PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

The Hands-On Workshop (HOW) Series

Session 7

Colfax International — *colfaxresearch.com*

September 2016

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

- HOW to Program Intel Architecture
  - 01. Parallelism, specialization, guided tour – Sep 26
  - 02. Programming Intel Xeon Phi (KNC, KNL) – Sep 27
- HOW to Express Parallelism
  - 03. Automatic vectorization – Sep 28
  - 04. Multi-threading with OpenMP – Sep 29
- HOW to Get Performance
  - 05. Comprehensive demo – Sep 30
  - 06. Scalar & vectorization tuning – Oct 3
  - 07. Multi-threading: common issues – Oct 4
  - 08. Multi-threading: memory aspect – Oct 5
  - 09. Memory traffic – Oct 6
- HOW to Scale
  - 10. Distributed Computing: MPI – Oct 7

September 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	

October 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					

— Webinar+remote access

HOW ONLINE

Course page: colfaxresearch.com/how-16-09

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

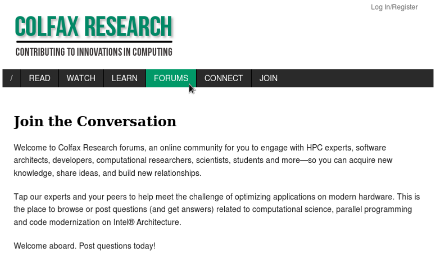
Additional resources:

- More workshops like this one: colfaxresearch.com/training
- Video courses: colfaxresearch.com/video-courses

GET YOUR QUESTIONS ANSWERED

Chat (current):
colfaxresearch.com/how-16-09

Forums (technical):
colfaxresearch.com/discussion

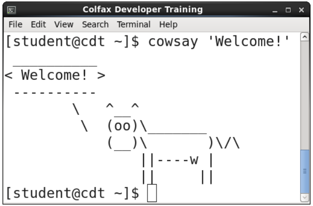


Email (organizational):

`training@colfax-intl.com`

HANDS-ON EXERCISES AND REMOTE ACCESS

- 96 people receive a remote access token
- Virtualized Intel Xeon CPU, real Intel Xeon Phi coprocessor (1st gen, KNC), SW tools
- Can access the system the entire 2 weeks of the workshop

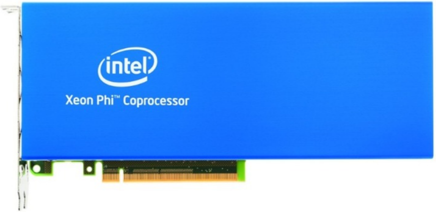


- Not among the 96? Stay tuned: follow along with instructor, use own system, or wait for a seat
- Use it or lose it: if you do not log in for a while, remote access token goes to next student on the list

§2. REFRESH

PERFORMANCE OPTIMIZATION

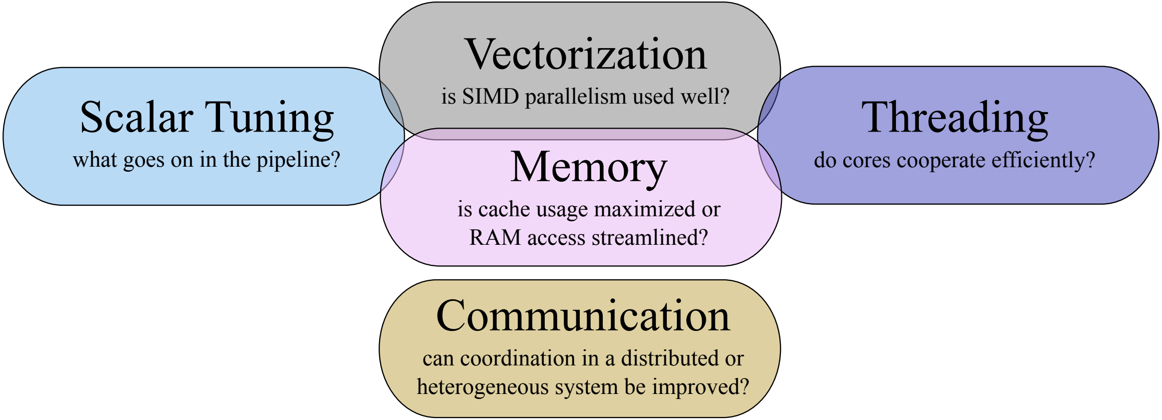
COMPUTING PLATFORMS

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

Multi-Core Architecture — Intel Xeon

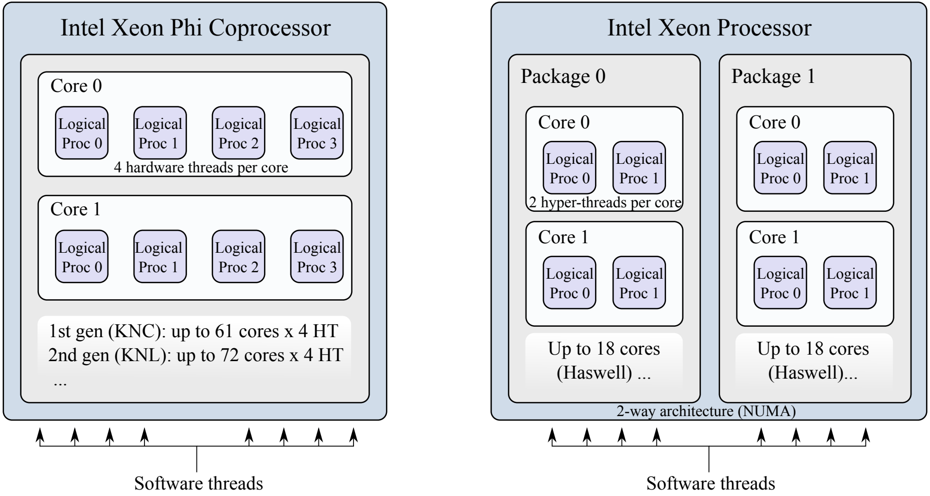
Intel Many Integrated Core (MIC) Architecture — Intel Xeon Phi (KNC, KNL)

OPTIMIZATION AREAS



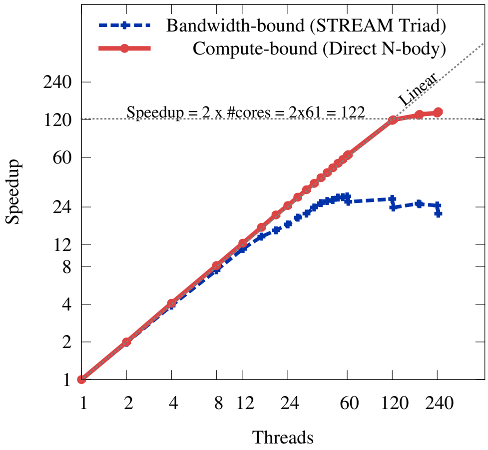
CORES, THREADS AND OPENMP

PROCESSOR HIERARCHY

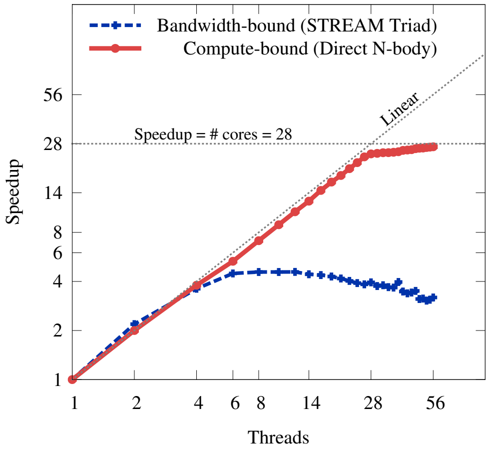


Scalability Expectations: MIC versus CPU

Performance on the MIC architecture



Performance on the CPU architecture



"HELLO WORLD" OPENMP PROGRAM

```
#include <omp.h>
#include <cstdio>

int main(){
  // This code is executed by only 1 thread
  const int nt=omp_get_max_threads();
  printf("OpenMP with %d threads\n", nt);

#pragma omp parallel
  {
    // This code is executed in parallel
    // by multiple threads
    printf("Hello World from thread %d\n",
                          omp_get_thread_num());
  }
}
```

- OpenMP = "Open Multi-Processing" = computing-oriented framework for shared-memory programming
- Threads – streams of instructions that share memory address space
- Distribute threads across CPU cores for parallel speedup

§3. MULTI-THREADING: COMMON ISSUES

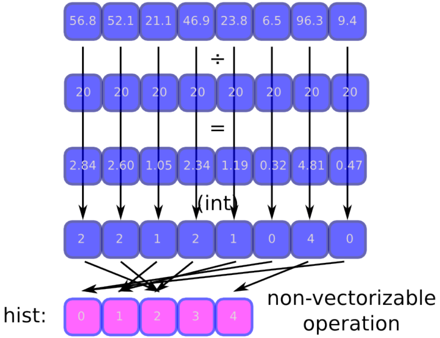
TOO MUCH SYNCHRONIZATION

EXAMPLE: BINNING PROBLEM

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,
  const float group_width) {
    for (int i = 0; i < n; i++) {
      const int j = int(age[i]/group_width);
      hist[j]++;
    }
}
```

- Vector dependence in `hist[j]++`
- Strip-mine or use conflict detection

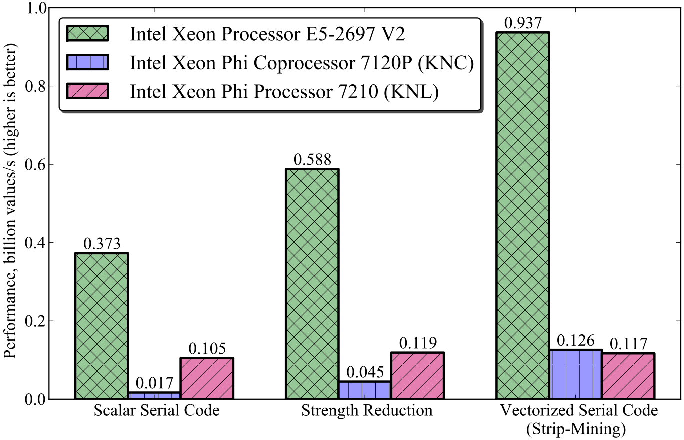


THE SAME CALCULATION, STRIP-MINED, VECTORIZED

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

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

```
#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++)
    // Multiple threads will write into a single shared container
    // These data races lead to incorrect results!
    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 critical
    { hist[index[c]]++; }
}
```

HISTOGRAM CALCULATION EXAMPLE: ADDING THREAD PARALLELISM

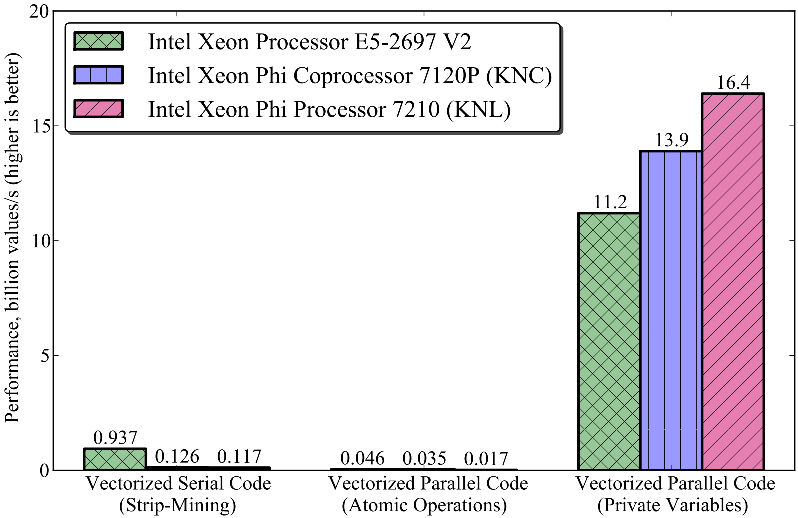
Correct, but inefficient solution:

```
#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

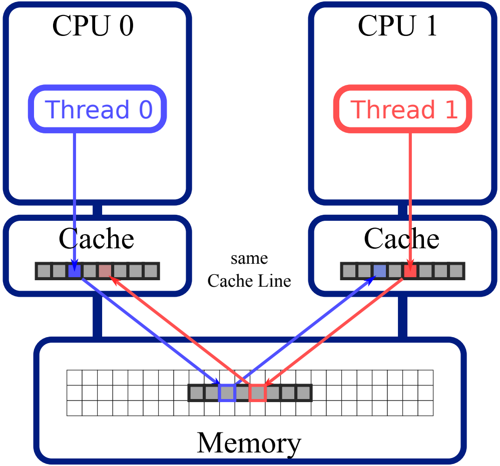
```
#pragma omp parallel
{
  int hist_priv[m]; // Better idea: thread-private storage
  hist_priv[:] = 0;
  int index[vecLen] __attribute__((aligned(64)));
#pragma omp for schedule(guided)
  for (int ii = 0; ii < n; ii += vecLen) {
#pragma vector aligned
    for (int i = ii; i < ii + vecLen; i++)
  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



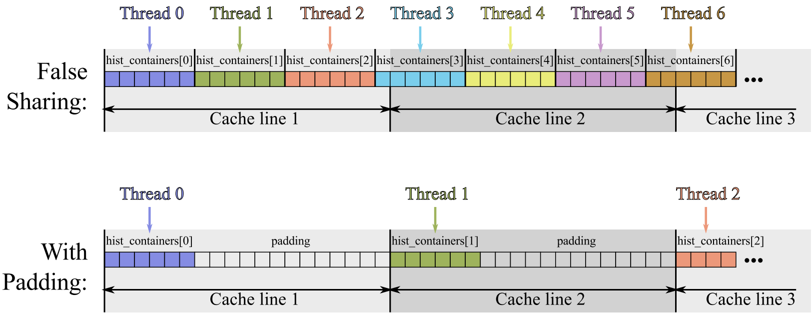
- Occurs when 2 or more threads acess the same cache line, and at least one of the accesses is for writing
- Caused by *coherent caches*
- Cache line is 64-byte wide (in modern Intel architectures)

FALSE SHARING. DATA PADDING AND PRIVATE VARIABLES

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

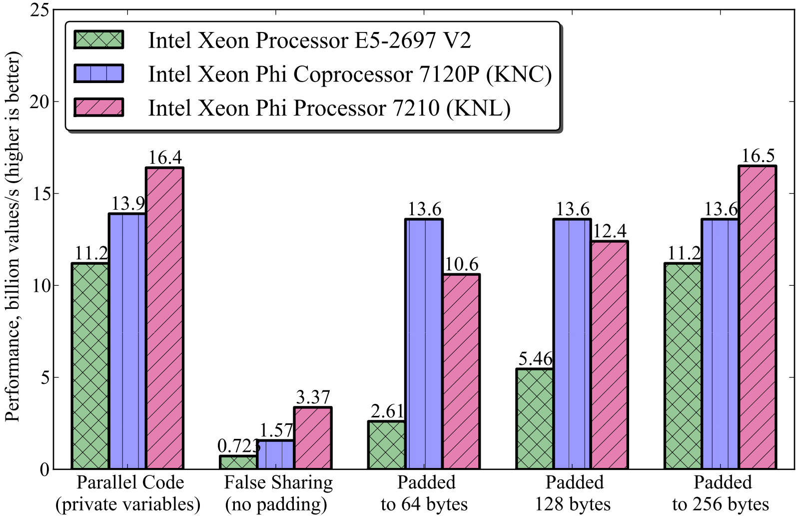
- The value of `m=5` is small
- Array elements `hist_thr[0][:]` are within `m*sizeof(int)=20` bytes of array elements `hist_thr[1][:]`

PADDING TO AVOID FALSE SHARING



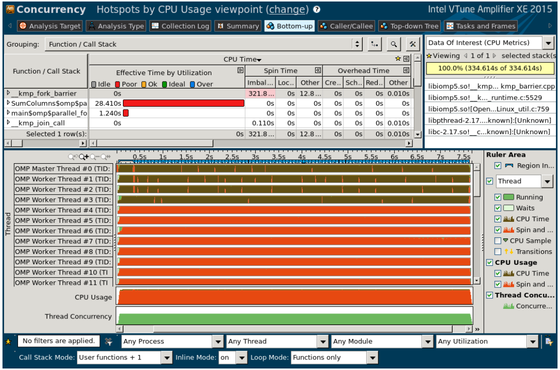

```
// Padding to avoid sharing a cache line between threads
const int paddingBytes = 64;
const int paddingElements = paddingBytes / sizeof(int);
const int mPadded = m + (paddingElements-m%paddingElements);
int hist_containers[nThreads][mPadded]; // New container
```

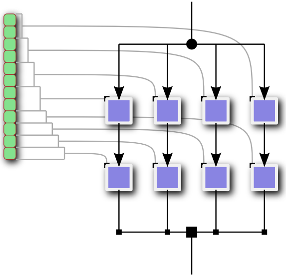
PADDING TO AVOID FALSE SHARING



INSUFFICIENT PARALLELISM

Analysis in Intel VTune Amplifier XE





▷ Occurs when there are not enough iterations or parallel work-items exposed to the parallel loop in OpenMP.

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

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

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 total = 0;
#pragma omp parallel for reduction(+: total)
    for (int j = 0; j < n; j++) { // n=100000000
      total += M[i*n + j];
    }
    s[i] = total;
  }
}
```

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.

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

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

DOES NOT WORK, BUT CORRECT DIRECTION: LOOP COLLAPSE

Loop collapse applied to the wide short matrix example:

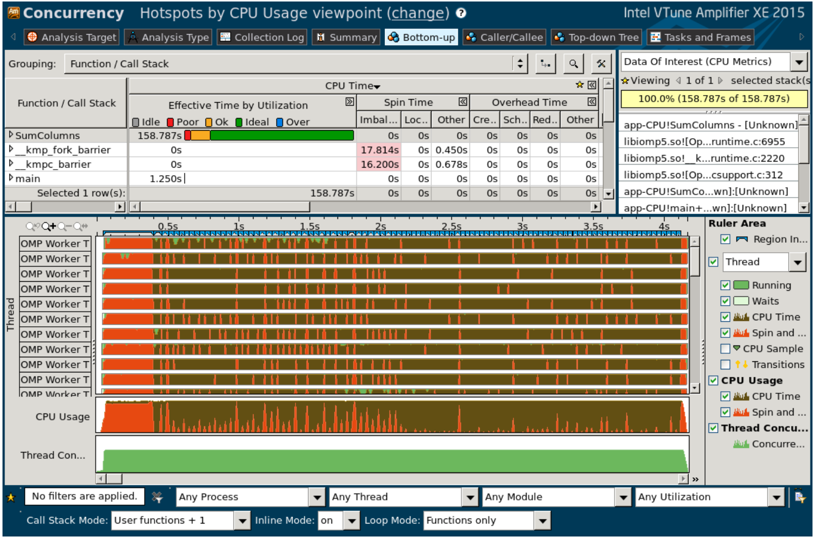
```
void SumCollapse(const int m, const int n, long* M, long* s){
  s[:]=0;
#pragma omp parallel
  { // Each thread will have a private container
    long total[m];  total[:] = 0;
#pragma omp for collapse(2)
    for (int i = 0; i < m; i++) // m=4
      for (int j = 0; j < n; j++) // n=100000000
    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

Does not work: automatic vectorization fails.

EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE

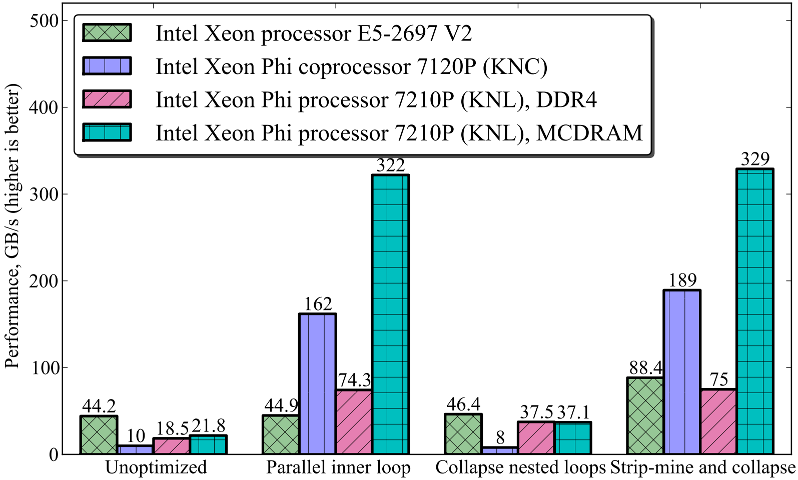
```
void sum_stripmine(const int m, const int n, long* M, long* s){
  const int STRIP=1024;
  assert(n%STRIP==0);
  s[0:m]=0;
#pragma omp parallel
  {
    long total[m];   total[0:m]=0;
#pragma omp for collapse(2) schedule(guided)
    for (int i=0; i<m; i++)
      for (int jj=0; jj<n; jj+=STRIP)
#pragma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

EXPOSING PARALLELISM: STRIP-MINING AND LOOP COLLAPSE



INSUFFICIENT PARALLELISM

DEALING WITH INSUFFICIENT PARALLELISM



§4. REVIEW AND WHAT'S NEXT

SUMMARY

This session:

1. Synchronization is necessary to resolve data races
2. Mutexes must be moved out of innermost loops
3. False sharing can be resolved with padding
4. Loop collapse can help to expose parallelism
5. Strip-mining to make vectorization co-exist with threading

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

LEARN MORE

HOW SERIES: KNIGHTS LANDING

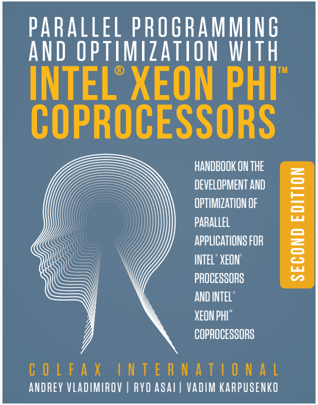
colfaxresearch.com/how-knl/

TEXTBOOK

ISBN: 978-0-9885234-0-1 (508 pages, Electronic or Print)

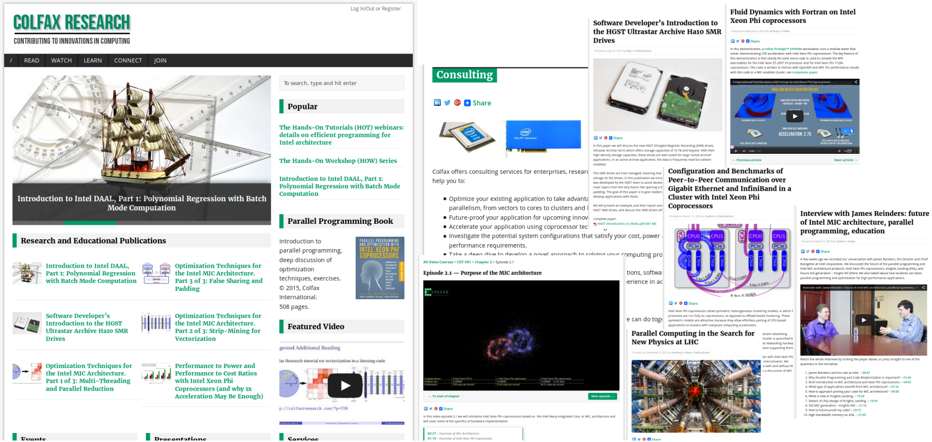
Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors



http://xeonphi.com/book

COLFAX RESEARCH



http://colfaxresearch.com/

LEARN MORE