COLFAX
Customized Solutions

PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

The Hands-On Workshop (HOW) Series

Session 6

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)

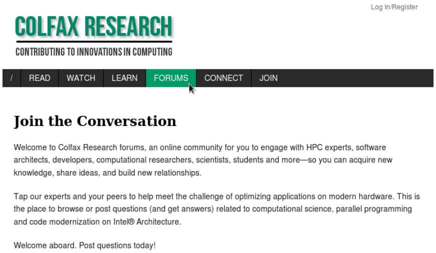
Additional resources:

- ▷ 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

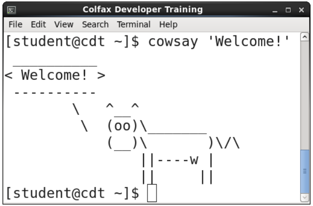


Email (organizational):

`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

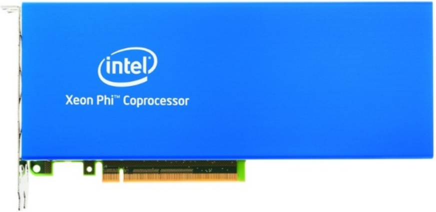
§2. PERFORMANCE OPTIMIZATION

COMPUTING PLATFORMS

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

* socket and coprocessor versions

Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

OPTIMIZATION AREAS

Vectorization
is SIMD parallelism used well?

Scalar Tuning
what goes on in the pipeline?

Threading
do cores cooperate efficiently?

Memory
is cache usage maximized or RAM access streamlined?

Communication
can coordination in a distributed or heterogeneous system be improved?

§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:

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

FLOATING-POINT SEMANTICS

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

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

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, infinites: 4, denormals[1]: 8, zeroes: 16.

[1] by default, on Intel Xeon Phi, denormals are flushed to zero in hardware, but supported in SVML

PROGRAMMING PRACTICES

STRENGTH REDUCTION

Common Subexpression Elimination.

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

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

Replace division with multiplication.

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

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

Use functions with Hardware support.

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

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

CONSISTENCY OF PRECISION: CONSTANTS

```
// Bad: 2 is "int"
long b=a*2;

// Bad: overflow
long n=100000*100000;

// Bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q=2*p;

// Bad: 1e9 is "double"
float r=1e9*p;

// Bad: 1 is "int"
double t=s+1;
```

```
// Good: 2L is "long"
long b=a*2L;

// Good: correct
long n=100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2.0f is "float"
float q=2.0f*p;

// Good: 1e9f is "float"
float r=1e9f*p;

// Good: 1.0 is "double"
double t=s+1.0;
```

CONSISTENCY OF PRECISION: FUNCTIONS

```
// Bad: 3.14 is a double
float x = 3.14;

// Bad: sin() is a
// double precision function
float s = sin(x);

// Bad: round() takes double
// and returns double
long v = round(x);

// Bad: abs() is not from IML
// it takes int and returns int
int v = abs(x);
```

```
// Good: 3.14f is a float
float x = 3.14f;

// Good: sin() is a
// single precision function
float s = sinf(x);

// Good: lroundf() takes float
// and returns long
long v = lroundf(x);

// Good: fabsf() is from IML
// It takes and returns a float
float v = fabsf(x);
```

CONSISTENCY OF PRECISION: FUNCTIONS

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

```
vega@lyra% ./Scalar-TestFOverload
Proof that exp() is not overloaded:
exp (1.0f)=2.718281828459045
exp (1.0 )=2.718281828459045
Exact:   e=2.7182818284590452353...

Proof that expf() gives lower precision:
expf(1.0f)=2.718281745910645
expf(1.0 )=2.718281745910645
Exact:   e=2.7182818284590452353...

Overloading in namespace std:
std::exp(1.0f)=2.718281745910645
std::exp(1.0 )=2.718281828459045
Exact:       e=2.7182818284590452353...
```

MOVE BRANCHES OUTSIDE OF LOOPS

```
// Elegant, but bad for performance
for (i = 0; i < n; i++) {
  if (i == 0) {
    // Absorbing boundary
    B[i] = 0.0;
  } else if (i == n - 1) {
    // Injection at boundary
    B[i] = B[i] + 1.0;
  } else {
    // Diffusion between boundaries
    B[i] = 0.25*(A[i-1] +
                 2.0*A[i] + A[i+1]);
  }
}
```

```
// Moving branches out of loops


// Absorbing boundary
B[i] = 0.0;

for (i = 1; i < n - 1; i++) {
  // Diffusion between boundaries
  B[i] = 0.25*(A[i-1] + 2.0*A[i] +
                          A[i+1]);
}

// Injection at boundary
B[n-1] = B[n-1] + 1.0;
```

REDUNDANT CODE IS OK

```
// Elegant, but bad for performance
for (ii = 0; ii < n; ii+=16) {
  for (i = ii; i < ii+16; i++)
    // Branch causes unnecessary
    // masking of vector iterations
    if (i < n) {
      A[k*n + i] = ...
    }
}
```

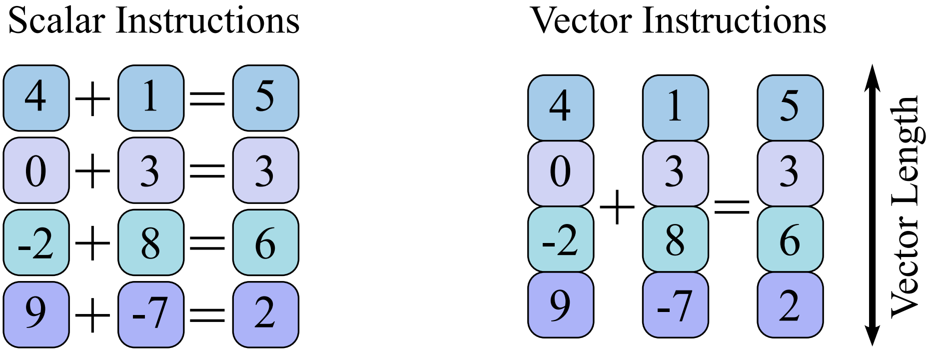
```
// Redundant code, but faster
const int nTrunc = n - 16;
for (ii = 0; ii < nTrunc; ii+=16) {
  for (i = ii; i < ii+16; i++)
      A[k*n + i] = ...

for (i = nTrunc; i < n; i++)
  A[k*n + i] = ...
}
```

§4. VECTORIZATION

SHORT VECTOR SUPPORT

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



DATA STRUCTURES AND MEMORY ACCESS

UNIT-STRIDE ACCESS

Unit-stride access is optimal:

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

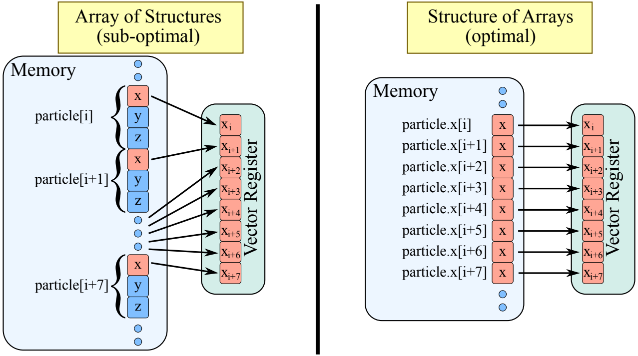
Non-unit stride is slower:

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

Stochastic access may be vectorized (but not efficient):

```
for (int i = 0; i < n; i++)
  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)`.

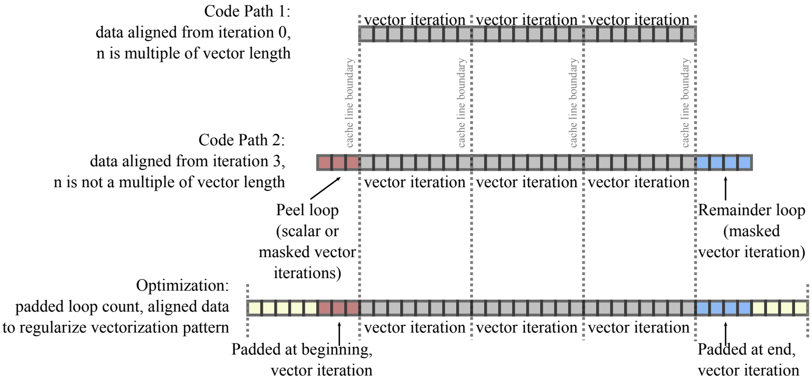
Processor	**Operation**	**Alignment**
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load,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

Why align: speed up vector load/stores, avoid false sharing (see Session 7), accelerate RDMA.

WHAT HAPPENS WITHOUT ALIGNMENT

Compiler may implement peel and remainder loops:

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



CREATING ALIGNED DATA CONTAINERS

- Data alignment on the stack

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

- Data alignment on the heap

```
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 MULTI-DIMENSIONAL CONTAINERS 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.

Incorrect:

```
// A - matrix of size (n x n)
// n is not a multiple of 16
float* A =
 _mm_malloc(sizeof(float)*n*n, 64);

for (int i = 0; i < n; i++)
  // A[i*n + 0] may be unaligned
  for (int j = 0; j < n; j++)
    A[i*n + j] = ...
```

Correct:

```
// ... Padding inner dimension
int lda=n + (16-n%16); // lda%16==0
float* A =
 _mm_malloc(sizeof(float)*n*lda, 64);

for (int i = 0; i < n; i++)
  // A[i*lda + 0] aligned for any i
  for (int j = 0; j < n; j++)
    A[i*lda + j] = ...
```

DATA ALIGNMENT HINTS

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

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

```
void LU_decomp(const int n, float* const A) {
  // LU decomposition (Doolittle algorithm)
  // In-place decomposition of form A=LU
  // L is returned below main diagonal of A
  // U is returned at and above main diagonal
  for (int b = 0; b < n; b++) {
    // Strength reduction:
    const float recAbb = 1.0f/A[b*n + b];
    for (int i = b+1; i < n; i++) {
      A[i*n + b] = A[i*n + b]*recAbb;
#pragma simd
      for (int j = b+1; j < n; j++)
        A[i*n + j] -= A[i*n + b]*A[b*n + j];
    }
  }
}
```

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

Based on publication: http://xeonphi.com/papers/l

Non-optimal Vectorization Pattern.

- Unaligned
- Irregular loop count

LU DECOMPOSITION: REGULARIZING VECTORIZATION

Before:

```
for (int b = 0; b < n; b++) {
  // ...
  // ...
  for (int i = b+1; i < n; i++) {
    // ...
    for (int j = b+1; j < n; j++)
      A[i*n+j] -= A[i*n+b]*A[b*n+j];
  }
}
```

After:

```
for (int b = 0; b < n; b++) {
  // ...
  const int jMin = (b+1) - (b+1)%16;
  for (int i = b+1; i < n; i++) {
    // ...
    for (int j = jMin; j < n; j++)
      A[i*n+j] -= L[i*n+b]*A[b*n+j];
  }
}
```

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:

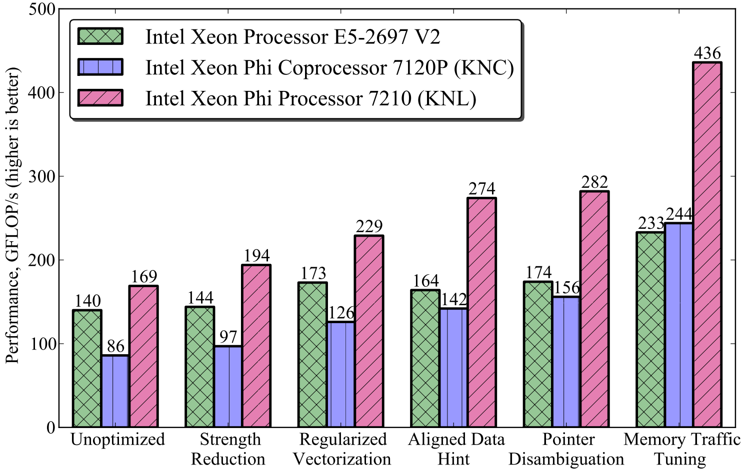
```
for (int b = 0; b < n; b++) {
  const int jMin = (b+1)-(b+1)%tile;
  const float recAbb = 1.0f/A[b*n+b];
  for (int i = b+1; i < n; i++) {
    L[i*n + b] = A[i*n + b]*recAbb;


#pragma simd
    for (int j = jMin; j < n; j++)
      A[i*n+j] -= L[i*n+b]*A[b*n+j];
  }
}
```

After:

```
for (int b = 0; b < n; b++) {
  const int jMin = (b+1)-(b+1)%tile;
  const float recAbb = 1.0f/A[b*n+b];
  for (int i = b+1; i < n; i++) {
    L[i*n + b] = A[i*n + b]*recAbb;
#pragma vector aligned
#pragma ivdep
#pragma simd
    for (int j = jMin; j < n; j++)
      A[i*n+j] -= L[i*n+b]*A[b*n+j];
  }
}
```

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:

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

Strip-mined implementation:

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

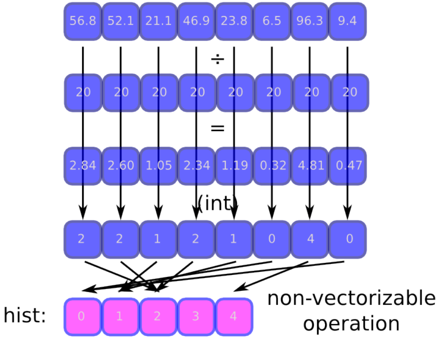
EXAMPLE: BINNING

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

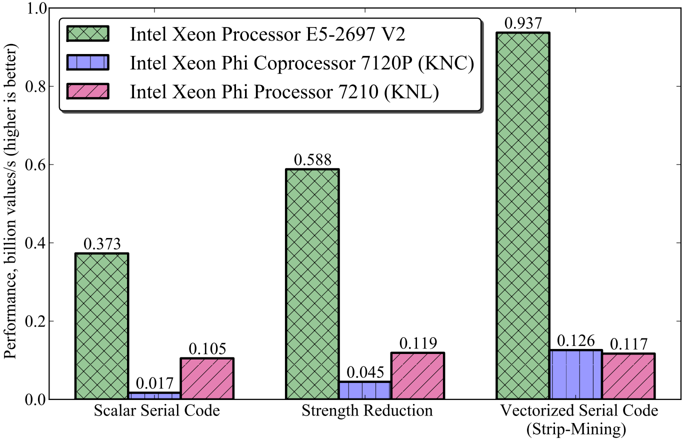


THE SAME CALCULATION, STRIP-MINED, VECTORIZED

```
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.



§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

colfaxresearch.com/how-knl/

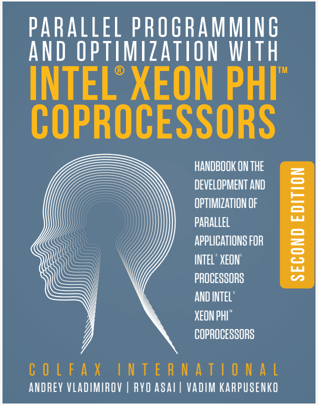
LEARN MORE

TEXTBOOK

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

Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

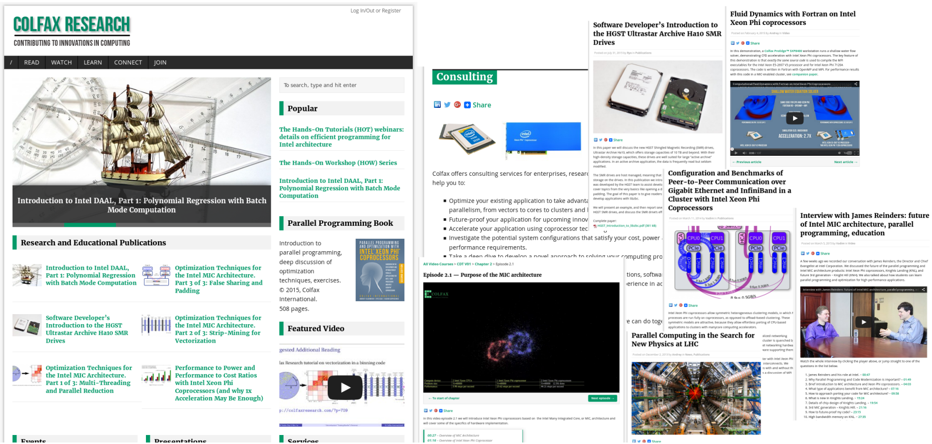
Handbook on the Development and Optimization of Parallel Applications for Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors



http://xeonphi.com/book

LEARN MORE

COLFAX RESEARCH



http://colfaxresearch.com/

LEARN MORE