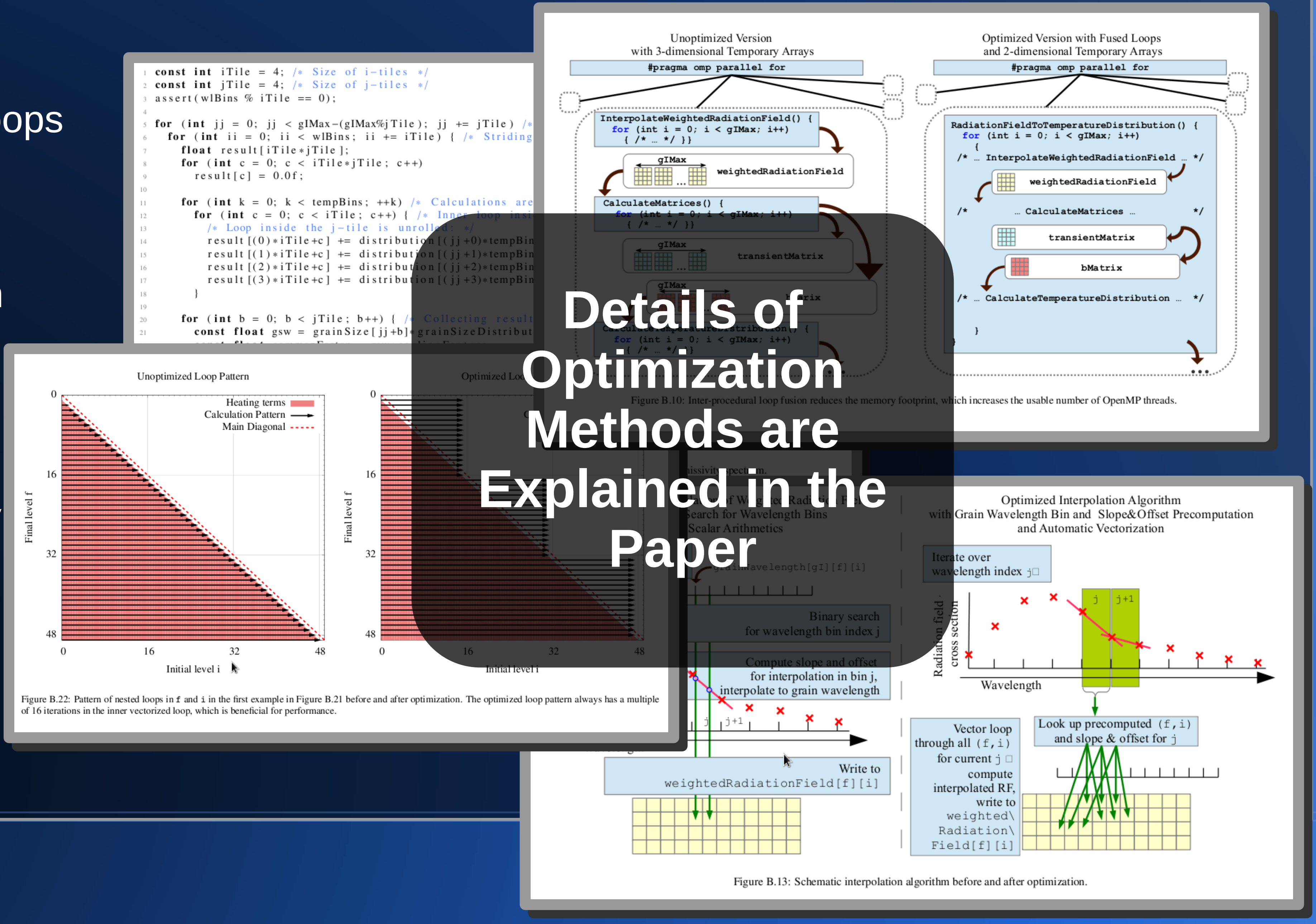
Optimization Strategies: “Without Your Space Helmet, Dave, You Are Going to Find That Rather Difficult”

1) Scale to 240 threads:
- Reduce memory footprint
- Increase iteration space, collapse nested loops
- Avoid synchronization

2) Vectorization:
- Rely on Intel compiler for auto-vectorization
- Guide compiler with pragma hints
- Pad/modify loop bounds

3) Memory traffic:
- Change order of operations for data locality
- Avoid dynamic memory allocation

4) General:
- Single precision everywhere
- Optimized math functions
- Precompute but not too much



```
const int iTile = 4; /* Size of i-tiles */
const int jTile = 4; /* Size of j-tiles */
assert(wlBins % iTile == 0);

for (int jj = 0; jj < gIMax-(gIMax%jTile); jj += jTile) /*
  for (int ii = 0; ii < wlBins; ii += iTile) { /* Striding
    float result[iTile*jTile];
    for (int c = 0; c < iTile*jTile; c++)
      result[c] = 0.0f;

    for (int k = 0; k < tempBins; ++k) /* Calculations are
      for (int c = 0; c < iTile; c++) { /* Inner loop insi
        /* Loop inside the j-tile is unrolled: */
        result[(0)*iTile+c] += distribution[(jj+0)*tempBin
        result[(1)*iTile+c] += distribution[(jj+1)*tempBin
        result[(2)*iTile+c] += distribution[(jj+2)*tempBin
        result[(3)*iTile+c] += distribution[(jj+3)*tempBin
      }

    for (int b = 0; b < jTile; b++) { /* Collecting result
      const float gsw = grainSize[jj+b]*grainSizeDistribut
```

Figure B.10: Inter-procedural loop fusion reduces the memory footprint, which increases the usable number of OpenMP threads.

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.

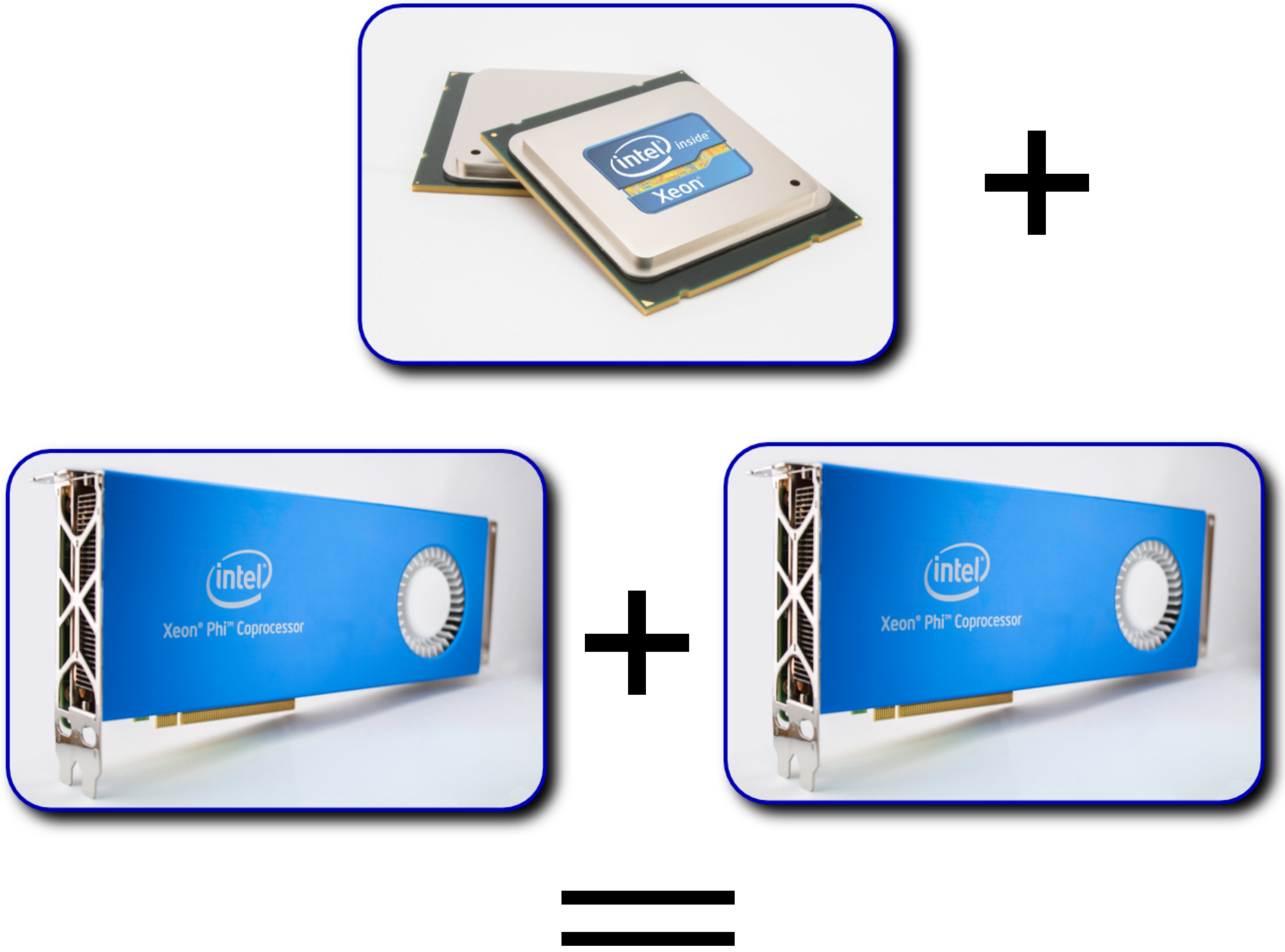
Figure B.13: Schematic interpolation algorithm before and after optimization.

Details of Optimization Methods are Explained in the Paper

Heterogeneous Computing: Solid Rocket Boosters

- “Embarrassingly parallel” => easy to use the CPU in tandem with multiple coprocessors

- We use dynamic scheduling, assigning work-items to compute devices as they become available

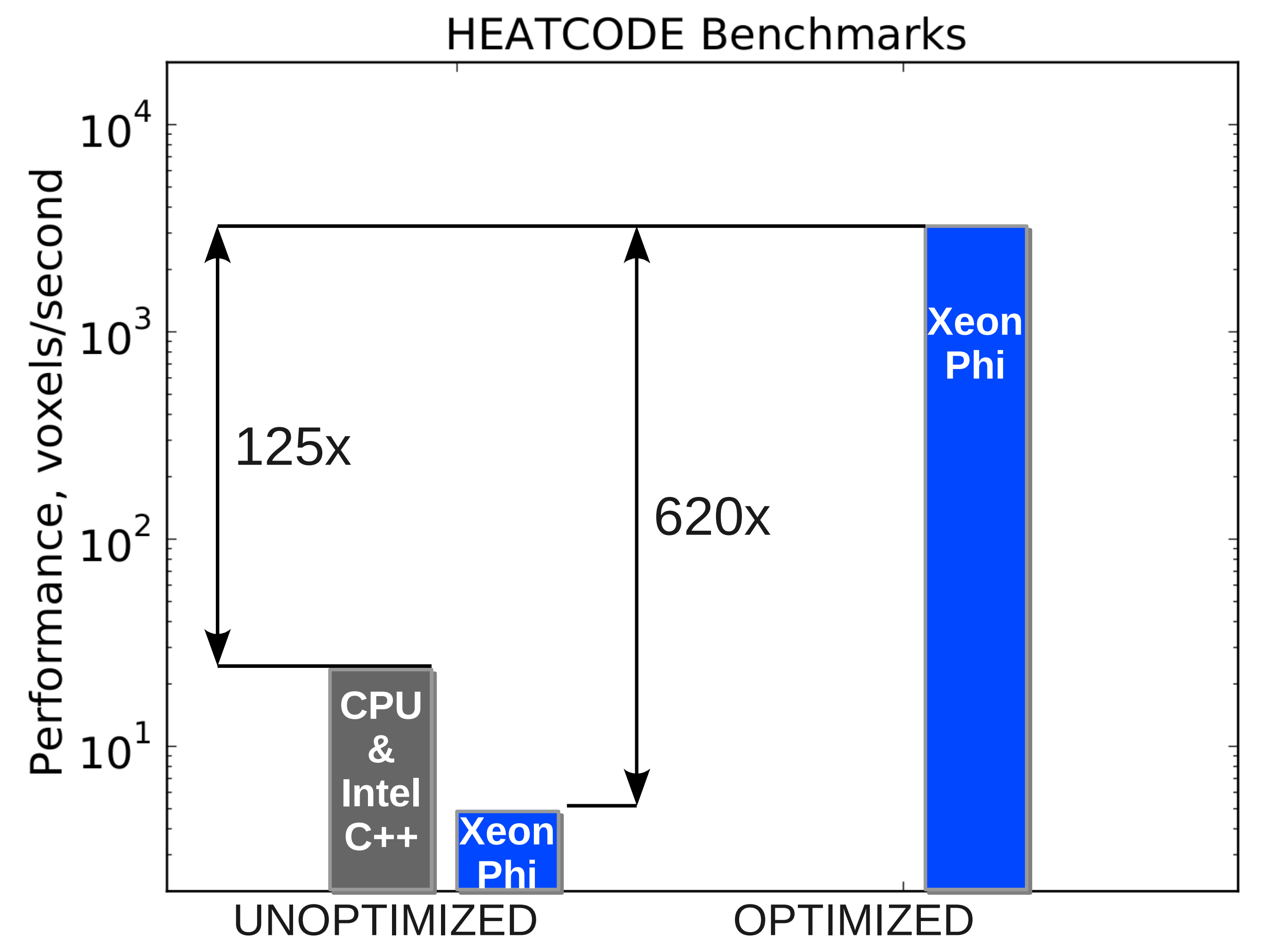


HETEROGENEOUS SYSTEM

Performance Benchmarks: The Big Bang Experience

After optimization, performance on Xeon Phi improved tremendously.

Did we achieve a MIC vs CPU speedup of 125x ?



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

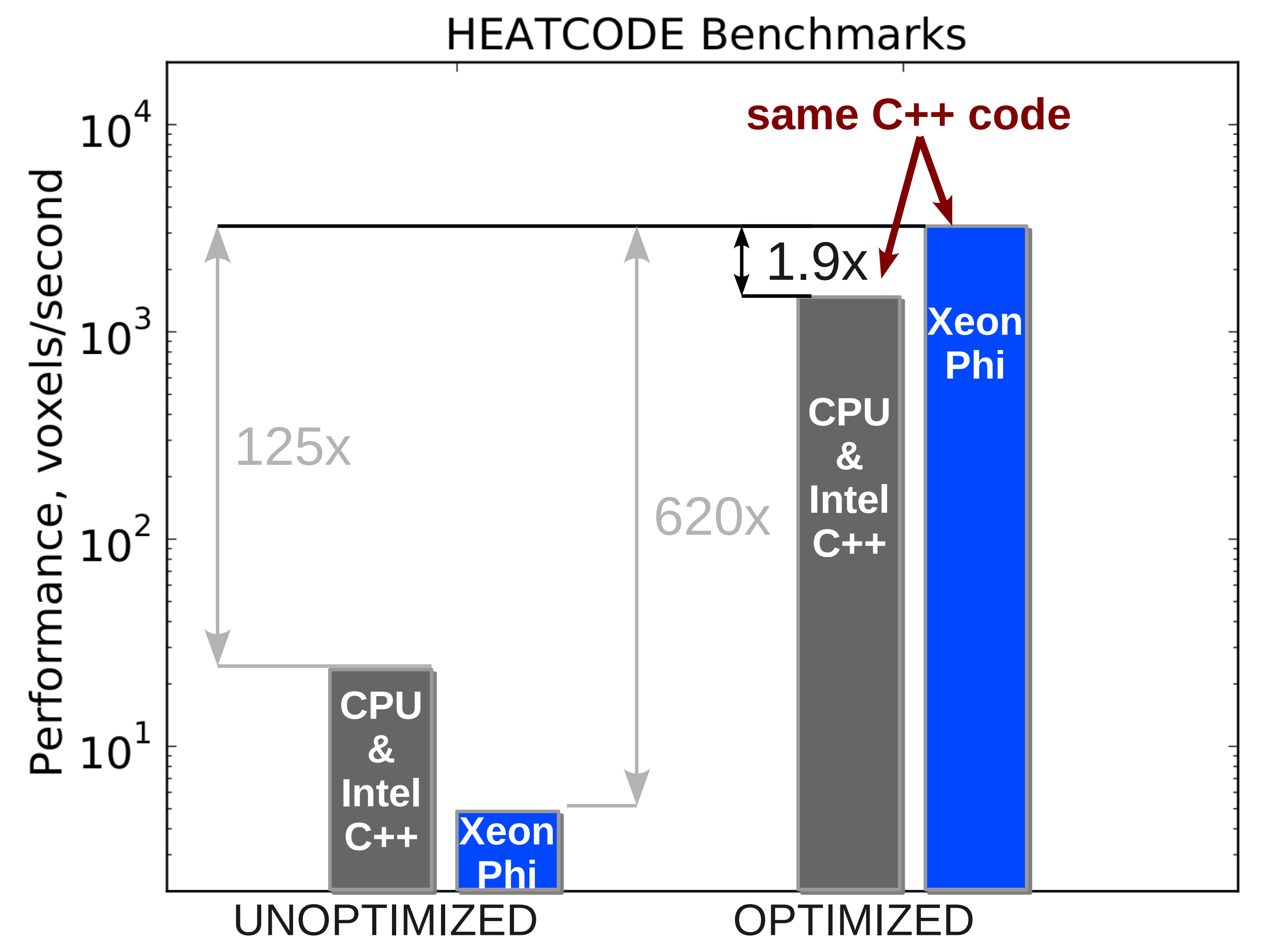
Performance Benchmarks: The Big Bang Experience

After optimization, performance on Xeon Phi improved tremendously.

Did we achieve a MIC vs CPU speedup of 125x ?

Not really, because the CPU performance also grew by a large factor!

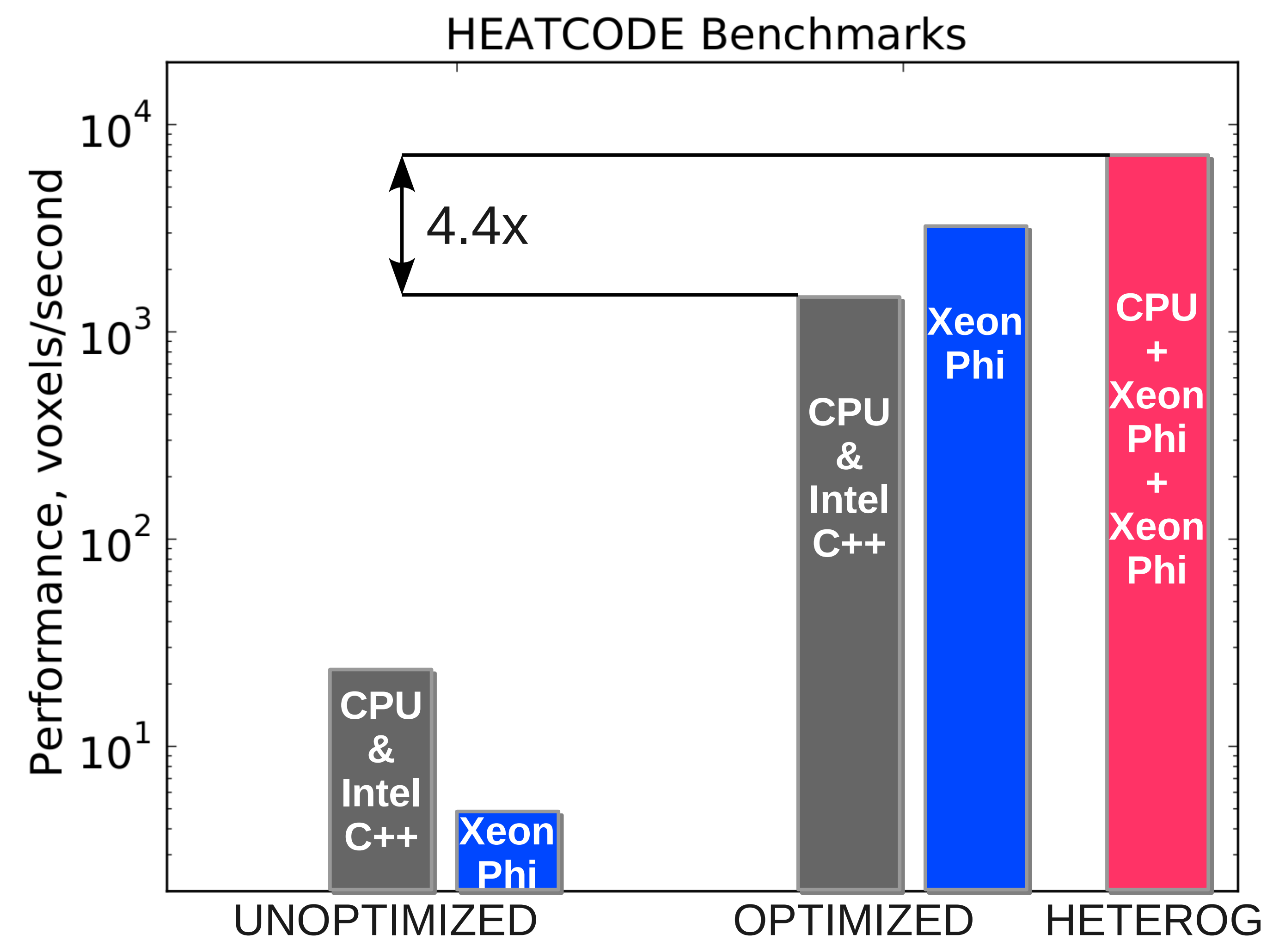
(important for end-users without a Xeon Phi)



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

Performance Benchmarks: The Big Bang Experience

Heterogeneous calculation (CPU + two Xeon Phi coprocessors) improves per-node performance even further.

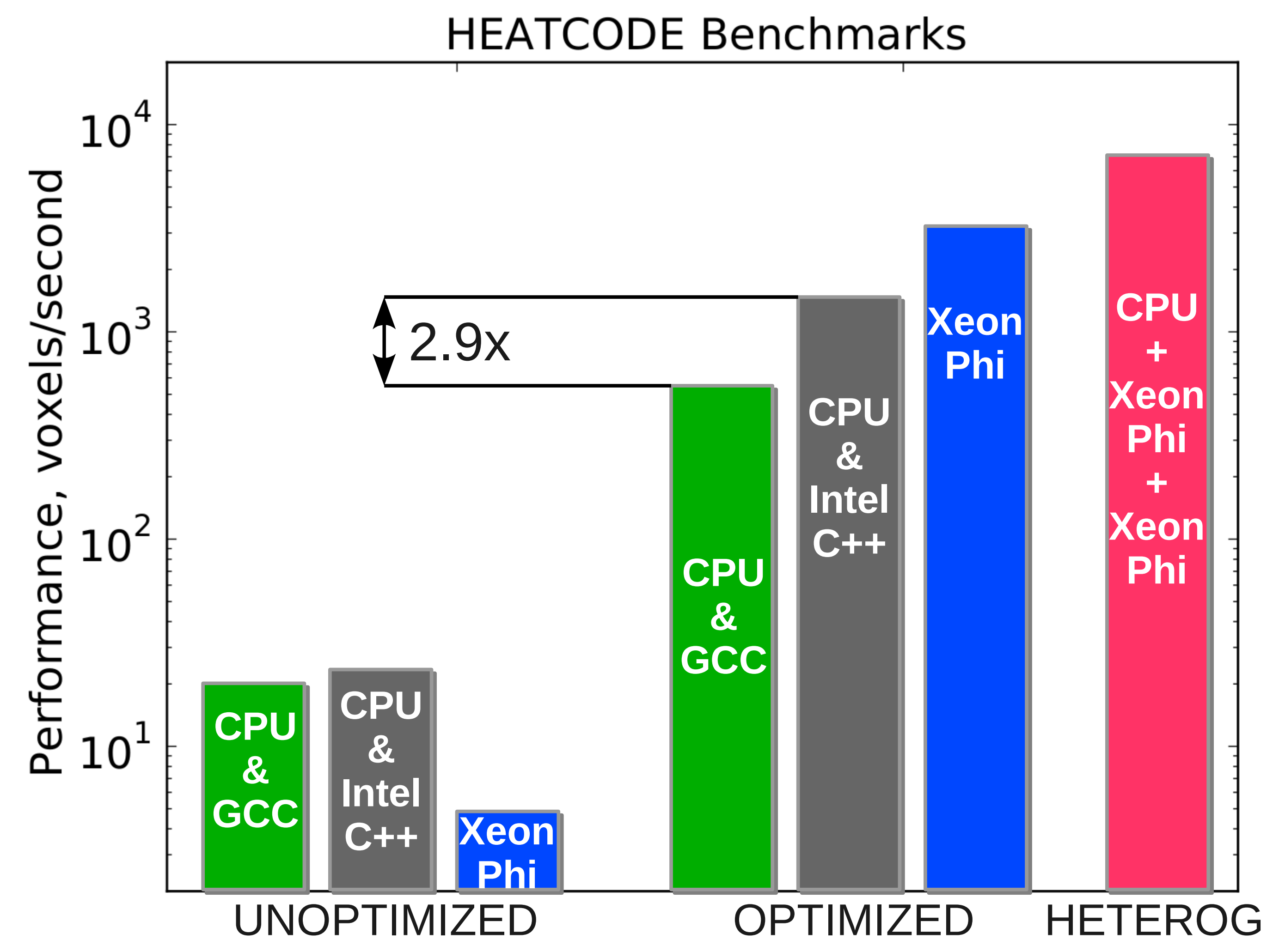


Dual-socket Intel Xeon E5-2670 CPU (16 cores total)
versus
Intel Xeon Phi 5110P coprocessor (60 cores)

Performance Benchmarks: The Big Bang Experience

Our optimizations had a positive effect with the GNU compiler

(important for end-users without an Intel compiler)



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

Peer Review: Houston, Do You Read Me?

How difficult is it to **read, understand and modify** the optimized MIC code for **end-users** without Xeon Phi programming knowledge?

Example 1

BEFORE OPTIMIZATION

```
for (int f = fMax[gI]; f >= 1; --f) {
  for (int i = 0; i <= f; ++i) {
...
```

AFTER OPTIMIZATION

```
for (int f = fMax[gI]; f >= 1; --f) {
  const int uB = (f - 1) + (FLOATS_IN_ALIGN_BYTES-
                 (f - 1)%FLOATS_IN_ALIGN_BYTES) - 1;
  const int iMax = (uB <= tempBins-1 ? uB : tempBins-1);
  for (int i = 0; i <= iMax; ++i) {
    ... /* More regular vectorization pattern */
```

Example 2

```
for (int i = 0; i < wlBins; ++i) {
  for (int j = 0; j < gsBins; ++j) {
    for (int k = 0; k < tempBins; ++k)
      dist[k] =
         planckDistribution[i*tempBins + k]*
               distribution[j*tempBins + k];

    double result = 0;
    result = std::accumulate(&dist[0],
                &dist[tempBins], result);
...
```

```
for (int jj = 0; jj < gIMax-(gIMax%jTile); jj += jTile)
  for (int ii=0; ii<wlBins; ii+=iTile) { /* Loop tiling */
    float result[iTile*jTile];
    for (int c = 0; c < iTile*jTile; c++) result[c] = 0.0f;

    for (int k = 0; k < tempBins; ++k)
      for (int c = 0; c < iTile; c++) {
        result[(0)*iTile+c] +=
                      distribution[(jj+0)*tempBins+k]*
                planckDistribution[(ii+c)*tempBins+k];
        result[(1)*iTile+c] += ...
```

The code may get... stylistically challenging. But it is still C/C++ with OpenMP parallelism.

Click to Download
Even If You Do Not Have a Xeon Phi

- CPU & each coprocessor is an independent compute device.
- Distribute & balance work across compute devices.
- Without coprocessors, fall back to the CPU.
- Same performance-critical code for CPUs and coprocessors.

```
/* nComputeDevices is the number of coprocessors that the end user
    wishes to employ, plus 1 if the host CPU is used as well */
omp_set_nested(1);

omp_set_num_threads(nComputeDevices)
#pragma omp parallel for if (nComputeDevices > 1) schedule(dynamic,1)
for (int m = 0; m < nChunks; m++) {
  /* Bind one OpenMP thread to each device for scheduling */
  int iMic = omp_get_thread_num() - computeOnHost;

  #pragma offload target(mic:iMic)  if (iMic >= 0) in (...) out(...)
  {
    omp_set_num_threads(defaultThreadsForThisDevice)
    /* Spawn OpenMP threads within each compute device for processing */
    #pragma omp parallel for
    for (int r = 0; r < thisChunkSize; r++) {
      /* ... Code that runs either on the CPU or on the coprocessor
             goes here. */
    }
  }
}
```

Click to Download
Even If You Do Not Have an Intel Compiler

```
/* Compiler-specific hints can be
   protected with the preprocessor macro
   __INTEL_COMPILER to avoid compilation
   warnings from non-Intel compilers */
#ifdef __INTEL_COMPILER
#pragma vector aligned
#pragma simd
#endif
for (int i = 0; i < tempBins; i++) {
  /* ... */
}

/* Also for compiler-specific tuning */
#ifdef __INTEL_COMPILER
#define FASTLOG log2f
#define FASTEXP exp2f
#else
#define FASTLOG logf
#define FASTEXP expf
#endif
```

```
/* Code specific to Xeon Phi coprocessor programming
   can be protected with the macro __INTEL_OFFLOAD.
   It is defined only in Intel compilers that support
   the MIC architecture */
#ifdef __INTEL_OFFLOAD
#pragma offload_attribute(push, target(mic))
#endif

  void RadiationFieldToTemperatureDistribution(...);

#ifdef __INTEL_OFFLOAD
#pragma offload_attribute(pop)
#endif
```

```
/* Macro __MIC__ protects coprocessor-specific tuning */
#ifdef __MIC__
   const int tuningParameter = 16;
#else
   const int tuningParameter = 8;
#endif
```

Summary: Acceleration with Xeon Phi Coprocessors for Public Domain Applications

- Same code for Xeon and Xeon Phi → Do optimization only once
- CPU optimization is often a “low-hanging fruit”: ~100x for HEATCODE

- Users without Xeon Phi can still use the application on the CPU
- Users without the Intel compiler can still use GCC
- Users without CUDA or OpenCL knowledge can understand and modify the code

- Forward-scalable to future many-/multi-core platforms

Multithreaded Code + **Offload** + **Optimization** + **Maintain One Code** =

☑ MIC-accelerated application
☑ Xeon Phi + CPU is possible
☑ No Xeon Phi? Use the CPU

GPU Kernels + **Data Movement** + **Optimization** + **Maintain GPU Code** =

☑ GPU-only application
☒ GPU + CPU not possible
☒ No GPU? No application