PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

Hands-On Workshop (HOW) Series "Deep Dive"

Session 7

Colfax International — *colfaxresearch.com*

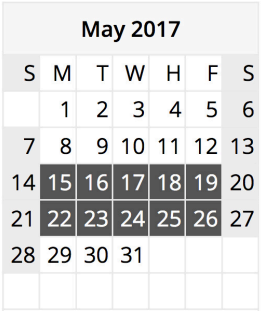
May 2017

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

- Module I. Programming
  - 01. Intel Architecture and Modern Code – May 15
  - 02. Xeon Phi, Coprocessors, Omni-Path – May 16
- Module II. Expressing Parallelism
  - 03. Automatic vectorization – May 17
  - 04. Multi-threading with OpenMP – May 18
  - 05. Distributed Computing, MPI – May 19
- Module III. Optimization
  - 06. Optimization Overview: N-body – May 22
  - 07. Scalar tuning, Vectorization – May 23
  - 08. Common Multi-threading Problems – May 24
  - 09. Multi-threading, Memory Aspect – May 25
  - 10. Access to Caches and Memory – May 26



HOW SERIES ONLINE

Course page:
colfaxresearch.com/how-17-05

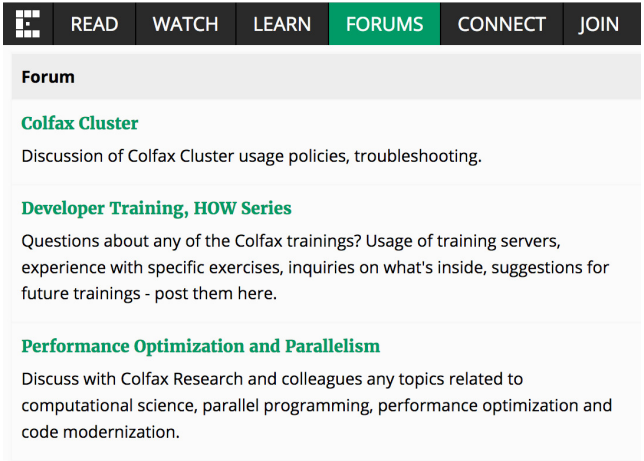
- Slides
- Code
- Video
- Chat

More workshops:
colfaxresearch.com/training

GET YOUR QUESTIONS ANSWERED: CHAT

colfaxresearch.com/how-17-05

GET YOUR QUESTIONS ANSWERED: FORUMS



colfaxresearch.com/discussion

HANDS-ON EXERCISES AND REMOTE ACCESS

- All registrants receive an invitation from cluster@colfaxresearch.com
- Queue-based access to Intel Xeon E5, Intel Xeon Phi (KNC and KNL)
- Can access the cluster the entire 2 weeks of the workshop

§2. PERFORMANCE OPTIMIZATION

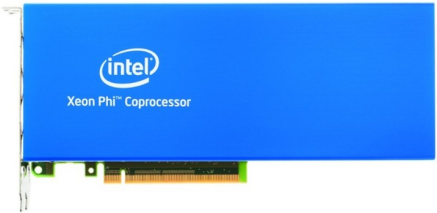
COMPUTING PLATFORMS

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

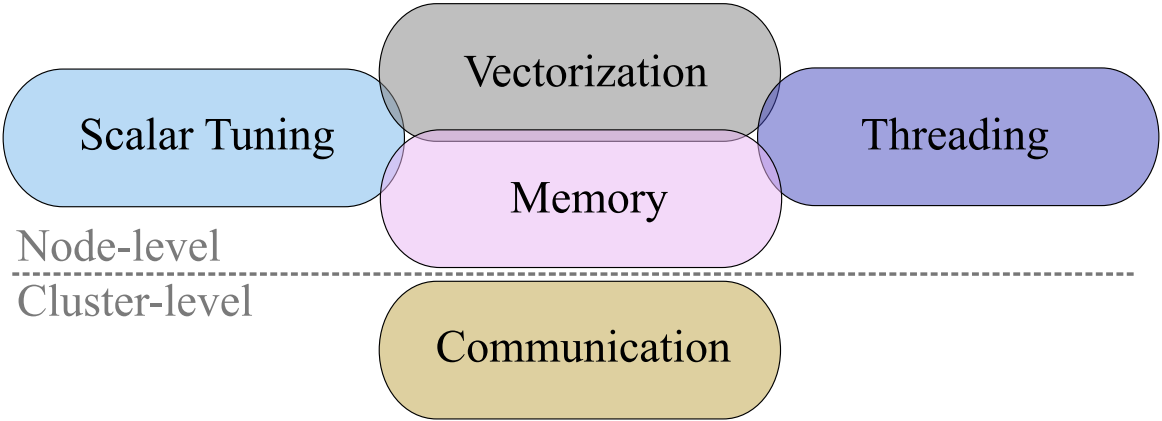
Intel Many Integrated Core (MIC) Architecture

Intel Xeon Phi Processor, 2nd generation*

* socket and coprocessor versions

Knights Landing (KNL)

OPTIMIZATION AREAS



§3. SCALAR TUNING

COMPILER ARGUMENTS

PERFORMANCE-RELATED ARGUMENTS FOR INTEL COMPILERS

`-O<n>` – optimization level (`n`=0, 1, 2, 3 or s)

`-g` – debugging symbols (resets optimization to `-O0`)

`-fp-model <model>` – floating-point semantics (`model=strict`, `precise`, `fast=1` or `fast=2`)

`-fimf-precision=<value>` – transcendental function precision (`value=high`, `medium` or `low`)

`-x<code>` – target arch. (`code=MIC-AVX512`, `CORE-AVX2`, etc., or `host`)

`-ax<code>` – dispatch for multiple target architectures

`-ipo` – inter-procedural optimization

Click above for links in the Intel C++ Compiler User and Reference Guide

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() {
  //...
}
```

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));
```

PERFORMANCE OF VECTOR INSTRUCTIONS IN KNL

All values in cycles. Lower is better.

Instruction	Latency	1/Throughput
Most vector math and FMA	6	0.5
64-bit exp2a23, rcp28 and rsqrt28	7	2
32-bit exp2a23, rcp28 and rsqrt28	8	3
Floating-point division and sqrt	38	10
Simple integer math	2	2
32-bit scalar division	25	20
64-bit scalar division	40	30
Type conversion (same width)	2	1
Type conversion (different widths)	6	5

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

```
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-TestFOverload
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

```
// 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] = A[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] = A[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 - 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

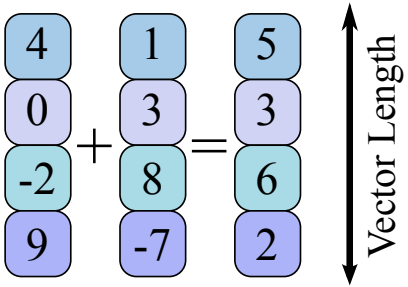
SHORT VECTOR SUPPORT

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

Scalar Instructions

$$4 + 1 = 5$$
$$0 + 3 = 3$$
$$-2 + 8 = 6$$
$$9 + -7 = 2$$

Vector Instructions



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];
```

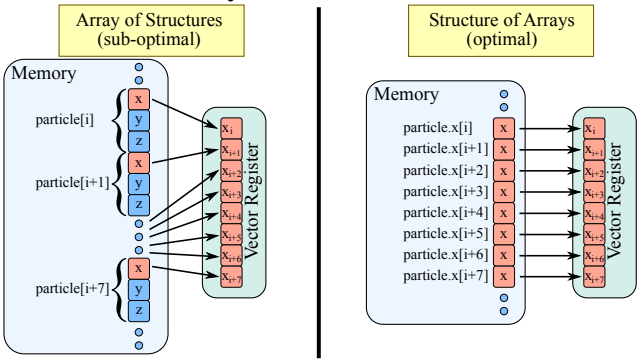
Non-unit stride is slower:

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

Stochastic access may be vectorized (but not efficient):

```
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)`.

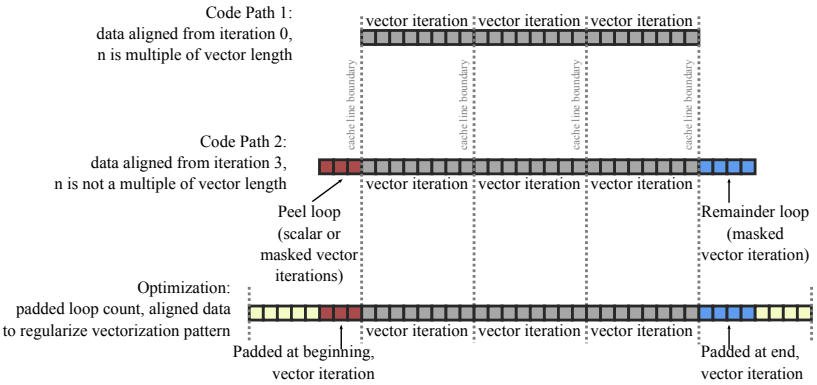
Processor	Operation	Alignment
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load, store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

Why align: speed up vector load/stores, avoid false sharing, 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

```
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 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:

```
1 // A - matrix of size (n x n)
2 // n is not a multiple of 16
3 float* A =
4  _mm_malloc(sizeof(float)*n*n, 64);
5
6 for (int i = 0; i < n; i++)
7   // A[i*n + 0] may be unaligned
8   for (int j = 0; j < n; j++)
9     A[i*n + j] = ...
```

Correct:

```
1 // ... Padding inner dimension
2 int lda=n + (16-n%16); // lda%16==0
3 float* A =
4  _mm_malloc(sizeof(float)*n*lda, 64);
5
6 for (int i = 0; i < n; i++)
7   // A[i*lda + 0] aligned for any i
8   for (int j = 0; j < n; j++)
9     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:

```
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`

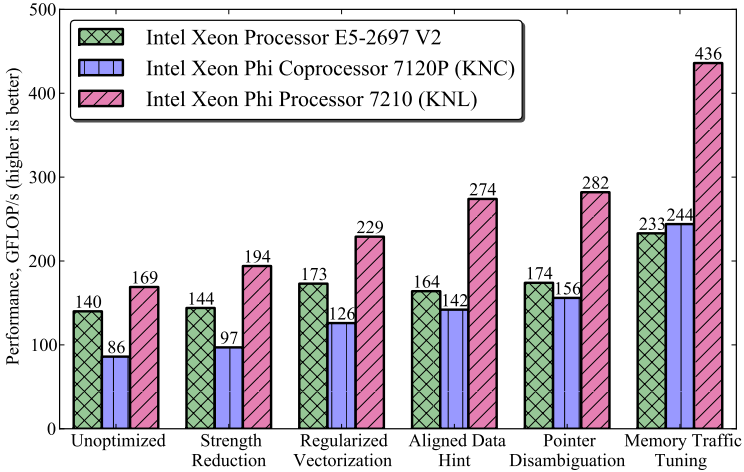
Before:

```
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

STRIP-MINING FOR VECTORIZATION

STRIP-MINING FOR VECTORIZATION

- Programming technique that turns one loop into two nested loops.
- Used to expose vectorization opportunities.

Original:

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

Strip-mined:

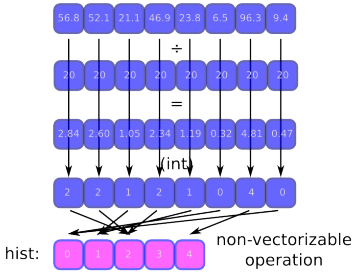
```
1 const int STRIP=1024;
2 const int nPrime = n - n%STRIP;
3 for (int ii=0; ii<nPrime; ii+=STRIP)
4   for (int i=ii; i<ii+STRIP; i++)
5     // ... do work
6
7 for (int i=nPrime; i<n; i++)
8   // ... do work
```

EXAMPLE: BINNING

EXAMPLE: BINNING PROBLEM

```
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   const float grpWidth) {
11     for (int i = 0; i < n; i++) {
12       const int j = int(age[i]/grpWidth);
13       hist[j]++;
14     }
15 }
```

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



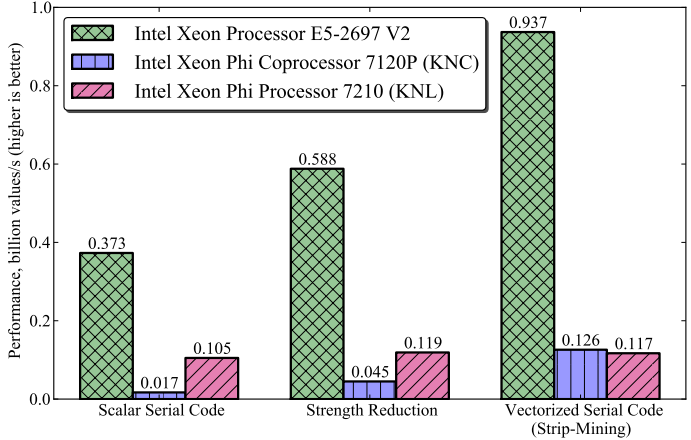
THE SAME CALCULATION, STRIP-MINED, VECTORIZED

```
1  const float recGrpWidth = 1.0f/grpWidth; // precompute the reciprocal
2
3  for (int ii = 0; ii < n; ii += 16) { // strip-mining
4
5    int index[16]; // a block of indices
6    for (int i = ii; i < ii + 16; i++) // vectorizable
7      index[i-ii] = (int) ( age[i] * recGrpWidth ); // unit-stride access
8
9    for (int c = 0; c < 16; c++) // not vectorizable
10     hist[index[c]]++; // indirect access
11 }
```

STRIP-MINING FOR VECTORIZATION

Vectorization improves performance.

More work is needed to take advantage of multiple cores.



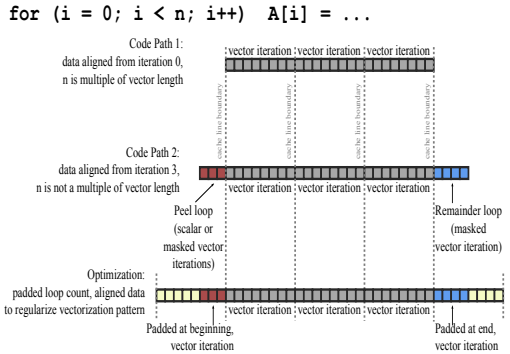
§5. REVIEW AND WHAT'S NEXT

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.

LOOP WAS VECTORIZED, NOW WHAT?

1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**



LOOP WAS VECTORIZED, NOW WHAT?

1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**

Vector Arithmetics is Cheap, Memory Access is Expensive

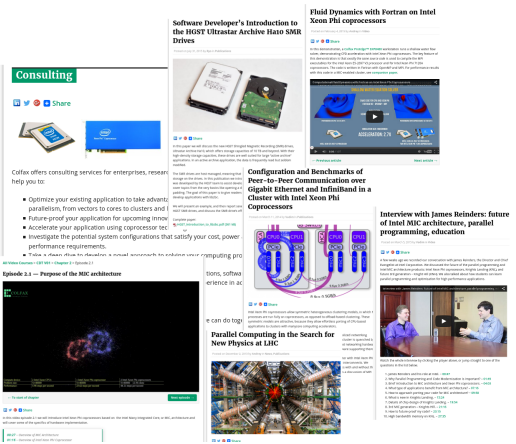
If you don't optimize cache usage, vectorization will not matter.
You will be bottlenecked by memory access.

NEXT SESSION

Next class: optimization of thread parallelism, common issues.

1. Controlling synchronization in parallel reduction
2. Eliminating false sharing
3. Dealing with insufficient parallelism

COLFAX RESEARCH



http://colfaxresearch.com/