Programming and Optimization for Intel® Architecture

The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl

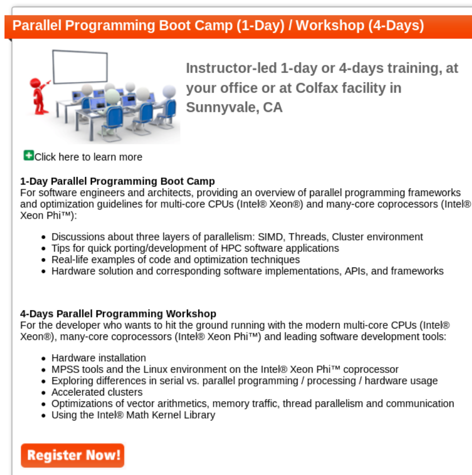
April 2016 , Rev. 02b

 © Colfax International, 2013–2016

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.

© Colfax International, 2013–2016



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 – April 18
   - Programming model continuity – April 18
2. Programming models for Xeon Phi coprocessors
   - Native programming – April 18
   - Offload programming – April 19
3. Expressing Parallelism
   - Introduction to vectorization – April 20
   - Crash-course on OpenMP – April 21
4. Optimization – intro on April 22
   - Vectorization tuning – April 25
   - Multi-threading – April 26, 27
   - Memory traffic – April 28
5. Distributed Computing: MPI – April 29

April 2016

S	M	T	W	H	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

— Lecture+remote access

May 2016

S	M	T	W	H	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

— Self-study/remote access

© Colfax International, 2013–2016

HOW Online

Course page: colfaxresearch.com/how-16-04

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

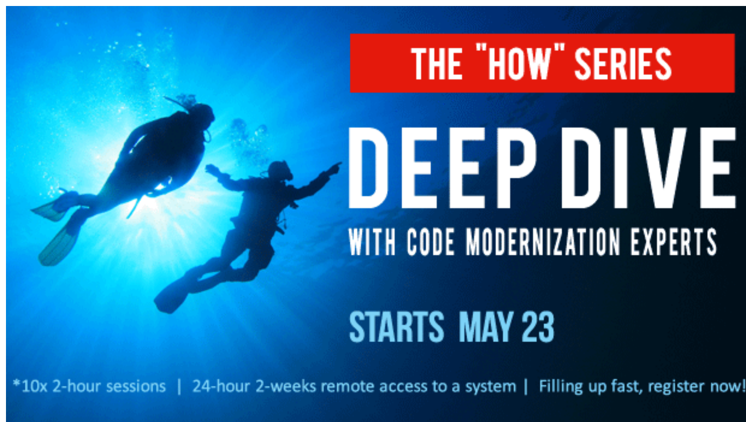
Additional resources:

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

© Colfax International, 2013–2016

HOW Series “Deep Dive” in May

Wish you had joined us earlier? Want to recomment the HOW series to a friend? Another HOW Series run coming up in May.



colfaxresearch.com/how-series/

 © Colfax International, 2013–2016

HOW Series “Tools”

http://colfaxresearch.com/how-tools/

 © Colfax International, 2013–2016

§2. Refresh

© Colfax International, 2013–2016

Performance Optimization

© Colfax International, 2013–2016

Computing Platforms

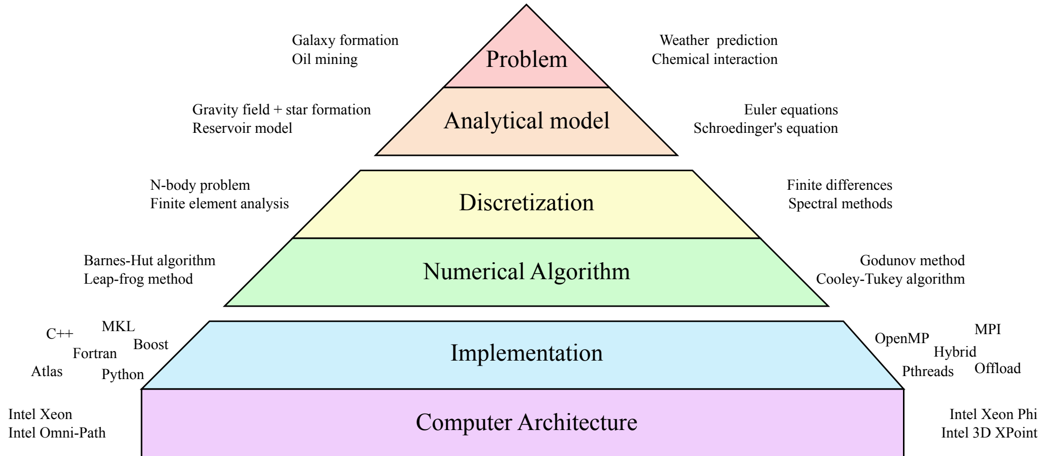
Intel Xeon Processor	Intel Xeon Phi Corocessor, 1st generation	Intel Xeon Phi Processor, 2nd generation*
		* socket and coprocessor versions
Current: Broadwell Upcoming: Skylake	Current: Knights Corner (KNC)	Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

© Colfax International, 2013–2016

Computing in Science and Engineering



 © Colfax International, 2013–2016

Optimization Areas

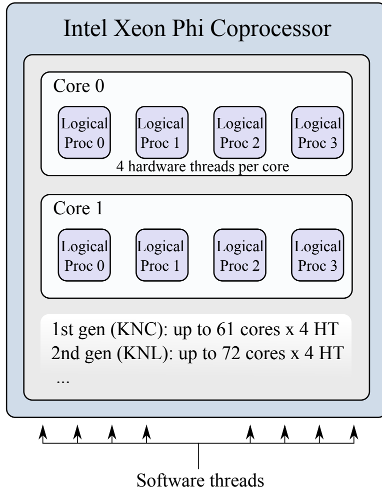
1. **Scalar optimization** (compiler-friendly practices)
2. **Vectorization** (must use 16- or 8-wide vectors)
3. **Multi-threading** (must scale to 100+ threads)
4. **Memory access** (streaming access or tiling)
5. **Communication** (offload, MPI traffic control)

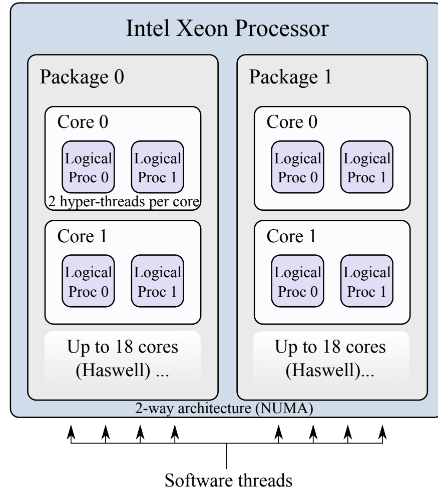
© Colfax International, 2013–2016

Cores, Threads and OpenMP

© Colfax International, 2013–2016

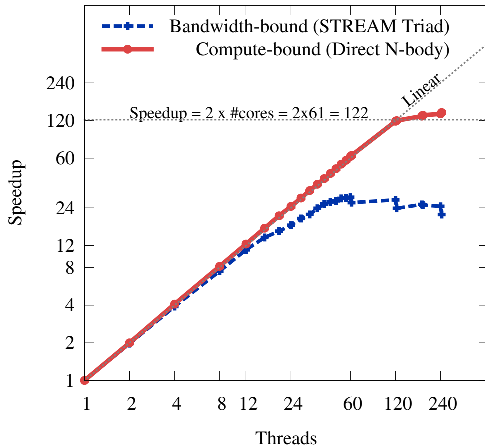
Processor Hierarchy

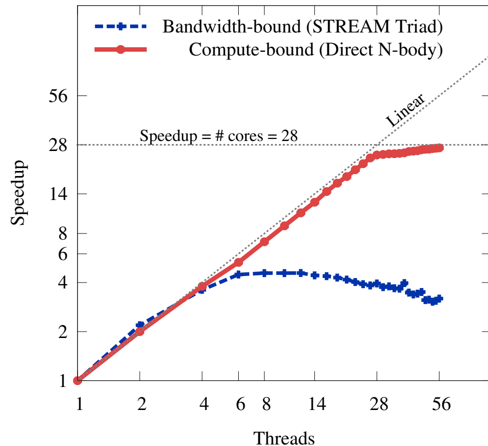




 © Colfax International, 2013–2016

Scalability Expectations: MIC versus CPU





© Colfax International, 2013–2016

“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 }
```


© Colfax International, 2013–2016

§3. Optimization of Multi-Threading I

 © Colfax International, 2013–2016

Too Much Synchronization

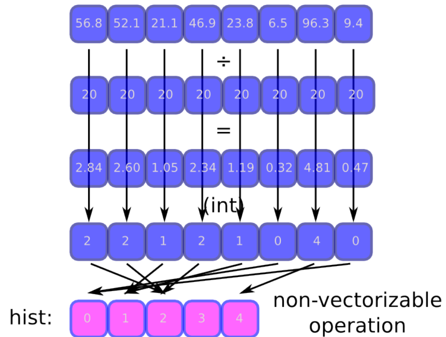
 © Colfax International, 2013–2016

Example: Strip-Mining for Vectorization

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

```
void Histogram(
  // Ages, values from 0.0f to 100.0f:
  const float* age,
  // Size of array age, n=100000000:
  const int n,
  // Output: counts in groups:
  int* const hist,
  // Size of array hist, m=5:
  const int m,
  // group_width=20.0f
  const float group_width) {
    for (int i = 0; i < n; i++) {
      const int j = int(age[i]/group_width);
      hist[j]++;
    }
}
```

- Code cannot be auto-vectorized
- True vector dependence



 © Colfax International, 2013–2016

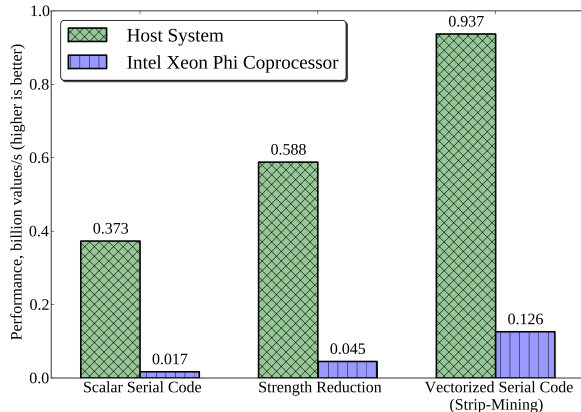
The Same Calculation, Strip-Mined, Vectorized

```
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 }
```

 © Colfax International, 2013–2016

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.



 © Colfax International, 2013–2016

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 }
```


© Colfax International, 2013–2016

Histogram Calculation Example: Adding Thread Parallelism

Correct, but inefficient solution:

```
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      // Protect the ++ operation with the atomic mutex (inefficient!)
9  #pragma omp critical
10     { hist[index[c]]++; }
11 }
```


© Colfax International, 2013–2016

Histogram Calculation Example: Adding Thread Parallelism

Correct, but inefficient solution:

```
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      // Protect the ++ operation with the atomic mutex (inefficient!)
9  #pragma omp atomic
10     hist[index[c]]++;
11 }
```

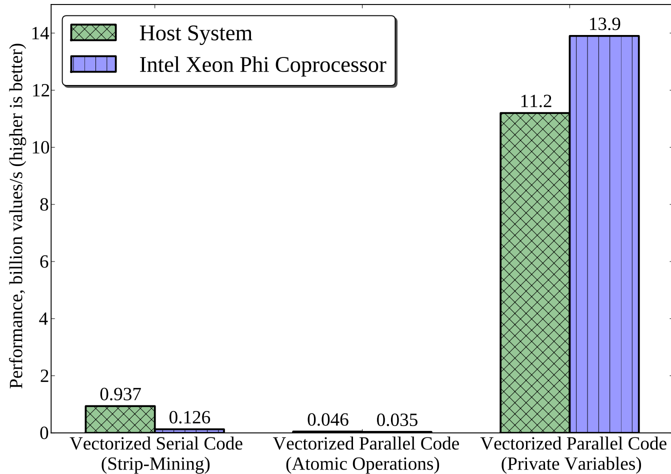
 © Colfax International, 2013–2016

Correct and Efficient Solution with Reduction

```
1  #pragma omp parallel
2  {
3    int hist_priv[m]; // Better idea: thread-private storage
4    hist_priv[:] = 0;
5    int index[vecLen] __attribute__((aligned(64)));
6  #pragma omp for schedule(guided)
7    for (int ii = 0; ii < n; ii += vecLen) {
8  #pragma vector aligned
9      for (int i = ii; i < ii + vecLen; i++)
10       index[i-ii] = (int) ( age[i] * invGroupWidth );
11     for (int c = 0; c < vecLen; c++)
12       hist_priv[index[c]]++;
13   }
14   for (int c = 0; c < m; c++) {
15 #pragma omp atomic
16     hist[c] += hist_priv[c];
17 } }
```


© Colfax International, 2013–2016

Using Reduction instead of Synchronization

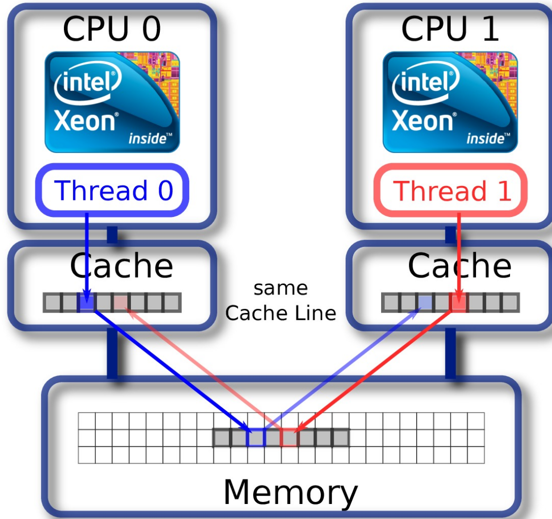


 © Colfax International, 2013–2016

False Sharing

 © Colfax International, 2013–2016

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)

 © Colfax International, 2013–2016

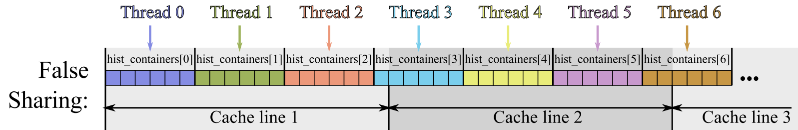
False Sharing. Data Padding and Private Variables

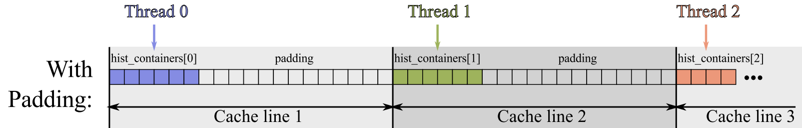
```
1  const int m = 5;
2  int hist_thr[nThreads][m];
3  #pragma omp parallel for
4  for (int ii = 0; ii < n; ii += vecLen) {
5    // ...
6    // False sharing occurs here
7    for (int c = 0; c < vecLen; c++)
8      hist_thr[iThread][index[c]]++;
9  }
10 // Reducing results from all threads to the common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

 © Colfax International, 2013–2016

Padding to Avoid False Sharing

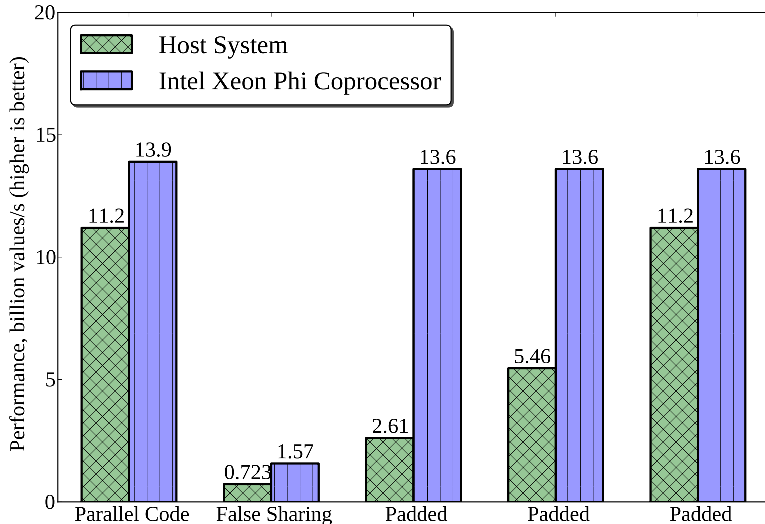





```
1 // Padding to avoid sharing a cache line between threads
2 const int paddingBytes = 64;
3 const int paddingElements = paddingBytes / sizeof(int);
4 const int mPadded = m + (paddingElements-m%paddingElements);
5 int hist_containers[nThreads][mPadded]; // New container
```

 © Colfax International, 2013–2016

Padding to Avoid False Sharing



 © Colfax International, 2013–2016

Insufficient Parallelism

© Colfax International, 2013–2016

Example: Dealing with Insufficient Parallelism

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

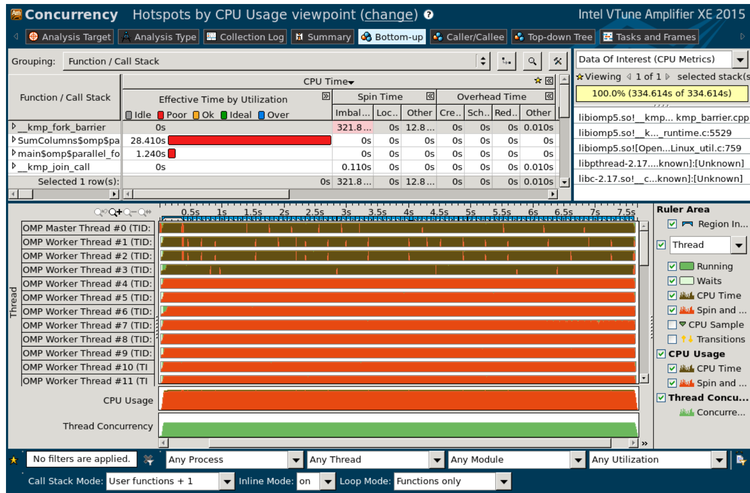
- m=4 is small, smaller than the number of threads in the system
- n $\approx 10^8$ is large enough so that matrix does not fit into cache

```
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; }}
```

 © Colfax International, 2013–2016

Dealing with Insufficient Parallelism

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



 © Colfax International, 2013–2016

Does not Work: Parallelizing Inner Loop

Inner loop has more iterations, parallelize there?

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

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

 © Colfax International, 2013–2016

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   }
```


© Colfax International, 2013–2016

Does not Work, but Correct Direction: Loop Collapse

Loop collapse applied to the wide short matrix example:

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

Does not work: automatic vectorization fails.

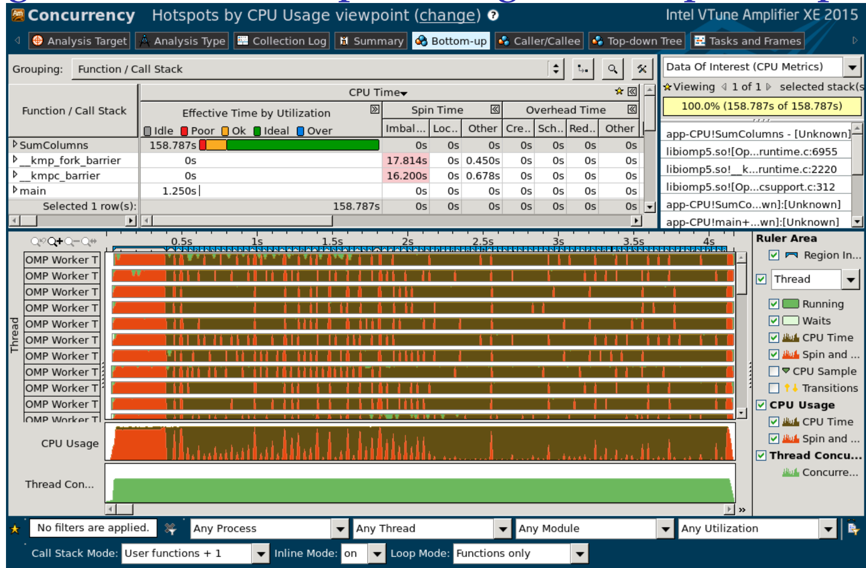
 © Colfax International, 2013–2016

Exposing Parallelism: Strip-Mining and Loop Collapse

```
void sum_stripmine(const int m, const int n, long* M, long* s){
  const int STRIP=1024;
  assert(n%STRIP==0);
  s[0:m]=0;
#pragma omp parallel
  {
    long sum[m];   sum[0:m]=0;
#pragma omp for collapse(2) schedule(guided)
    for (int i=0; i<m; i++)
      for (int jj=0; jj<n; jj+=STRIP)
#pragma vector aligned
        for (int j=jj; j<jj+STRIP; j++)
          sum[i]+=M[i*n+j];
    for (int i=0; i<m; i++)          // Reduction
#pragma omp atomic
      s[i]+=sum[i];
}   }
```

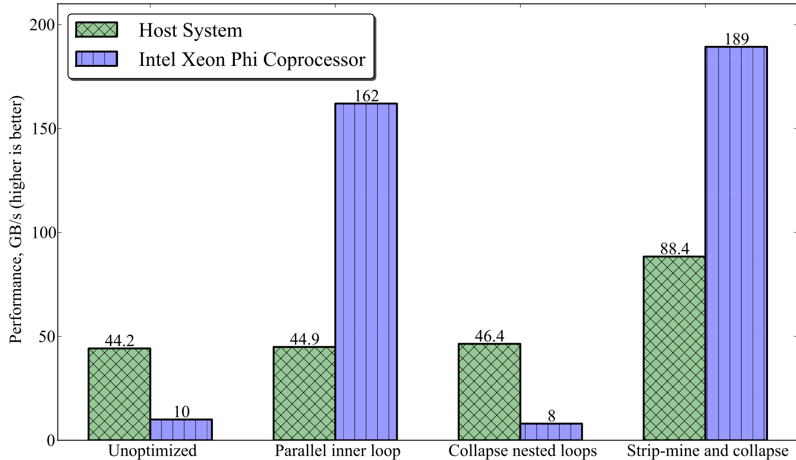
 © Colfax International, 2013–2016

Exposing Parallelism: Strip-Mining and Loop Collapse



 © Colfax International, 2013–2016

Dealing with Insufficient Parallelism



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

 © Colfax International, 2013–2016

§4. Review and What's Next

© Colfax International, 2013–2016

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.

© Colfax International, 2013–2016