



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

The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl

July 2016 , Rev. 03a

# 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 – July 25
  - ▶ 02. Programming Intel Xeon Phi (KNC, KNL) – July 26
- HOW to Express Parallelism
  - ▶ 03. Automatic vectorization – July 27
  - ▶ 04. Multi-threading with OpenMP – July 28
- HOW to Get Performance
  - ▶ 05. Comprehensive demo – July 29
  - ▶ 06. Scalar & vectorization tuning – August 1
  - ▶ 07. Multi-threading I – August 2
  - ▶ 08. Multi-threading II – August 3
  - ▶ 09. Memory traffic – August 4
- HOW to Scale
  - ▶ 10. Distributed Computing: MPI – August 5

July 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						
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			
 — 8:00am UTC Lecture+remote access						

# HOW Online

Course page: [colfaxresearch.com/how-16-07](http://colfaxresearch.com/how-16-07)

- 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-16-07](http://colfaxresearch.com/how-16-07)



## 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.

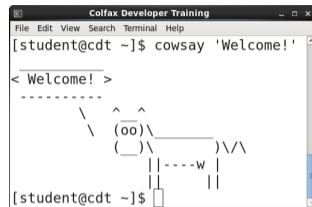
Welcome aboard. Post questions today!

## 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. Refresh

# Performance Optimization

# Computing Platforms

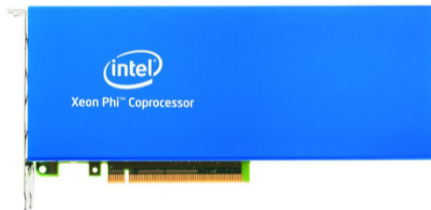
## Intel Xeon Processor



Current: Broadwell  
Upcoming: Skylake

Multi-Core Architecture

## Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

## Intel Xeon Phi Processor, 2nd generation\*

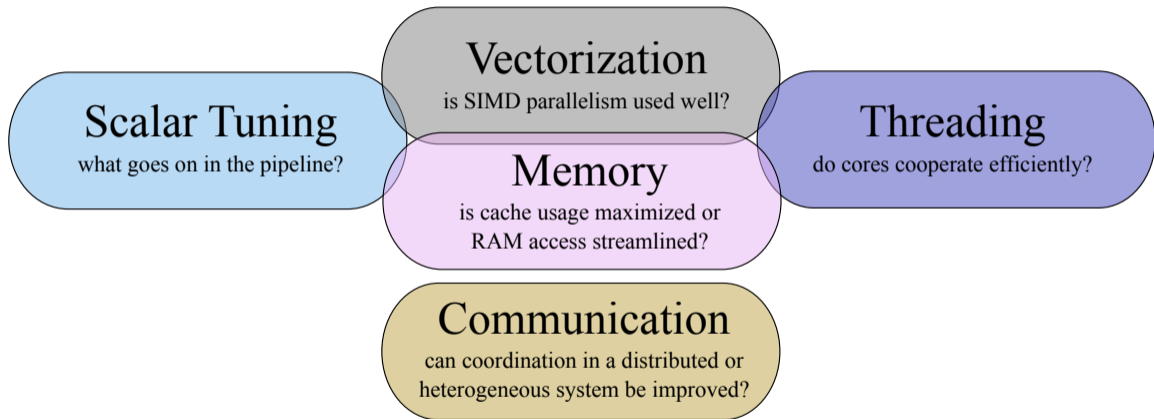


\* socket and coprocessor versions

Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

# Optimization Areas

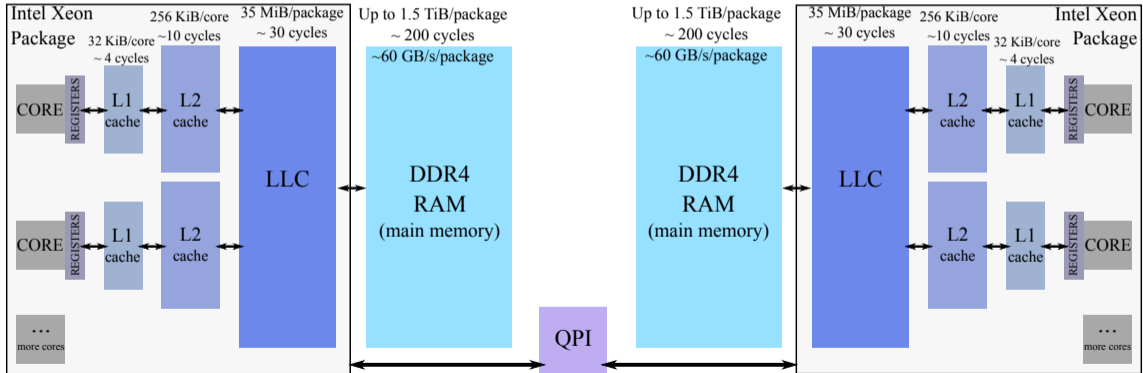


## §3. Memory Traffic Tuning

# Memory Hierarchy

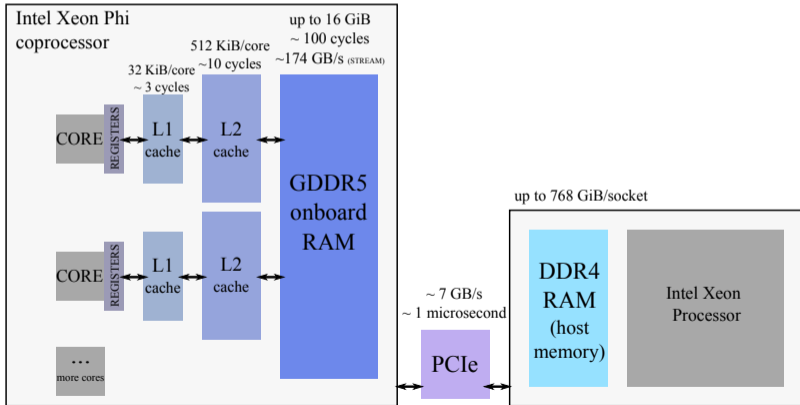
# Intel Xeon CPU: Memory Organization

- Hierarchical cache structure
- Two-way processors have NUMA architecture



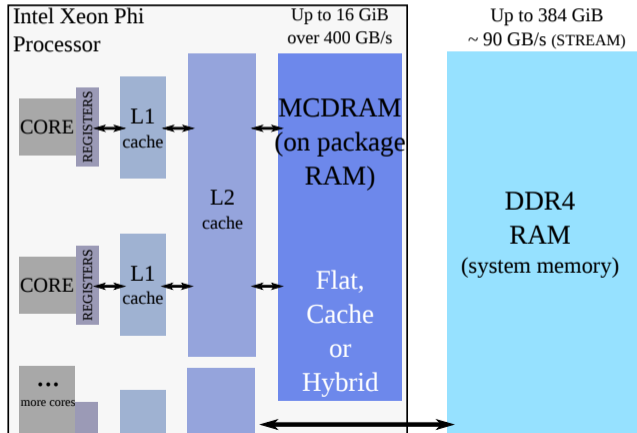
# KNC Memory Organization

- Direct access to  $\leq 16$  GiB of cached GDDR5 memory on board
- No access to system DDR4, connected to host via PCIe



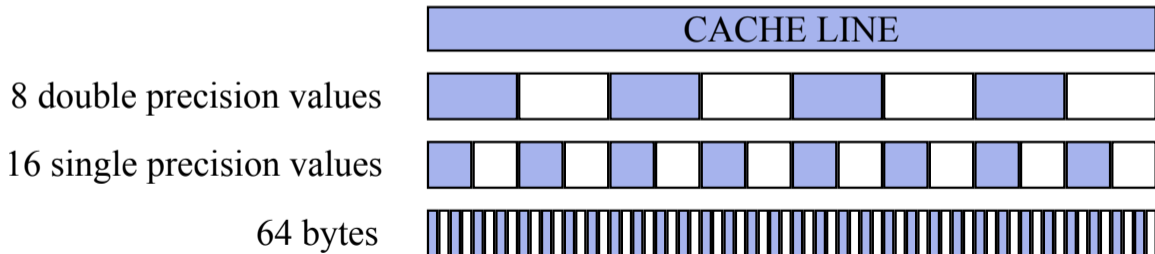
# KNL Memory Organization

- Direct access to on-package MCDRAM *and* system DDR4 (socket)
- Use MCDRAM as cache, in flat mode, or as hybrid



# Cache Lines

- Minimal block of data transferred between memory and cache
- 64 bytes long in Intel Architecture
- Aligned on 64-byte boundaries in memory



# Memory Re-Use and Algorithms

# 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.

# How Cheap are FLOPs?

## Theoretical estimates, Intel Xeon E5-2697 V3 processor

Performance = 28 cores  $\times$  2.7 GHz  $\times$  (256/64) vec.lanes  $\times$  2 FMA  $\times$  2 FPU  $\approx$  1.2 TFLOP/s

Required Data Rate = 1.2 TFLOP/s  $\times$  8 bytes  $\approx$  10 TB/s

Memory Bandwidth =  $\eta \times 2 \times 59.7 \approx$  0.1 TB/s

Ratio = 10/0.1  $\approx$  100 (FLOPs)/(Memory Access)

If the Arithmetic Intensity is...

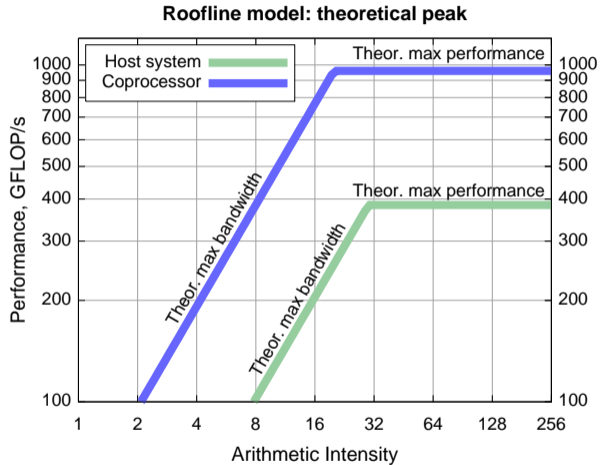
- $> 100$  (FLOPs)/(Memory Access) — Compute Bound Application
- $< 100$  (FLOPs)/(Memory Access) — Bandwidth Bound Application

# On Computational Complexity of Algorithms

Type	Properties	Examples
$O(N)$	Each data element is used a fixed number of times. Memory-bound unless the number of times is large.	Array scaling, image brightness adjustment, vector dot-product.
$O(N^\alpha)$	Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha \gtrsim 2$ . Good implementation can be compute-bound, poor one – memory-bound.	Matrix-matrix multiplication: $O(N^{3/2})$ ( $N$ = amount of data in matrix), direct N-body calculation: $O(N^2)$
$O(N \log N)$	Each element is used $\log N$ times. For small problems – memory-bound, for very large problems transitions to compute-bound	Fast Fourier transform, merge sort
$O(\log N)$	Always memory-bound.	Binary search

$N$  = data size

# Arithmetic Intensity and Roofline Model

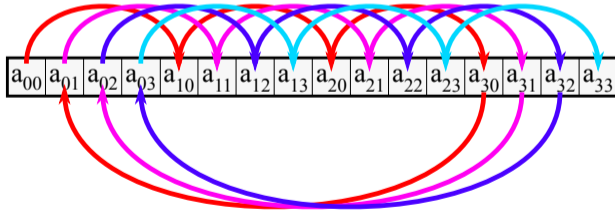
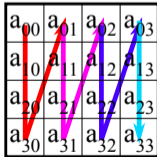
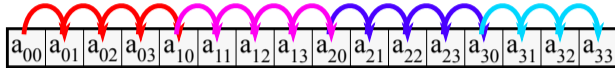
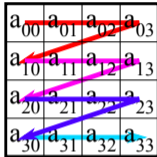


More on roofline model: [Williams et al.](#)

# Loop Permutation

# Principle

Choose loop order to maintain unit-stride memory access



Compiler may or may not be able to automate loop permutation.

# Example: Over-simplified Matrix-Matrix Multiplication

$$C = AB \quad \Leftrightarrow \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$

Before:

```

1  #pragma omp parallel for
2  for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4      #pragma vector aligned
5          for (int k = 0; k < n; k++)
6              C[i*n+j] += A[i*n+k] * B[k*n+j];

```

After:

```

1  #pragma omp parallel for
2  for (int i = 0; i < n; i++)
3      for (int k = 0; k < n; k++)
4      #pragma vector aligned
5          for (int j = 0; j < n; j++)
6              C[i*n+j] += A[i*n+k] * B[k*n+j];

```

# Principle

- The order of nested loops must be chosen for best locality of data access
- At -O2 and above, the compiler automatically interchanges loops in some cases
- In other cases, loop interchange must be investigated manually

# Loop Fusion

# Loop Fusion Technique

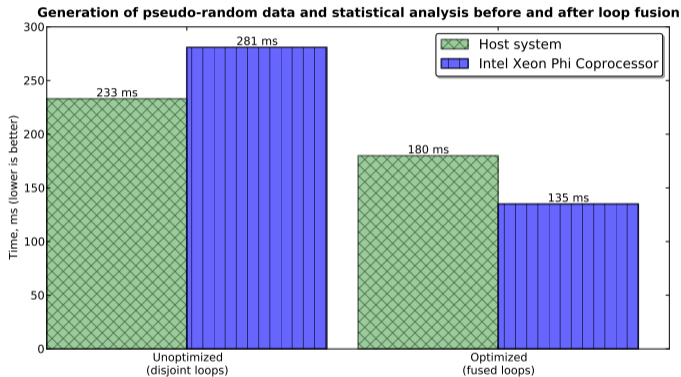
Re-use data in cache by fusing loops in a data processing pipeline

```
1 MyDataType* data = new MyDataType(n);
2
3 for (int i = 0; i < n; i++)
4     Initialize(data[i]);
5
6 for (int i = 0; i < n; i++)
7     Stage1(data[i]);
8
9 for (int i = 0; i < n; i++)
10    Stage2(data[i]);
```

```
1 MyDataType* data = new MyDataType(n);
2
3 for (int i = 0; i < n; i++) {
4
5     Initialize(data[i]);
6
7     Stage1(data[i]);
8
9     Stage2(data[i]);
10 }
```

Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance (see, e.g., [this paper](#)).

# Example Application – Performance



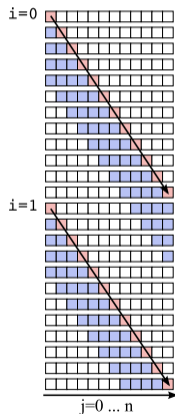
See [labs/4/4.09-memory-loop-fusion-statistics/](https://labs/4/4.09-memory-loop-fusion-statistics/)

# Loop Tiling

# Loop Tiling

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=*b[j];
```



- - cached, LRU eviction policy
- - cache miss (read from memory, slow)
- - cache hit (read from cache, fast)

Cache size: 4

TILE=4

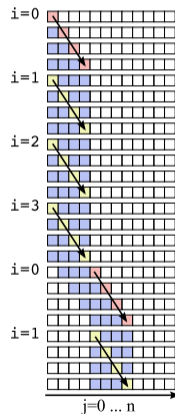
(must be tuned to cache size)

Cache hit rate without tiling: 0%

Cache hit rate with tiling: 75%

Tiled:

```
for (jj=0; jj<n; jj+=TILE)
  for (i=0; i<m; i++)
    for (j=jj; j<jj+TILE; j++)
      ...=*b[j];
```



# Loop Tiling (Cache Blocking) – Procedure

```
1  for (int i = 0; i < m; i++) // Original code:
2      for (int j = 0; j < n; j++)
3          compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine inner loop
2  for (int i = 0; i < m; i++)
3      for (int jj = 0; jj < n; jj += TILE)
4          for (int j = jj; j < jj + TILE; j++)
5              compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute
2  for (int jj = 0; jj < n; jj += TILE)
3      for (int i = 0; i < m; i++)
4          for (int j = jj; j < jj + TILE; j++)
5              compute(a[i], b[j]); // Re-use to j=jj sooner
```

# Loop Tiling (Unroll-and-Jam/Register Blocking)

```

1  for (int i = 0; i < m; i++) // Original code:
2      for (int j = 0; j < n; j++)
3          compute(a[i], b[j]); // Memory access is unit-stride in j

```

```

1  // Step 1: strip-mine outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3      for (int i = ii; i < ii + TILE; i++)
4          for (int j = 0; j < n; j++)
5              compute(a[i], b[j]); // Same order of operation as original

```

```

1  // Step 2: permute and vectorize outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3  #pragma simd
4      for (int j = 0; j < n; j++)
5          for (int i = ii; i < ii + TILE; i++)
6              compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times

```

# Loop Tiling (Unroll-and-Jam) – Alternative Implementation

```
1  for (int i = 0; i < m; i++)    // Original code:
2      for (int j = 0; j < n; j++)
3          compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine both loops
2  for (int ii = 0; ii < m; ii += TILE)
3      for (int i = ii; i < ii + TILE; i++)
4          for (int jj = 0; jj < n; jj += VECLLEN)
5              for (int j = jj; j < jj + VECLLEN; j++)
6                  compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute middle two loops
2  for (int ii = 0; ii < m; ii += TILE)
3      for (int jj = 0; jj < n; jj += VECLLEN)
4          for (int i = ii; i < ii + TILE; i++)
5              for (int j = jj; j < jj + VECLLEN; j++)
6                  compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```

# Cache-Oblivious Recursion

# Principle

Matrix A

$j=0,1,2, \dots, n-1$

0	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
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

$i=0,1,2, \dots, m-1$

$j=0,1,2, \dots, n-1$

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

$i=0,1,2, \dots, m-1$

# Example 1: Matrix Transposition, Tiling



# Matrix Transposition

Before:

```
1  #pragma omp parallel for  
2  for (int i = 0; i < n; i++)  
3      for (int j = 0; j < n; j++)  
4          B[i*n + j] = A[j*n + i];
```

After:

```
1  const int tile = 200;  
2  if (n%tile != 0) exit(1);  
3  
4  #pragma omp parallel for  
5  for (int ii=0; ii<n; ii+=tile)  
6      for (int jj=0; jj<n; jj+=tile)  
7          for (int i=ii; i<ii+tile; i++)  
8              for (int j=jj; j<jj+tile; j++)  
9                  B[i*n + j] = A[j*n + i];
```

## Example 2: Matrix-Vector Multiplication, Tiling

## Example: Matrix-vector Multiplication

$$c_i = \sum_{j=0}^n A_{ij} b_j, \quad i = 0, 1, \dots, (m-1). \quad (1)$$

```

1 void Multiply(const double* const A, const double* const b,
2              double* const c, const long n, const long m){
3     assert(n%64 == 0);
4     #pragma omp parallel for
5     for (long i = 0; i < m; i++)
6     #pragma vector aligned
7         for (long j = 0; j < n; j++) // Each value of A[i*n+j] is used only once
8             c[i] += A[i*n+j] * b[j]; // Each value of b[j] is used a total of m times
9 }

```

Non-optimal performance due to inefficient cache use

# Applying Tiling

```
1  const long jTile = 4096L; assert(n%jTile == 0);
2  #pragma omp parallel
3  {
4      double temp_c[m] __attribute__((aligned(64)));
5      temp_c[:] =0;
6      #pragma omp for
7          for (long jj =0; jj < n; jj+=jTile) // Loop Tiling in j
8              for (long i = 0; i < m; i++)
9                  #pragma vector aligned
10                     for (long j =jj; j < jj+jTile; j++)
11                         temp_c[i] += A[i*n+j] * b[j];
12
13     for(long i = 0; i < m; i++) { // Reduction
14         #pragma omp atomic
15             c[i]+= temp_c[i];
16     } } }
```

# Cache Blocking + Strip-Mine and Collapse

```
1  const long iTile = 64L;   assert(m%iTile == 0);
2  const long jTile = 4096L; assert(n%jTile == 0);
3  #pragma omp parallel
4  {
5      double temp_c[m] __attribute__((aligned(64))); temp_c[:] =0;
6  #pragma omp for collapse(2)
7      for (long ii = 0; ii < m; ii += iTile)
8          for (long jj = 0; jj < n; jj += jTile)
9              for (long i = ii; i < ii+iTile; i++)
10 #pragma vector aligned
11                 for (long j =jj; j < jj+jTile; j++)
12                     temp_c[i] += A[i*n+j] * b[j];
13
14     for(long i = 0; i < m; i++) {
15 #pragma omp atomic
16         c[i] += temp_c[i];
17     } } }
```

## Example 3: Matrix-Vector Multiplication, Recursion

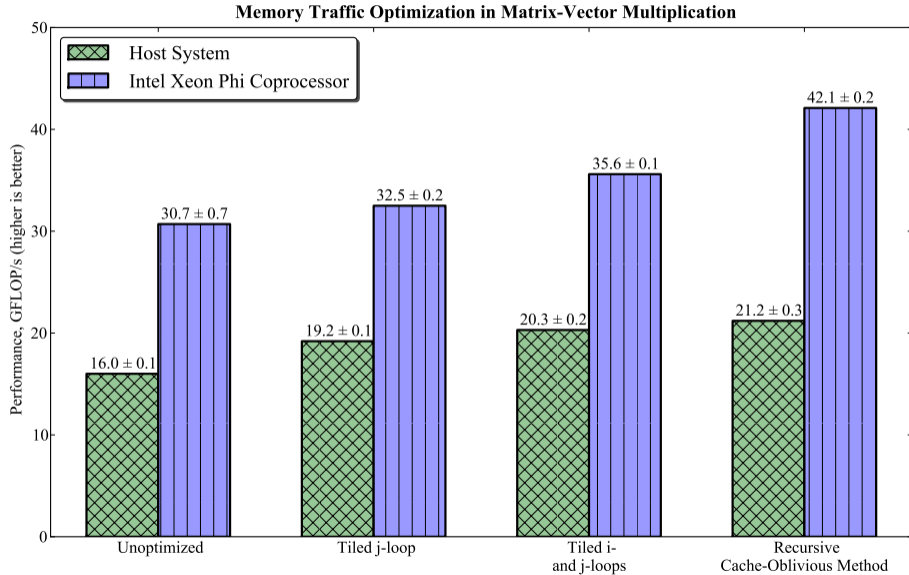
# Example: Matrix-Vector Multiplication

```

1 void RecursMultiply(const double* const A, const double* const b,
2     double* const c, const long n, const long m, const long lda){
3     const long jThreshold = 8192L; assert(n%jThreshold == 0);
4     const long iThreshold = 64L;  assert(m%iThreshold == 0);
5     if ((m<=iThreshold) && (n<=jThreshold)) { // Recursion threshold
6         // .... Base Case: Compute the result inside the tile ... //
7     } else { // Recursive divide-and-conquer
8         if (m*jThreshold > n*iThreshold) { // Split i-wise
9             double c1[m/2] __attribute__((aligned(64)));
10            #pragma omp task
11                { RecursMultiply(&A[0*lda + 0], &b[0], c1, n, m/2, lda); }
12            double c2[m/2] __attribute__((aligned(64)));
13            RecursMultiply(&A[(m/2)*lda + 0], &b[m/2], c2, n, m/2, lda);
14            #pragma omp taskwait
15                c[0:m/2] += c1[0:m/2]; c[m/2:m/2] += c2[0:m/2]; // Reduction
16        } else { // .... Split j-wise .... // }
17    } }

```

# Performance of Matrix Vector Multiplication

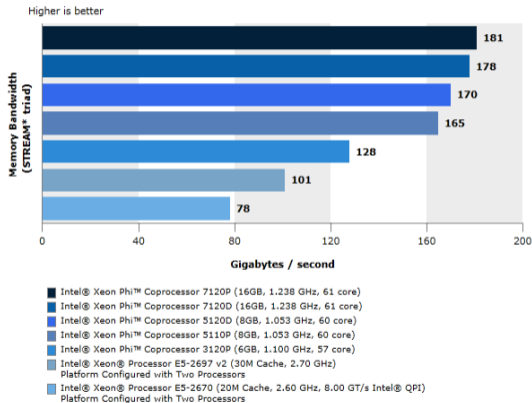


# Bandwidth Tuning

# STREAM Benchmark

- Industry-standard tool for memory bandwidth measurement
- 4 tests: COPY, ADD, SCALE and TRIAD
- Download from Dr. John McCalpin's site:  
[www.cs.virginia.edu/stream](http://www.cs.virginia.edu/stream)

Expected for KNL:  $\approx 400$  GB/s



# STREAM Benchmark on Tuning

- Set 1 thread per core on all platforms (-1 for offload)
- Set affinity “scatter”: [white paper](#) (Colfax Research)
- Tune prefetching: [white paper](#) (Intel)

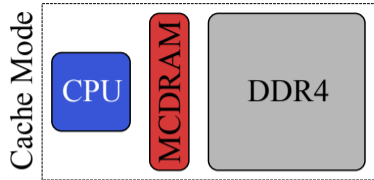
In addition, secret sauce for your own STREAM-like application:

- Parallel first touch (see NUMA discussion in Session 08)
- Essential element – streaming stores: [discussion](#)

# Using High-Bandwidth Memory (MCDRAM) in KNL

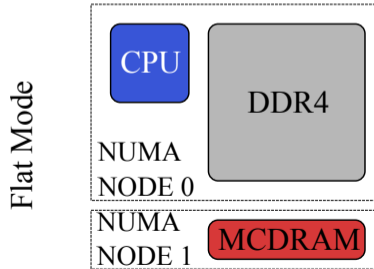
## Option 1 : cache/hybrid mode

- Treat it as LLC
- Data locality techniques
- Miss latency 2x the direct DDR4 access



## Option 2 : flat mode

- Application fits in 16 GiB? `numactl`
- More than 16 GiB data? Use special allocators (e.g., `memkind`)



# Binding to NUMA Nodes with numactl

- numactl – a Linux tool for controlling NUMA policy for processes

```
vega@lyra% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 65457 MB
node 0 free: 24426 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 65536 MB
node 1 free: 53725 MB
node distances:
node  0  1
  0:  10  21
  1:  21  10
vega@lyra% numactl --membind=<nodes> --cpunodebind=<nodes> ./myApplication
```

# HBM in Knights Landing

- Finding HBM (MCDRAM) in an Intel Xeon Phi processor x200 (KNL):

```
user@knl% # In Flat mode with All-to-All or Quadrant
user@knl% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ... 249 250 251 252 253 254 255
node 0 size: 98207 MB
node 0 free: 94798 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15991 MB
```

- Binding the application to HBM (Flat/Hybrid)

```
user@knl% numactl --membind 1 ./runme
// ... Application running in HBM ... //
```

## §4. Review and What's Next

# Summary

This session:

- ① Memory traffic optimization: spatial and temporal data locality
- ② Loop tiling: cache blocking and unroll-and-jam
- ③ Cache-oblivious recursion
- ④ First-touch allocation in NUMA systems
- ⑤ Loop fusion

Next session: distributed computation with MPI, specifics of message passing with coprocessors.

[Learn More](#)

# HOW Series: Knights Landing

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

Free 2-Hour Webinar  
AUGUST 11, 18

Register Today >



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

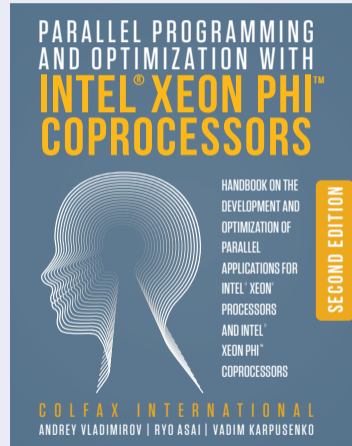
# Textbook

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: Multi-Threading and Parallel Reduction**

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

**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)**


**Events** **Discussions**

---

**Conferences**

### Consulting

Share

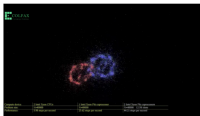


Colfax offers consulting services for enterprises, research help you to:

- 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 MIC architecture is a novel approach to solve your computing problem. It is designed to be used in a variety of applications, from data centers to edge computing. It is designed to be used in a variety of applications, from data centers to edge computing. It is designed to be used in a variety of applications, from data centers to edge computing.

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

In this webinar, a Colfax Research webinar series, we will discuss the use of Fortran on Intel Xeon Phi coprocessors. We will discuss the use of Fortran on Intel Xeon Phi coprocessors. We will discuss the use of Fortran on Intel Xeon Phi coprocessors.




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



This paper will discuss the use of Intel Xeon Phi coprocessors in a cluster. We will discuss the use of Intel Xeon Phi coprocessors in a cluster. We will discuss the use of Intel Xeon Phi coprocessors in a cluster.


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

A new series of an interview with James Reinders, the Intel and Intel Advisor lead programmer, will discuss the future of parallel programming and Intel MIC architecture. We will discuss the future of parallel programming and Intel MIC architecture. We will discuss the future of parallel programming and Intel MIC architecture.



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

This paper will discuss the use of Intel Xeon Phi coprocessors in the search for new physics at the LHC. We will discuss the use of Intel Xeon Phi coprocessors in the search for new physics at the LHC. We will discuss the use of Intel Xeon Phi coprocessors in the search for new physics at the LHC.



<http://colfaxresearch.com/>