



# Programming and Optimization for Intel<sup>®</sup> Architecture

The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl


August 2016 , Rev. 03b

# 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

Parallel Programming Boot Camp (1-Day) / Workshop (4-Days)



Instructor-led 1-day or 4-days training, at your office or at Colfax facility in Sunnyvale, CA

[Click here to learn more](#)

**1-Day Parallel Programming Boot Camp**  
 For software engineers and architects, providing an overview of parallel programming frameworks and optimization guidelines for multi-core CPUs (Intel® Xeon®) and many-core coprocessors (Intel® Xeon Phi™):

- Discussions about three layers of parallelism: SIMD, Threads, Cluster environment
- Tips for quick porting/development of HPC software applications
- Real-life examples of code and optimization techniques
- Hardware solution and corresponding software implementations, APIs, and frameworks

**4-Days Parallel Programming Workshop**  
 For the developer who wants to hit the ground running with the modern multi-core CPUs (Intel® Xeon®), many-core coprocessors (Intel® Xeon Phi™) and leading software development tools:

- Hardware installation
- MPSS tools and the Linux environment on the Intel® Xeon Phi™ coprocessor
- Exploring differences in serial vs. parallel programming / processing / hardware usage
- Accelerated clusters
- Optimizations of vector arithmetics, memory traffic, thread parallelism and communication
- Using the Intel® Math Kernel Library

Register Now!

[colfaxresearch.com/how-series](http://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

- HOW to Program Intel Architecture
  - ▶ 01. Parallelism, specialization, guided tour – Aug 29
  - ▶ 02. Programming Intel Xeon Phi (KNC, KNL) – Aug 30
- HOW to Express Parallelism
  - ▶ 03. Automatic vectorization – Aug 31
  - ▶ 04. Multi-threading with OpenMP – Sep 1
- HOW to Get Performance
  - ▶ 05. Comprehensive demo – Sep 2
  - ▶ 06. Scalar & vectorization tuning – Sep 5
  - ▶ 07. Multi-threading I – Sep 6
  - ▶ 08. Multi-threading II – Sep 7
  - ▶ 09. Memory traffic – Sep 8
- HOW to Scale
  - ▶ 10. Distributed Computing: MPI – Sep 9

August 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			

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	

 — Webinar+remote access

# HOW Online

Course page: [colfaxresearch.com/how-16-08](http://colfaxresearch.com/how-16-08)

- 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](http://colfaxresearch.com/training)
- Video courses: [colfaxresearch.com/video-courses](http://colfaxresearch.com/video-courses)

# Get Your Questions Answered

## Chat (current):

[colfaxresearch.com/how-16-08](http://colfaxresearch.com/how-16-08)



## Forums (technical):

[colfaxresearch.com/discussion](http://colfaxresearch.com/discussion)

Log In/Register

### COLFAX RESEARCH

CONTRIBUTING TO INNOVATIONS IN COMPUTING

/ READ WATCH LEARN **FORUMS** CONNECT JOIN

### Join the Conversation

Welcome to Colfax Research forums, an online community for you to engage with HPC experts, software architects, developers, computational researchers, scientists, students and more—so you can acquire new knowledge, share ideas, and build new relationships.

Tap our experts and your peers to help meet the challenge of optimizing applications on modern hardware. This is the place to browse or post questions (and get answers) related to computational science, parallel programming and code modernization on Intel® Architecture.

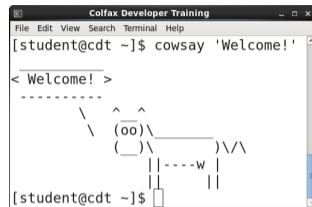
Welcome aboard. Post questions today!

## Email (organizational):

[training@colfax-intl.com](mailto: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



```
Colfax Developer Training
File Edit View Search Terminal Help
[student@cdt ~]$ cowsay 'Welcome!'
< Welcome! >
-----
      \   ^__^
         (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

[student@cdt ~]$
```

## §2. Performance Optimization

# Computing Platforms

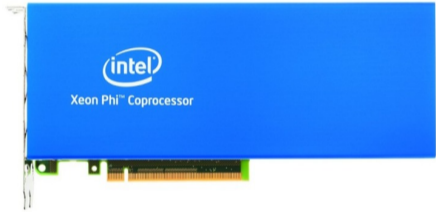
Intel Xeon Processor



Current: Broadwell  
Upcoming: Skylake

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation\*

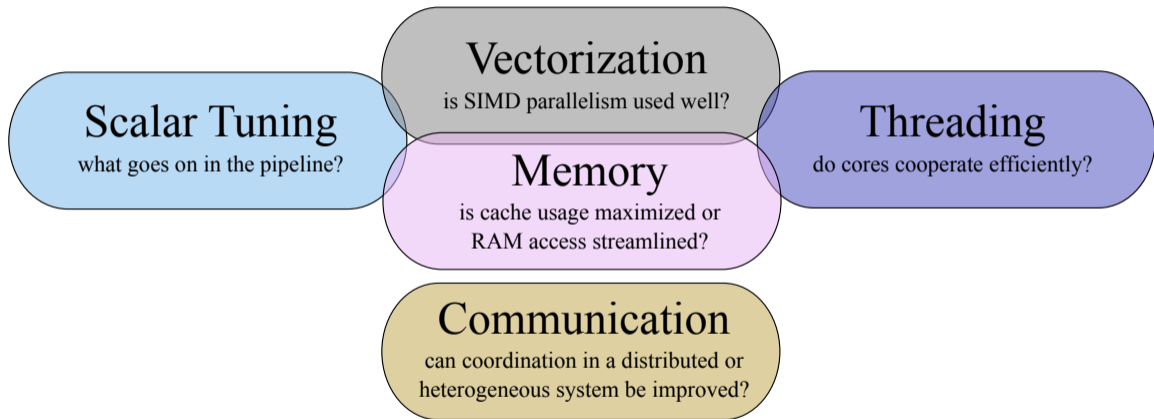


\* socket and coprocessor versions

Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

# Optimization Areas



## §3. Scalar Tuning

# Compiler Arguments

# Optimization Level

## Default optimization level -O2

- optimization for speed
- automatic vectorization
- inlining
- constant propagation
- dead-code elimination
- loop unrolling

## Optimization level -O3

- aggressive optimization
- loop fusion
- block-unroll-and-jam
- if-statement collapse
- *may or may not be better than -O2*

# Setting Optimization Level

For the entire file:

```
vega@lyra% icpc -o mycode -O3 source.cc
```

For a specific function:

```
1  #pragma intel optimization_level 3  
2  void my_function() {  
3      //...  
4  }
```

## Precision Control for Transcendental Functions

- `-fimf-precision=value[:funclist]` Defines the precision for math functions. `value` is one of: `high`, `medium` or `low`
- `-fimf-max-error=ulps[:funclist]` The maximum allowable error expressed in `ulps` (*units in last place*)
- `-fimf-accuracy-bits=n[:funclist]` The number of correct bits required for mathematical function accuracy.
- `-fimf-domain-exclusion=n[:funclist]` Defines a list of special-value numbers that do not need to be handled.  
`int` `n` derived by the bitwise OR of types:  
extremes: 1, NaNs: 2, infinities: 4, denormals<sup>1</sup>: 8, zeroes: 16.

---

<sup>1</sup>by default, on Intel Xeon Phi, denormals are flushed to zero in hardware, but supported in SVML

## Floating-Point Semantics

The Intel C++ Compiler may represent floating-point expressions in executable code differently, depending on the *floating-point semantics*.

<code>-fp-model strict</code>	Only value-safe optimizations calculations are reproducible from run to run exceptions controlled using <code>-fp-model except</code> (default)
<code>-fp-model precise</code>	
<code>-fp-model fast=1</code>	Value-unsafe optimizations are allowed better performance at the cost of lower accuracy
<code>-fp-model fast=2</code>	
<code>-fp-model source</code>	Intermediate arithmetic results are rounded to the precision defined in the source code.
<code>-fp-model double</code>	Intermediate arithmetic results are rounded to 53-bit (double) precision.
<code>-fp-model extended</code>	Intermediate arithmetic results are rounded to 64-bit (extended) precision.
<code>-fp-model [no-]except</code>	controls floating-point exception semantics.

# Programming Practices

# Strength Reduction

## Common Subexpression Elimination.

```

1  for (int i = 0; i < n; i++) {
2      A[i] /= B;
3  }
```

```

1  const float Br = 1.0f/B;
2  for (int i = 0; i < n; i++)
3      A[i] *= Br;
```

## Replace division with multiplication.

```

1  for (int i = 0; i < n; i++) {
2      P[i] = (Q[i]/R[i])/S[i];
3  }
```

```

1  for (int i = 0; i < n; i++) {
2      P[i] = Q[i]/(R[i]*S[i]);
3  }
```

## Use functions with Hardware support.

```

1  double r = pow(r2, -0.5);
2  double v = exp(x);
3  double y = y0*exp(log(x/x0)*
4              log(y1/y0)/log(x1/x0));
```

```

1  double r = 1.0/sqrt(r2);
2  double v = exp2(x*1.44269504089);
3  double y = y0*exp2(log2(x/x0)*
4              log2(y1/y0)/log2(x1/x0));
```

# Consistency of Precision: Constants

```
1 // Bad: 2 is "int"
2 long b=a*2;
3
4 // Bad: overflow
5 long n=100000*100000;
6
7 // Bad: excessive
8 float p=6.283185307179586;
9
10 // Bad: 2 is "int"
11 float q=2*p;
12
13 // Bad: 1e9 is "double"
14 float r=1e9*p;
15
16 // Bad: 1 is "int"
17 double t=s+1;
```

```
1 // Good: 2L is "long"
2 long b=a*2L;
3
4 // Good: correct
5 long n=100000L*100000L;
6
7 // Good: accurate
8 float p=6.283185f;
9
10 // Good: 2.0f is "float"
11 float q=2.0f*p;
12
13 // Good: 1e9f is "float"
14 float r=1e9f*p;
15
16 // Good: 1.0 is "double"
17 double t=s+1.0;
```

# Consistency of Precision: Functions

```
1 // Bad: 3.14 is a double
2 float x = 3.14;
3
4 // Bad: sin() is a
5 // double precision function
6 float s = sin(x);
7
8 // Bad: round() takes double
9 // and returns double
10 long v = round(x);
11
12 // Bad: abs() is not from IML
13 // it takes int and returns int
14 int v = abs(x);
```

```
1 // Good: 3.14f is a float
2 float x = 3.14f;
3
4 // Good: sin() is a
5 // single precision function
6 float s = sinf(x);
7
8 // Good: lroundf() takes float
9 // and returns long
10 long v = lroundf(x);
11
12 // Good: fabsf() is from IML
13 // It takes and returns a float
14 float v = fabsf(x);
```

# Consistency of Precision: Functions

Transcendental functions are *not* overloaded (unless in namespace `std` in C++).

```
vega@lyra% ./Scalar-TestF0verload
```

```
Proof that exp() is not overloaded:
```

```
exp (1.0f)=2.7182818284590451
```

```
exp (1.0 )=2.7182818284590451
```

```
Exact:    e=2.71828182845904523536...
```

```
Proof that expf() gives lower precision:
```

```
expf(1.0f)=2.7182817459106445
```

```
expf(1.0 )=2.7182817459106445
```

```
Exact:    e=2.71828182845904523536...
```

```
Overloading in namespace std:
```

```
std::exp(1.0f)=2.7182817459106445
```

```
std::exp(1.0 )=2.7182818284590451
```

```
Exact:    e=2.71828182845904523536...
```

# Move Branches Outside of Loops

```
1 // Elegant, but bad for performance
2 for (i = 0; i < n; i++) {
3     if (i == 0) {
4         // Absorbing boundary
5         B[i] = 0.0;
6     } else if (i == n - 1) {
7         // Injection at boundary
8         B[i] = B[i] + 1.0;
9     } else {
10        // Diffusion between boundaries
11        B[i] = 0.25*(A[i-1] +
12                    2.0*A[i] + A[i+1]);
13    }
14 }
```

```
1 // Moving branches out of loops
2
3
4 // Absorbing boundary
5 B[i] = 0.0;
6
7 for (i = 1; i < n - 1; i++) {
8     // Diffusion between boundaries
9     B[i] = 0.25*(A[i-1] + 2.0*A[i] +
10                 A[i+1]);
11 }
12
13 // Injection at boundary
14 B[n-1] = B[n-1] + 1.0;
```

# Redundant Code is OK

```
1 // Elegant, but bad for performance
2 for (ii = 0; ii < n; ii+=16) {
3     for (i = ii; i < ii+16; i++)
4         // Branch causes unnecessary
5         // masking of vector iterations
6         if (i < n) {
7             A[k*n + i] = ...
8         }
9 }
```

```
1 // Redundant code, but faster
2 const int nTrunc = n - 16;
3 for (ii = 0; ii < nTrunc; ii+=16) {
4     for (i = ii; i < ii+16; i++)
5         A[k*n + i] = ...
6
7     for (i = nTrunc; i < n; i++)
8         A[k*n + i] = ...
9 }
```

# §4. Vectorization

# Short Vector Support

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

Scalar Instructions

$$\begin{array}{r} 4 + 1 = 5 \\ 0 + 3 = 3 \\ -2 + 8 = 6 \\ 9 + -7 = 2 \end{array}$$

Vector Instructions

$$\begin{array}{r} 4 \\ 0 \\ -2 \\ 9 \end{array} + \begin{array}{r} 1 \\ 3 \\ 8 \\ -7 \end{array} = \begin{array}{r} 5 \\ 3 \\ 6 \\ 2 \end{array}$$

↑  
Vector Length  
↓

# Data Structures and Memory Access

# Unit-Stride Access

Unit-stride access is optimal:

```
1 for (int i = 0; i < n; i++)
2   A[i] += B[i];
```

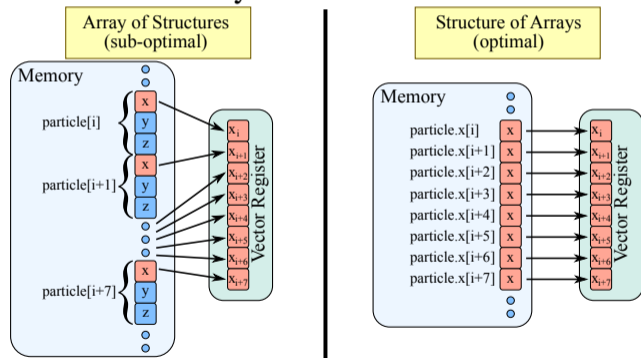
Non-unit stride is slower:

```
1 for (int i = 0; i < n; i++)
2   A[i*stride] += B[i];
```

Stochastic access cannot be vectorized:

```
1 for (int i = 0; i < n; i++)
2   A[offset[i]] += B[i];
```

It may be a question of changing the order of loop nesting, but sometimes you need to modify data structures:



# Suggested Exercise

Review lab 4.01 (N-body simulation) or perform lab 4.02 (Coulomb's law calculation) to re-visit the AoS to SoA conversion.

# Alignment and Padding

# Data Alignment Requirements

Array `char* p` is `n`-byte aligned if `((size_t)p%n==0)`.

Processor	Operation	Alignment
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load, store	64-byte (relaxed)

# Data Alignment

- Data alignment on the stack

```
1 float A[n] __attribute__((aligned(64))); // 64-byte alignment applied
```

- Data alignment on the heap

```
1 float *A = (float*) _mm_malloc(sizeof(float)*n, 64);
```

- A[0] is aligned on a 64-byte boundary.
- Very high alignment value may lead to wasted virtual memory.
- Fortran: directive or compiler argument `-align array64byte`

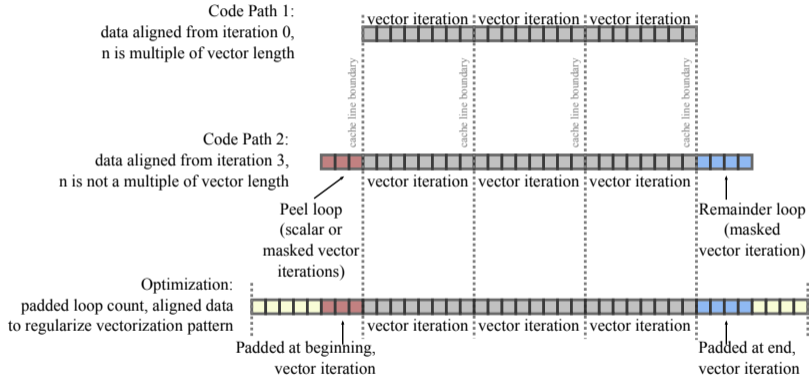
# Padding for Alignment

To use aligned instructions, you may need to pad inner dimension of multi-dimensional arrays to a multiple of 16 (in SP) or 8 (DP) elements.

```
1 // ... Padding inner dimension so that every row is aligned
2 int lda=n;
3 if (n % 16 != 0) lda += (16 - n%16); // now lda%16==0
4 float* A = _mm_malloc(sizeof(float)*n*lda, 64);
5
6 // If A[          ] is aligned AND lda%16==0, then
7 //   A[i*lda + 0] is aligned for any i
8 for (int i = 0; i < n; i++)
9     for (int j = 0; j < n; j++)
10        A[i*lda + j] = ...
```

# Regularizing Vectorization Pattern

```
for (i = 0; i < n; i++)  A[i] = ...
```



# Data Alignment Hints

Programmer may promise to the compiler (under penalty of segmentation fault) that alignment has been taken care of:

```
1 // Promising that A[i*lda + 0] is aligned for every i
2 // and the same for every other array in this loop
3 #pragma vector aligned
4     for (int j = 0; j < n; j++)
5         A[i*lda + j] -= ...
```

This can lead to significant speedups, because compiler will not implement runtime checks for alignment situation and *peel loops*.

# Example: LU Decomposition

# Example: LU Decomposition

```

1 void LU_decomp(const int n, float* const A) {
2     // LU decomposition (Doolittle algorithm)
3     // In-place decomposition of form A=LU
4     // L is returned below main diagonal of A
5     // U is returned at and above main diagonal
6     for (int b = 0; b < n; b++) {
7         // Strength reduction:
8         const float recAbb = 1.0f/A[b*n + b];
9         for (int i = b+1; i < n; i++) {
10            A[i*n + b] = A[i*n + b]*recAbb;
11        #pragma simd
12            for (int j = b+1; j < n; j++)
13                A[i*n + j] -= A[i*n + b]*A[b*n + j];
14        }
15    }
16 }

```

LU decomposition for small matrices. ( $n \approx 128$ )

Based on publication:

<http://xeonphi.com/papers/>

Non-optimal

Vectorization Pattern.

- Unaligned
- Irregular loop count

# LU Decomposition: Regularizing Vectorization

Before:

```

1 for (int b = 0; b < n; b++) {
2     // ...
3     // ...
4     for (int i = b+1; i < n; i++) {
5         // ...
6         for (int j = b+1; j < n; j++)
7             A[i*n+j] -= A[i*n+b]*A[b*n+j];
8     }
9 }

```

After:

```

1 for (int b = 0; b < n; b++) {
2     // ...
3     const int jMin = (b+1) - (b+1)%16;
4     for (int i = b+1; i < n; i++) {
5         // ...
6         for (int j = jMin; j < n; j++)
7             A[i*n+j] -= L[i*n+b]*A[b*n+j];
8     }
9 }

```

Loop in j always starts on a multiple of 64 →  
aligned access to A and L

# LU Decomposition: Compiler hints

- Data alignment hint: `#pragma vector aligned`

Before:

```

1  for (int b = 0; b < n; b++) {
2      const int jMin = (b+1)-(b+1)%tile;
3      const float recAbb = 1.0f/A[b*n+b];
4      for (int i = b+1; i < n; i++) {
5          L[i*n + b] = A[i*n + b]*recAbb;
6
7
8      #pragma simd
9          for (int j = jMin; j < n; j++)
10             A[i*n+j] -= L[i*n+b]*A[b*n+j];
11     }
12 }

```

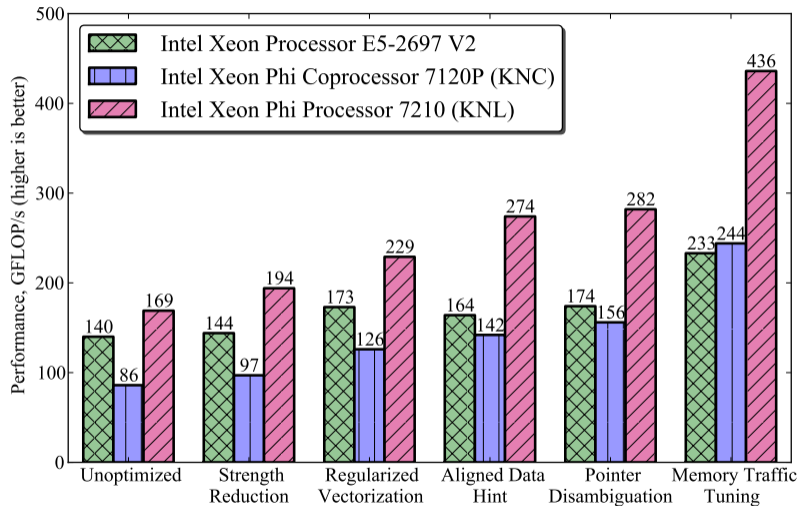
After:

```

1  for (int b = 0; b < n; b++) {
2      const int jMin = (b+1)-(b+1)%tile;
3      const float recAbb = 1.0f/A[b*n+b];
4      for (int i = b+1; i < n; i++) {
5          L[i*n + b] = A[i*n + b]*recAbb;
6          #pragma vector aligned
7          #pragma ivdep
8          #pragma simd
9              for (int j = jMin; j < n; j++)
10                 A[i*n+j] -= L[i*n+b]*A[b*n+j];
11     }
12 }

```

# LU Decomposition: Performance



Paper: <http://xeonphi.com/papers/lu>

## Loop Was Vectorized, Now What?

- 1 Ensure unit stride access
- 2 Align data
- 3 Pad multi-dimensional containers
- 4 Eliminate peel loops
- 5 Eliminate multiversioning
- 6 **Optimize data re-use in caches**

### Good to Know

Vector FLOPs are cheap compared to memory access.

If your data is served by RAM and not caches, it does not matter if you have vectorization: you will be bottlenecked by memory access.

# Strip-Mining for Vectorization

# Strip-Mining: Method

- Strip-mining is a programming technique that turns one loop into two nested loops.
- used to expose vectorization opportunities in the inner loop.

Original code:

```
1 for (int i = 0; i < n; i++) {  
2     // ... do work  
3 }
```

Strip-mined implementation:

```
1 const int STRIP=1024;  
2 for (int ii = 0; ii < n; ii += STRIP)  
3     for (int i = ii; i < ii+STRIP; i++) {  
4         // ... do work  
5     }
```

# Example: Binning

# Example: Strip-Mining for Vectorization

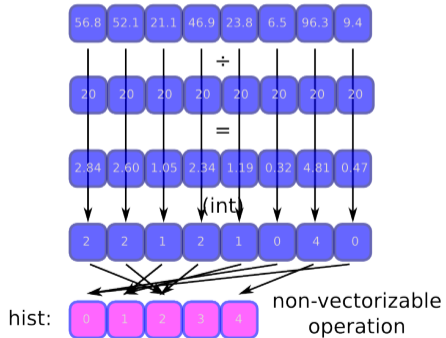
Computing a histogram ( $m \ll n$ ):

```

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

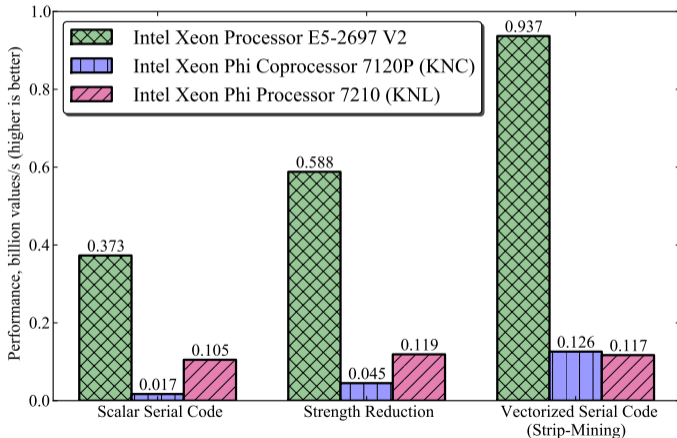


# The Same Calculation, Strip-Mined, Vectorized

```
1 void Histogram(const float* age, int* 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.



# §5. Review and What's Next

# Summary

- 1 Vector-Friendly Data Structures
  - ▶ Use data structures that allow for unit-stride vector load.
- 2 Regularization of Vectorization Pattern
  - ▶ Align data to 64-byte boundaries
  - ▶ Pad data containers and loop bounds
- 3 Remove Run-time Checks
  - ▶ Disable run-time checks for alignment and aliasing with compiler hints
- 4 Strip-Mining for Vectorization
  - ▶ Use strip-mining expose vectorization opportunities.

# Next Session

Next class: optimization of thread parallelism, part I.

- 1 Controlling synchronization in parallel reduction
- 2 Dealing with insufficient parallelism

[Learn More](#)

# HOW Series: Knights Landing



OPTIMIZATION FOR INTEL® XEON PHI™ PROCESSOR  
DEVELOPER'S GUIDE TO KNIGHTS LANDING

Free 2-Hour Webinar  
SEPTEMBER 13, 2016

Register Today >

The graphic features a green background with white and yellow text. On the right side, there is a stylized chess knight piece. The body of the knight is a golden-brown color with a grid pattern, while the head and neck are a solid blue color. A thin blue line extends from the top of the knight's head, suggesting it is hanging from above.

[colfaxresearch.com/how-knl/](http://colfaxresearch.com/how-knl/)

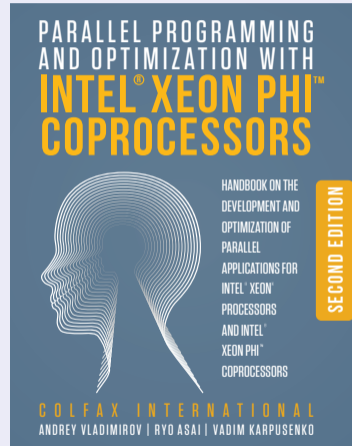
# Textbook

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

## Parallel Programming and Optimization with Intel® Xeon Phi™ Coproprocessors

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

© Colfax International, 2015



<http://xeonphi.com/book>

# Colfax Research

## COLFAX RESEARCH

CONTRIBUTING TO INNOVATIONS IN COMPUTING

[Log In/Out](#) or [Register](#)

---

READ WATCH LEARN CONNECT JOIN



Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

To search, type and hit enter

---

**Popular**

**The Hands-On Tutorials (HOT) webinars:** details an efficient programming for Intel architecture

**The Hands-On Workshop (HOW) Series**

**Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation**

**Parallel Programming Book**

Introduction to parallel programming, deep discussion of optimization techniques, exercises.

© 2015, Colfax International. 508 pages.

**Research and Educational Publications**

**Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation**

**Software Developer's Introduction to the HGST Ultrastar Archive HaaS SMR Drives**

**Optimization Techniques for the Intel MIC Architecture, Part 1 of 3: Strip-Mining for Vectorization**

**Optimization Techniques for the Intel MIC Architecture, Part 2 of 3: Multi-Threading and Parallel Reduction**

**Optimization Techniques for the Intel MIC Architecture, Part 3 of 3: False Sharing and Padding**

**Optimization Techniques for the Intel MIC Architecture, Part 2 of 3: Strip-Mining for Vectorization**

**Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors (and why TX Acceleration May be Enough)**

**Featured Video**

Intel Research National Lab visualization for a floating point

<http://colfaxresearch.com/?p=704>


**Events**

**Presentations**

**Courses**

### Consulting

Share

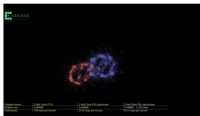


Colfax offers consulting services for enterprises, research help you to:

- Optimize your existing application to take advantage of parallelism, from vectors to cores to clusters and beyond
- Future-proof your application for upcoming innovations
- Accelerate your application using coprocessor technologies
- Investigate the potential system configurations that satisfy your cost, power performance requirements.
- Take a clean-room to develop a novel approach to solve your computing problem

All Video Courses - COP 901 - Chapter 2 - Episode 1.1

**Episode 2.1 - Purpose of the MIC architecture**




Intel Research National Lab visualization for a floating point

<http://colfaxresearch.com/?p=704>

### Software Developer's Introduction to the HGST Ultrastar Archive HaaS SMR Drives


Share



In this paper we will discuss the new HGST Ultrastar Archive HaaS SMR drives. Software Developer's Introduction to the HGST Ultrastar Archive HaaS SMR Drives highlights the key benefits of the new HGST Ultrastar Archive HaaS SMR drives. The paper is available for download at [http://www.hgst.com/ultraarchive](#). The paper is available for download at [http://www.hgst.com/ultraarchive](#).

### Fluid Dynamics with Fortran on Intel Xeon Phi coprocessors

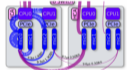
Share



In this book we will discuss the new Intel Xeon Phi coprocessors. The book focuses on the use of Fortran on Intel Xeon Phi coprocessors. The book is available for download at [http://www.intel.com/xeonphi](#). The book is available for download at [http://www.intel.com/xeonphi](#).

### Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors


Share



The Intel Xeon Phi coprocessors are designed to be used in a cluster environment. This paper discusses the configuration and benchmarks of a cluster of Intel Xeon Phi coprocessors. The paper is available for download at [http://www.intel.com/xeonphi](#). The paper is available for download at [http://www.intel.com/xeonphi](#).

### Interview with James Reinders: future of Intel MIC architecture, parallel programming, education


Share



In this video we will discuss the future of Intel MIC architecture, parallel programming, and education. The video is available for download at [http://www.intel.com/xeonphi](#). The video is available for download at [http://www.intel.com/xeonphi](#).

### Parallel Computing in the Search for New Physics at LHC

Share



In this paper we will discuss the use of parallel computing in the search for new physics at the LHC. The paper is available for download at [http://www.intel.com/xeonphi](#). The paper is available for download at [http://www.intel.com/xeonphi](#).

http://colfaxresearch.com/