Programming and Optimization for Intel® Architecture

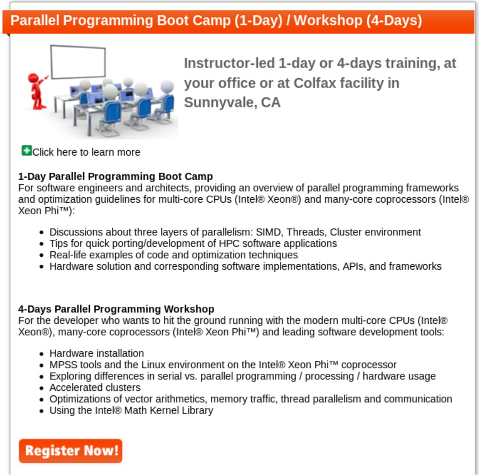
The Hands-On Workshop (HOW) Series

Colfax International — @colfaxintl

April 2016 , Rev. 02b

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.



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

1. Why Intel Parallel Architectures?
   - Parallelism and specialization – April 18
   - Programming model continuity – April 18
2. Programming models for Xeon Phi coprocessors
   - Native programming – April 18
   - Offload programming – April 19
3. Expressing Parallelism
   - Introduction to vectorization – April 20
   - Crash-course on OpenMP – April 21
4. Optimization – intro on April 22
   - Vectorization tuning – April 25
   - Multi-threading – April 26, 27
   - Memory traffic – April 28
5. Distributed Computing: MPI – April 29

April 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

— Lecture+remote access

May 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				

— Self-study/remote access

HOW Online

Course page: colfaxresearch.com/how-16-04

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

Additional resources:

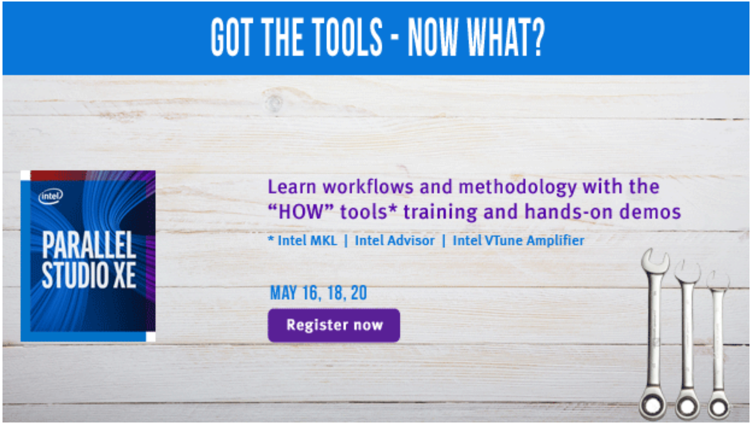
- More workshops like this one: colfaxresearch.com/how-series
- Video courses: colfaxresearch.com/video-courses
- Intel Many Integrated Core Architecture Forum

HOW Series "Deep Dive" in May

Wish you had joined us earlier? Want to recomment the HOW series to a friend? Another HOW Series run coming up in May.

colfaxresearch.com/how-series/

HOW Series “Tools”



Registration link coming soon.

Developer’s Guide to Knights Landing

colfaxresearch.com/knl-webinar/

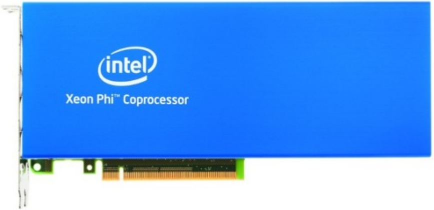
§2. Performance Optimization

Computing Platforms

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Corocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

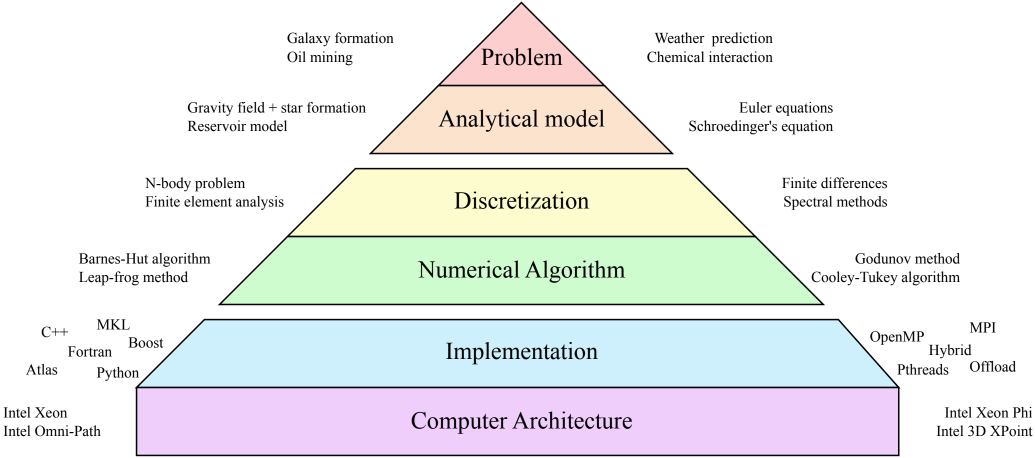
* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Computing in Science and Engineering



Optimization Areas

1. **Scalar optimization** (compiler-friendly practices)
2. **Vectorization** (must use 16- or 8-wide vectors)
3. **Multi-threading** (must scale to 100+ threads)
4. **Memory access** (streaming access or tiling)
5. **Communication** (offload, MPI traffic control)

§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% icc -o mycode -O3 source.c
```

For a specific function:

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

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

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.

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

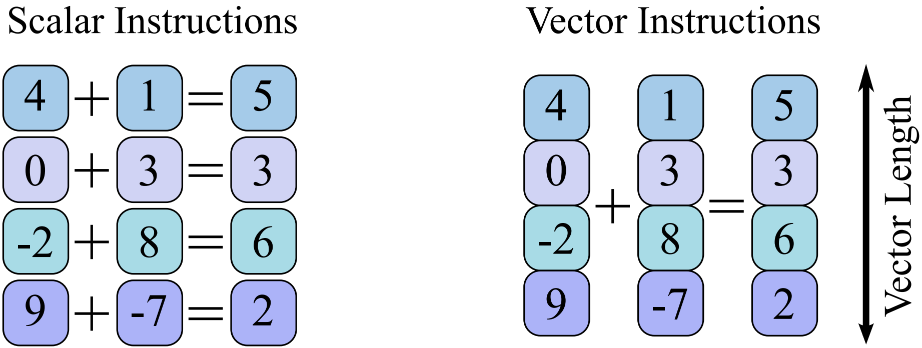
Transcendental functions are *not* overloaded (unless in namespace `std` in C++).

```
vega@lyra% ./Scalar-TestFOverload
Proof that exp() is not overloaded:
exp (1.0f)=2.718281828459045l
exp (1.0 )=2.718281828459045l
Exact:   e=2.7182818284590452353 6...

Proof that expf() gives lower precision:
expf(1.0f)=2.718281745910644 5
expf(1.0 )=2.718281745910644 5
Exact:   e=2.7182818284590452353 6...

Overloading in namespace std:
std::exp(1.0f)=2.718281745910644 5
std::exp(1.0 )=2.718281828459045l
Exact:       e=2.7182818284590452353 6...
```

§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];
```

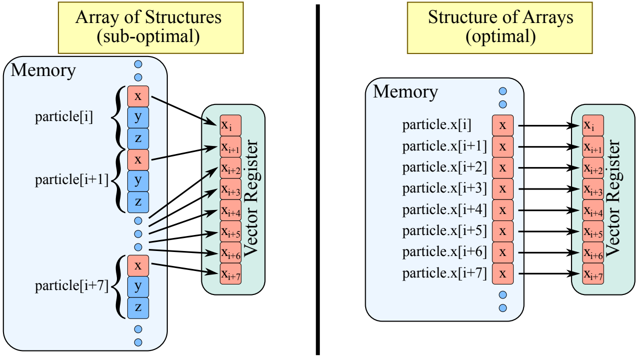
Non-unit stride is slower:

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

Stochastic access cannot be vectorized:

```
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)

Data Alignment

- 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`

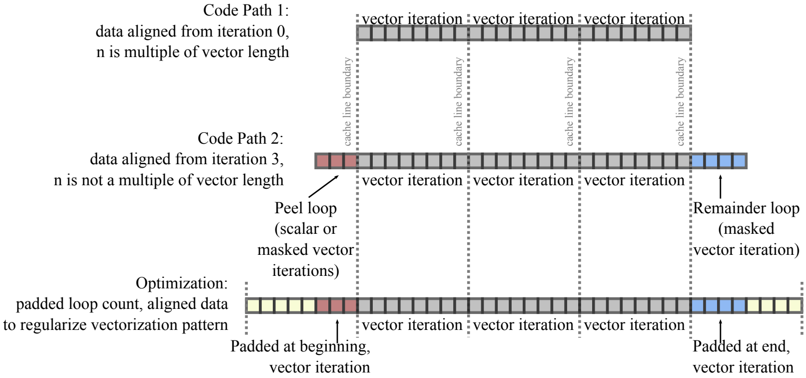
Padding for Alignment

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.

```
// ... Padding inner dimension so that every row is aligned
int lda=n;
if (n % 16 != 0) lda += (16 - n%16);  // now lda%16==0
float* A = _mm_malloc(sizeof(float)*n*lda, 64);

// If A[0         ] is aligned AND lda%16==0, then
//    A[i*lda + 0] is aligned for any i
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    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:

```
// 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/

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`

Pointer-Disambiguation: Multiversioning

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat code.optrpt
...
LOOP BEGIN at code.cc(4,1)
<Multiversioned v1>
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
LOOP BEGIN at code.cc(4,1)
<Multiversioned v2>
   remark #15304: loop was not vectorized: non-vectorizable loop instance ....
LOOP END
```

Aliasing (true vector dependence) checked at *runtime* to choose the implementation.

Pointer Disambiguation to Prevent Multiversioning

Prevent multiversioning by using `#pragma ivdep`

```
#pragma ivdep
  for (int i = 0; i < n; i++)
      // ...
```

```
user@host% icpc -c code.cc -qopt-report -qopt-report-phase:vec
user@host% cat vdep.optrpt
...
LOOP BEGIN at code.cc(4,1)
   remark #25228: LOOP WAS VECTORIZED
LOOP END
...
```

When keyword `restrict` is used instead, may not disambiguate different offsets of same pointer (e.g, `A[i*n+j] += A[b*n+j]`).

LU Decomposition: Compiler hints

- Data alignment hint: `#pragma vector aligned`

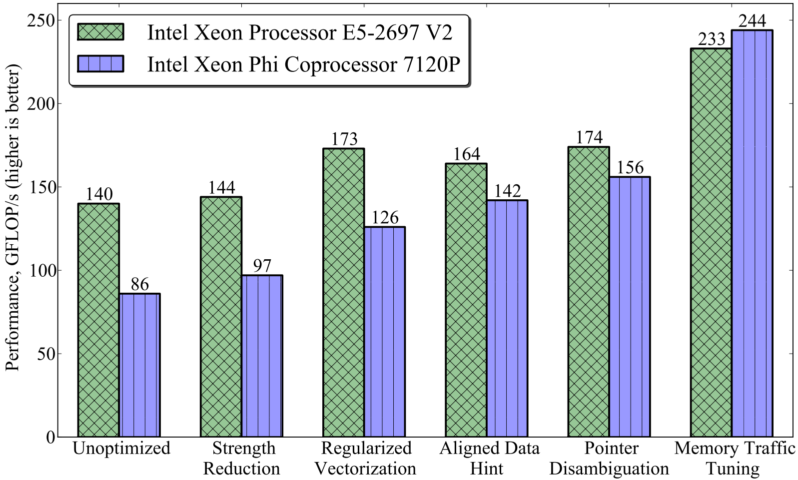
Before:

```
for (int b = 0; b < n; b++) {
  const int jMin = b - b%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 - b%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
}
```

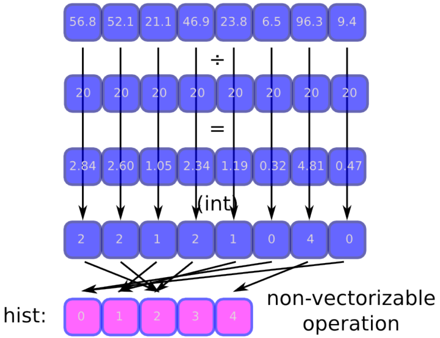

Example: Binning

Example: Strip-Mining for Vectorization

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,
  // group_width=20.0f
  const float group_width) {
    for (int i = 0; i < n; i++) {
      const int j = int(age[i]/group_width);
      hist[j]++;
    }
}
```

- Code cannot be auto-vectorized
- True vector dependence

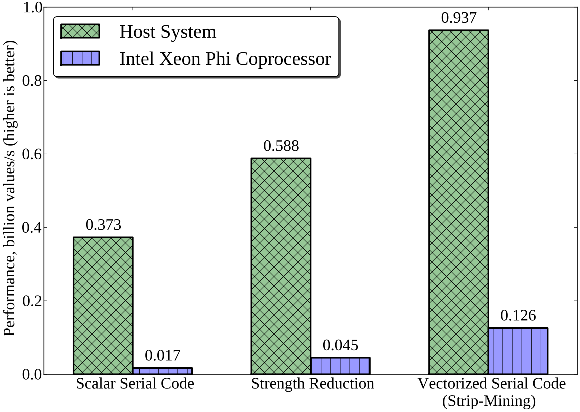


The Same Calculation, Strip-Mined, Vectorized

```
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.
   - Add padding to ensure alignment of multi-dimensional data, and to guarantee that the loop count is a multiple of vector length.
3. Remove Run-time Checks
   - Disable run-time checks for alignment and multi-versioning using the appropriate pragmas.
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

