Programming and Optimization for Intel® Architecture

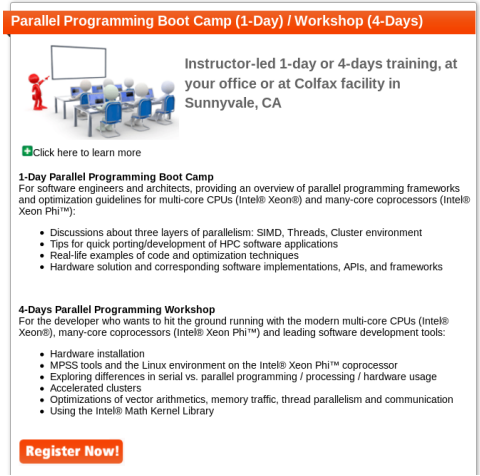
The Hands-On Workshop (HOW) Series

Andrey Vladimirov, PhD, and Ryo Asai
Colfax International — @colfaxintl

March 2016 , Rev. 02a

About This Document

This document represents the materials of a Web-based training “Programming and Optimization with Intel Architecture“ developed and run by Colfax International.



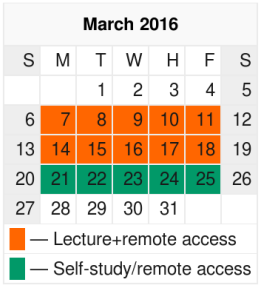
colfaxresearch.com/how-series

Disclaimer

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

Course Roadmap

1. Why Intel Parallel Architectures?
   - Parallelism and specialization – March 7
   - Programming model continuity – March 7
2. Programming models for Xeon Phi coprocessors
   - Native programming – March 7
   - Offload programming – March 8
3. Expressing Parallelism
   - Introduction to vectorization – March 9
   - Crash-course on OpenMP – March 10
4. Optimization – intro on March 11
   - Vectorization tuning – March 14
   - Multi-threading – March 15, 16
   - Memory traffic – March 17
5. ~~Tools: MKL and VTune~~ MPI – March 18



HOW Online

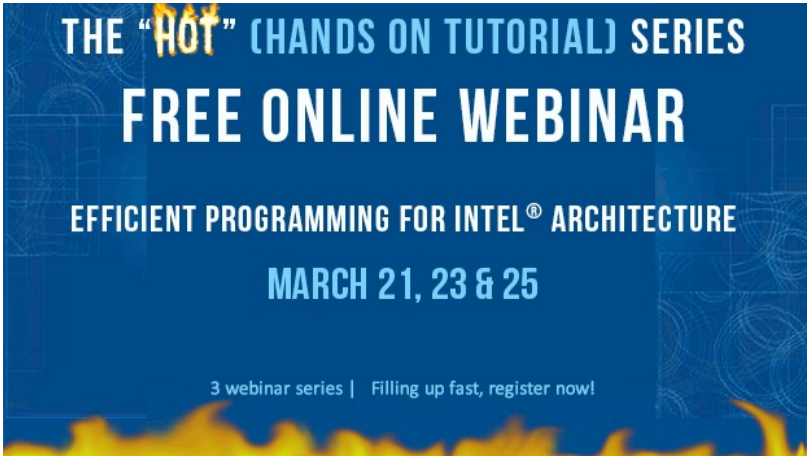
Course page: colfaxresearch.com/how-16-03

- Slides (including this one), code downloads
- Video of recorded sessions
- Chat (during webinars or offline)

Additional resources:

- More workshops like this one: colfaxresearch.com/how-series
- Video courses: colfaxresearch.com/video-courses
- Intel Many Integrated Core Architecture Forum

HOT Series



colfaxresearch.com/hot-16-03/

HOW Series, April 2016

colfaxresearch.com/how-16-04/

§2. Refresh

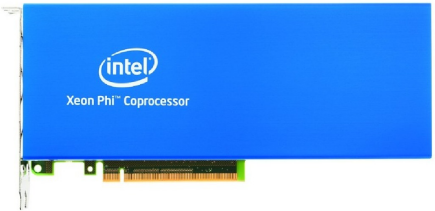
Performance Optimization

Computing Platforms

Intel Xeon Processor

Current: Haswell
Upcoming: Broadwell

Intel Xeon Phi Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

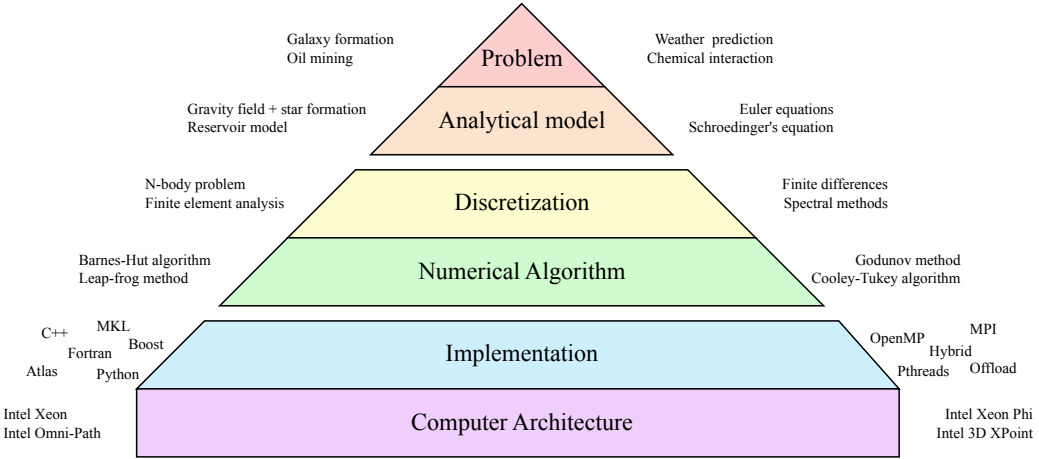
* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Computing in Science and Engineering

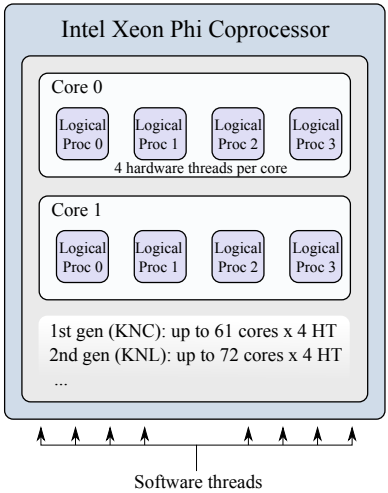


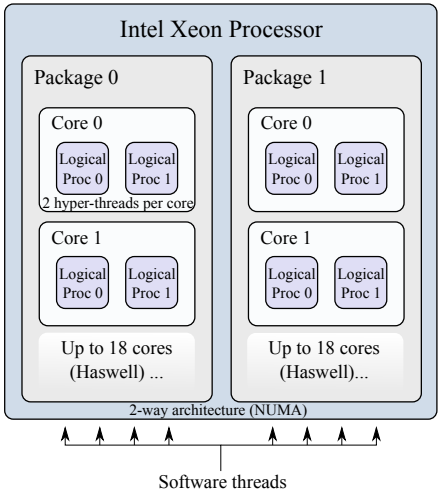
Optimization Areas

1. **Scalar optimization** (compiler-friendly practices)
2. **Vectorization** (must use 16- or 8-wide vectors)
3. **Multi-threading** (must scale to 100+ threads)
4. **Memory access** (streaming access or tiling)
5. **Communication** (offload, MPI traffic control)

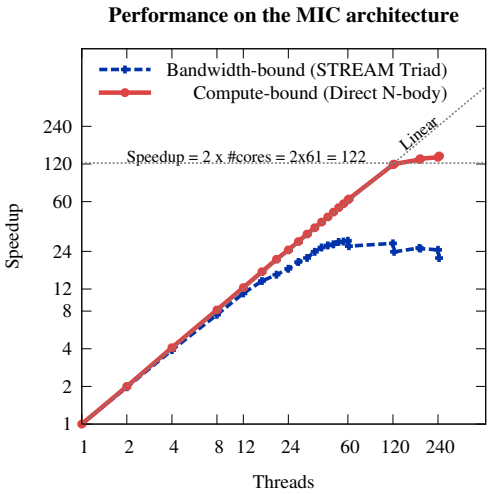
Cores, Threads and OpenMP

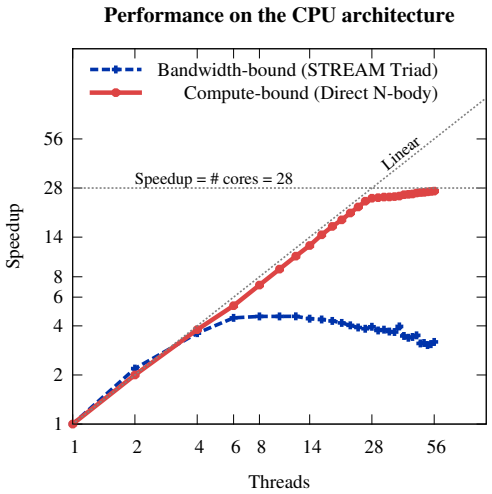
Processor Hierarchy





Scalability Expectations: MIC versus CPU





“Hello World” OpenMP Programs

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5    const int nt=omp_get_max_threads();
6    printf("OpenMP with %d threads\n", nt);
7
8  #pragma omp parallel
9    {
10     printf("Hello World from thread %d\n", omp_get_thread_num());
11   }
12 }
```


§3. Optimization of Multi-Threading I

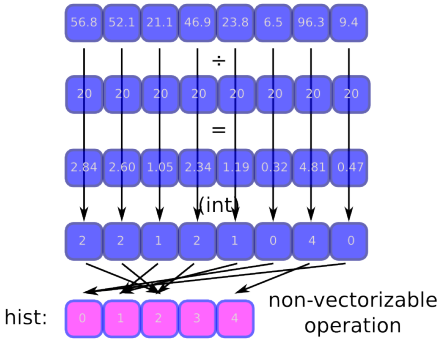
Too Much Synchronization

Example: Strip-Mining for Vectorization

Computing a histogram ($m << n$):

```
1  void Histogram(
2    // Ages, values from 0.0f to 100.0f:
3    const float* age,
4    // Size of array age, n=100000000:
5    const int n,
6    // Output: counts in groups:
7    int* const hist,
8    // Size of array hist, m=5:
9    const int m,
10   // group_width=20.0f
11   const float group_width) {
12     for (int i = 0; i < n; i++) {
13       const int j = int(age[i]/group_width);
14       hist[j]++;
15     }
16 }
```

- Code cannot be auto-vectorized
- True vector dependence

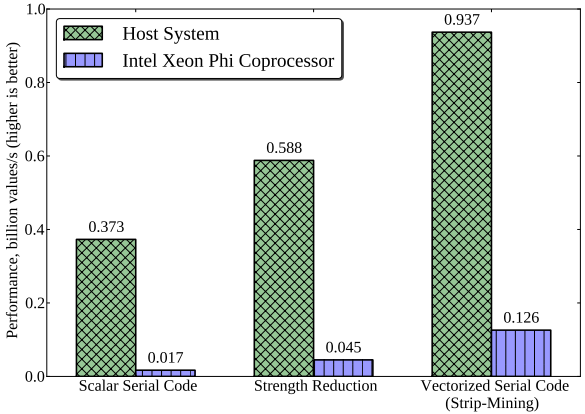


The Same Calculation, Strip-Mined, Vectorized

```
1  void Histogram(const float* age, int* const hist, const int n,
2  const float group_width, const int m) {
3    const int vecLen = 16; // Length of vectorized loop
4    const float invGroupWidth = 1.0f/group_width; // Pre-compute the reciprocal
5    // Strip-mining the loop in order to vectorize the inner short loop
6    // Note: this algorithm assumes n%vecLen == 0.
7    for (int ii = 0; ii < n; ii += vecLen) { //Temporary store vecLen indices
8      int index[vecLen] __attribute__((aligned(64)));
9      // Vectorize the multiplication and rounding
10 #pragma vector aligned
11     for (int i = ii; i < ii + vecLen; i++)
12       index[i-ii] = (int) ( age[i] * invGroupWidth );
13     // Scattered memory access, does not get vectorized
14     for (int c = 0; c < vecLen; c++)
15       hist[index[c]]++;
16   }
17 }
```


Strip-Mining for Vectorization

Vectorization improves performance on both platforms. However, more work is needed to take advantage of the MIC architecture. See materials on multi-threading.



Histogram Calculation Example: Adding Thread Parallelism

Incorrect solution: unprotected data races

```
1  #pragma omp parallel for schedule(guided)
2  for (int ii = 0; ii < n; ii += vecLen) {
3    int index[vecLen] __attribute__((aligned(64)));
4  #pragma vector aligned
5    for (int i = ii; i < ii + vecLen; i++)
6      index[i-ii] = (int) ( age[i] * invGroupWidth );
7    for (int c = 0; c < vecLen; c++)
8      // Multiple threads will write into a single shared container
9      // These data races lead to incorrect results!
10     hist[index[c]]++;
11 }
```


Histogram Calculation Example: Adding Thread Parallelism

Correct, but inefficient solution:

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Protect the ++ operation with the atomic mutex (inefficient!)
#pragma omp critical
    { hist[index[c]]++; }
}
```


Histogram Calculation Example: Adding Thread Parallelism

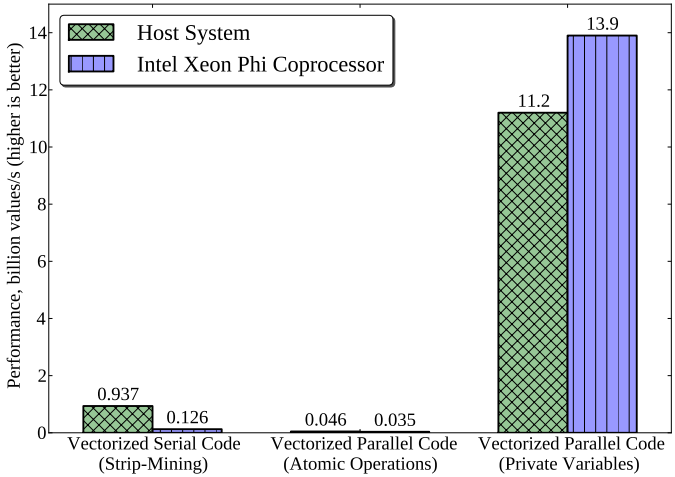
Correct, but inefficient solution:

```
#pragma omp parallel for schedule(guided)
for (int ii = 0; ii < n; ii += vecLen) {
  int index[vecLen] __attribute__((aligned(64)));
#pragma vector aligned
  for (int i = ii; i < ii + vecLen; i++)
    index[i-ii] = (int) ( age[i] * invGroupWidth );
  for (int c = 0; c < vecLen; c++)
    // Protect the ++ operation with the atomic mutex (inefficient!)
#pragma omp atomic
    hist[index[c]]++;
}
```


Correct and Efficient Solution with Reduction

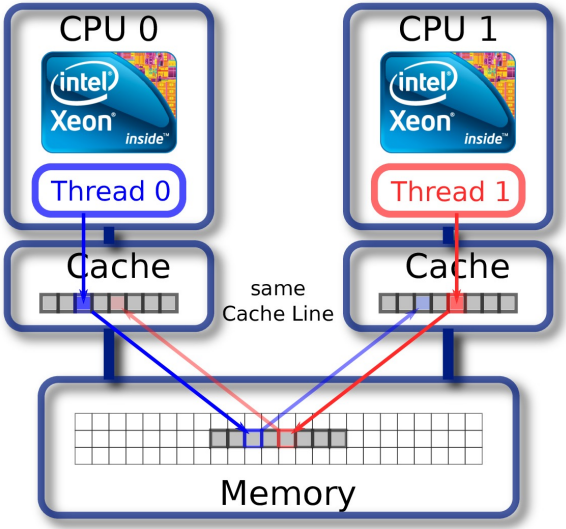
```
#pragma omp parallel
{
  int hist_priv[m]; // Better idea: thread-private storage
  hist_priv[:] = 0;
  int index[vecLen] __attribute__((aligned(64)));
#pragma omp for schedule(guided)
  for (int ii = 0; ii < n; ii += vecLen) {
#pragma vector aligned
    for (int i = ii; i < ii + vecLen; i++)
      index[i-ii] = (int) ( age[i] * invGroupWidth );
    for (int c = 0; c < vecLen; c++)
      hist_priv[index[c]]++;
  }
  for (int c = 0; c < m; c++) {
#pragma omp atomic
    hist[c] += hist_priv[c];
} }
```


Using Reduction instead of Synchronization



False Sharing

False Sharing. Data Padding and Private Variables



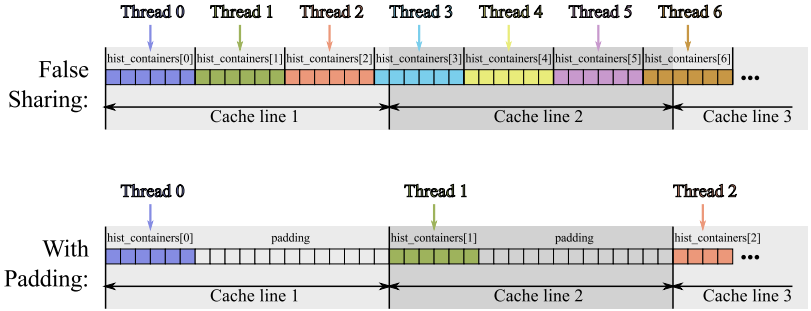
- False sharing is similar to race condition
- Threads accessing the same cache line
- Caused by coherent caches
- Cache line is 64-byte wide (in modern Intel architectures)

False Sharing. Data Padding and Private Variables

```
1  const int m = 5;
2  int hist_thr[nThreads][m];
3  #pragma omp parallel for
4  for (int ii = 0; ii < n; ii += vecLen) {
5    // ...
6    // False sharing occurs here
7    for (int c = 0; c < vecLen; c++)
8      hist_thr[iThread][index[c]]++;
9  }
10 // Reducing results from all threads to the common histogram hist
11 for (int iThread = 0; iThread < nThreads; iThread++)
12   hist[0:m] += hist_thr[iThread][0:m];
```

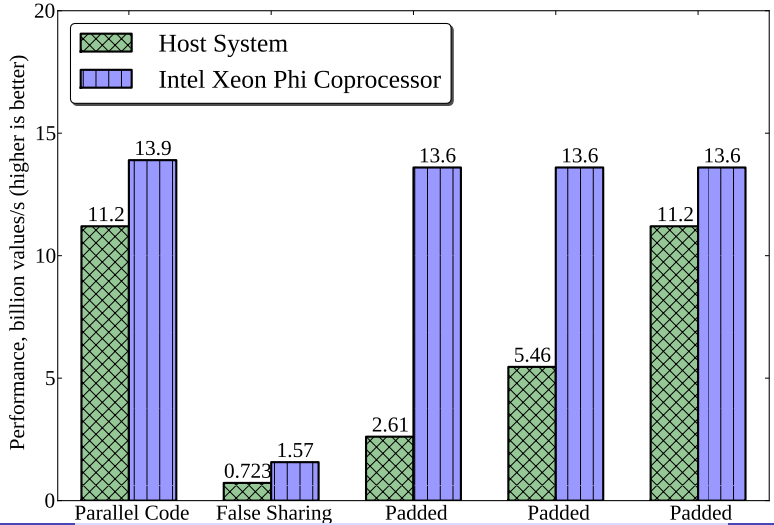
- The value of `m=5` is small
- Array elements `hist_thr[0][:]` are within `m*sizeof(int)=20` bytes of array elements `hist_thr[1][:]`

Padding to Avoid False Sharing




```
1 // Padding to avoid sharing a cache line between threads
2 const int paddingBytes = 64;
3 const int paddingElements = paddingBytes / sizeof(int);
4 const int mPadded = m + (paddingElements-m%paddingElements);
5 int hist_containers[nThreads][mPadded]; // New container
```


Padding to Avoid False Sharing



Insufficient Parallelism

Example: Dealing with Insufficient Parallelism

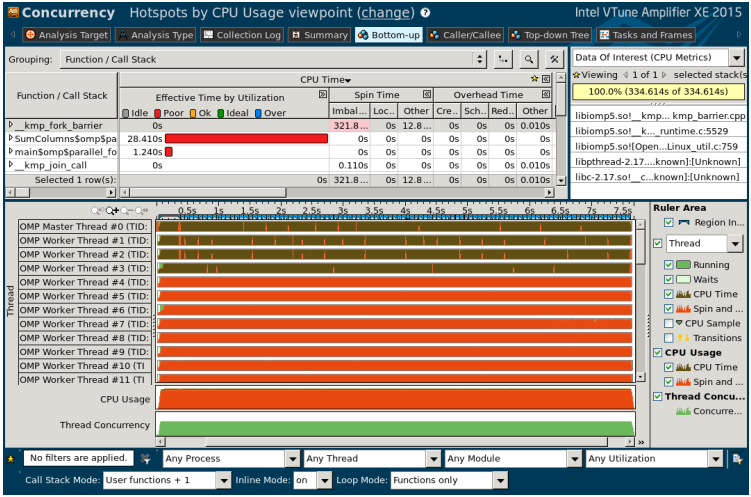
$$S_i = \sum_{j=0}^{n} M_{ij},\ i = 0 \ldots m. \tag{1}$$

- `m=4` is small, smaller than the number of threads in the system
- `n` $\approx 10^8$ is large enough so that matrix does not fit into cache

```
1 void sum_unoptimized(const int m, const int n, long* M, long* s){
2 #pragma omp parallel for
3   for (int i=0; i<m; i++) { // m=4
4     long sum=0;
5 #pragma vector aligned
6     for (int j=0; j<n; j++) // n=100000000
7       sum+=M[i*n+j];
8   s[i]=sum; }}
```


Dealing with Insufficient Parallelism

VTune Analysis: Row-Wise Reduction of a Short, Wide Matrix



Does not Work: Parallelizing Inner Loop

Inner loop has more iterations, parallelize there?

```
1  void SumParallelInnerLoop(const int m, const int n, long* M, long* s){
2    for (int i = 0; i < m; i++) { // m=4
3      long sumOfColumns = 0;
4  #pragma omp parallel for reduction(+: sumOfColumns)
5      for (int j = 0; j < n; j++) { // n=100000000
6        sumOfColumns += M[i*n + j];
7      }
8      s[i] = sumOfColumns;
9    }
10 }
```

Does not work well: code must spawn and stop threads many times; OpenMP does not see the entire parallel region.

Loop Collapse: Principle

Idea: combine iterations spaces of the inner loop and the outer loop.

```
1 #pragma omp parallel for collapse(2)
2   for (int i = 0; i < m; i++)
3     for (int j = 0; j < n; j++) {
4       // ...
5       // ...
6     }
```

```
1 #pragma omp parallel for
2   for (int c = 0; c < m*n; c++)  {
3     i = c / n;
4     j = c % n;
5     // ...
6   }
```

Does not Work, but Correct Direction: Loop Collapse

Loop collapse applied to the wide short matrix example:

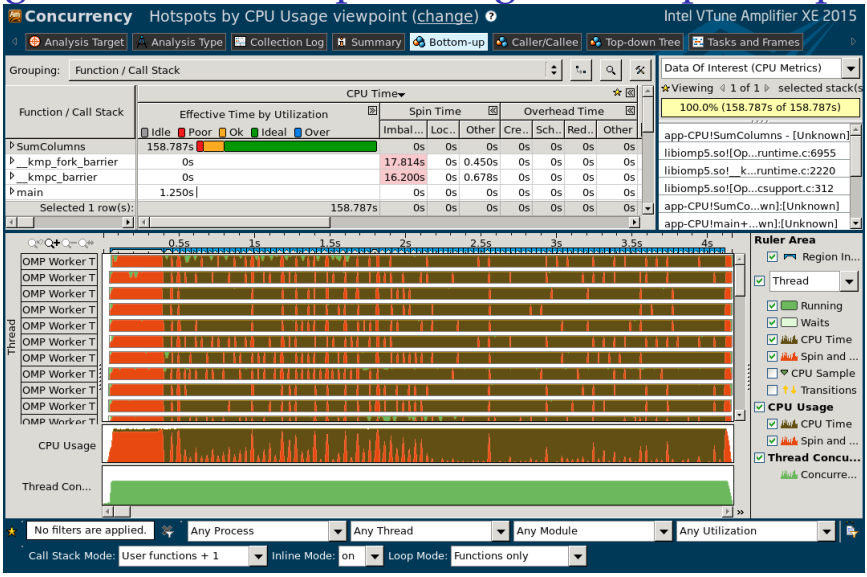
```
1  void SumCollapse(const int m, const int n, long* M, long* s){
2    s[:]=0;
3  #pragma omp parallel
4    { // Each thread will have a private container
5      long sumOfColumns[m];  sumOfColumns[:] = 0;
6  #pragma omp for collapse(2)
7      for (int i = 0; i < m; i++) // m=4
8        for (int j = 0; j < n; j++) // n=100000000
9          sumOfColumns[i] += M[i*n + j];
10     for (int i = 0; i < m; i++)
11 #pragma omp atomic
12       s[i]=sumOfColumns[i];
13 } }
```

Does not work: automatic vectorization fails.

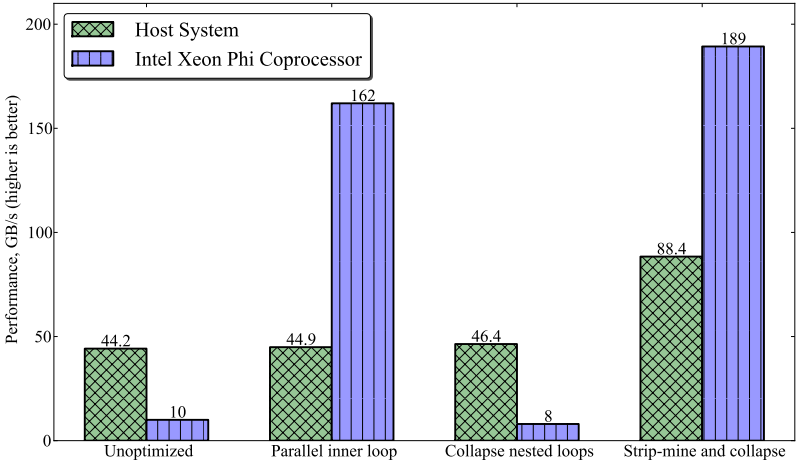
Exposing Parallelism: Strip-Mining and Loop Collapse

```
1  void sum_stripmine(const int m, const int n, long* M, long* s){
2    const int STRIP=1024;
3    assert(n%STRIP==0);
4    s[0:m]=0;
5  #pragma omp parallel
6    {
7      long sum[m];   sum[0:m]=0;
8  #pragma omp for collapse(2) schedule(guided)
9      for (int i=0; i<m; i++)
10       for (int jj=0; jj<n; jj+=STRIP)
11 #pragma vector aligned
12         for (int j=jj; j<jj+STRIP; j++)
13           sum[i]+=M[i*n+j];
14     for (int i=0; i<m; i++)          // Reduction
15 #pragma omp atomic
16       s[i]+=sum[i];
17 }   }
```


Exposing Parallelism: Strip-Mining and Loop Collapse



Dealing with Insufficient Parallelism



Techniques that did not work well are discussed in the book.

§4. Review and What's Next

Summary

This session:

1. Synchronization is necessary to resolve data races, but mutexes must be moved out of innermost loops
2. False sharing can be resolved with padding
3. Strip-mining can help to expose parallelism

Next session: optimization of thread affinity, NUMA locality, nested parallelism and loop scheduling.

