



Programming and Optimization for Intel[®] 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

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

- 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-16-07



Forums (technical):

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.

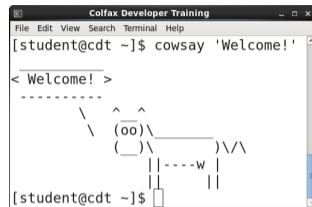
Welcome aboard. Post questions today!

Email (organizational):

training@colfax-intl.com

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. Expressing Data Parallelism

Vector Instructions in Intel Architecture

Computing Platforms

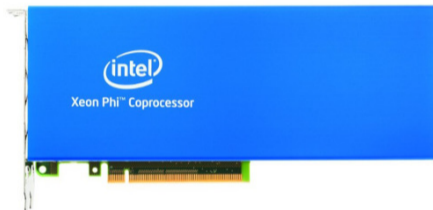
Intel Xeon Processor



Current: Broadwell
Upcoming: Skylake

Multi-Core Architecture

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*



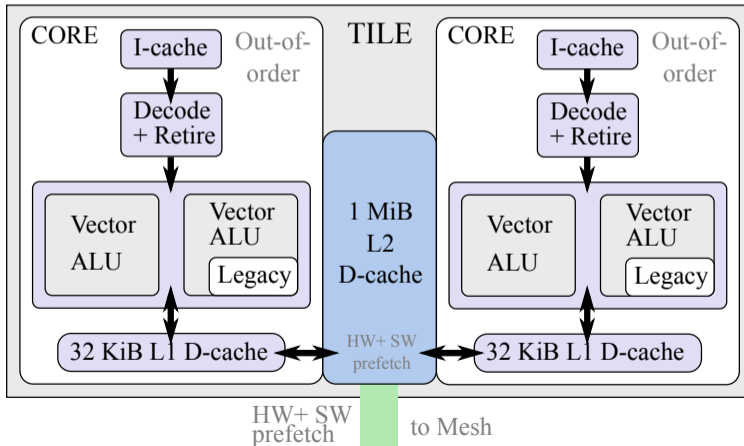
* socket and coprocessor versions

Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

KNL Cores

- Even more power in vector units
- Binary compatible with Xeon, but in legacy mode



Short Vector Support

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

Scalar Instructions

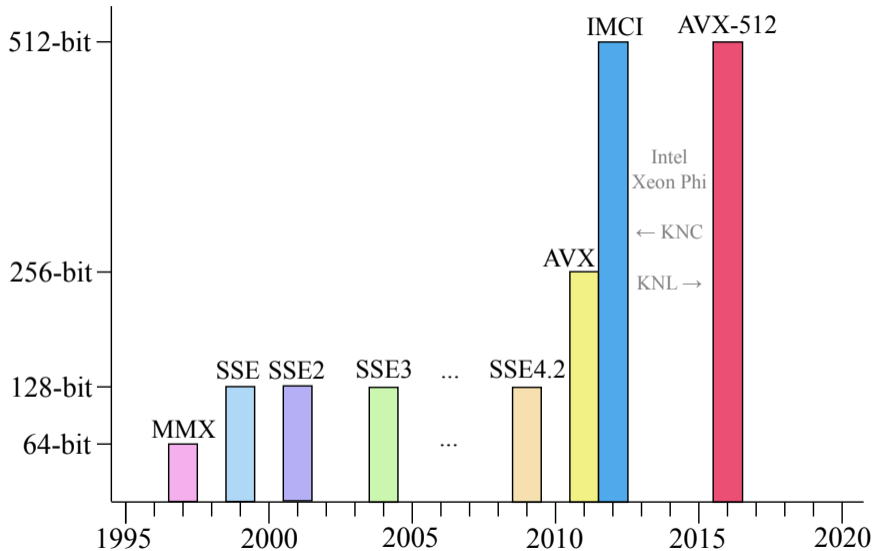
$$\begin{array}{r} 4 + 1 = 5 \\ 0 + 3 = 3 \\ -2 + 8 = 6 \\ 9 + -7 = 2 \end{array}$$

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

Instruction Sets in Intel Architecture



Intel Intrinsic Guide

<https://software.intel.com/sites/landingpage/IntrinsicsGuide>

The Intel Intrinsic Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

MMX
 SSE
 SSE2
 SSE3
 SSSE3
 SSE4.1
 SSE4.2
 AVX
 AVX2
 FMA
 AVX-512
 KNC
 SVML
 Other

Application-Targeted
 Arithmetic
 Bit Manipulation
 Cast
 Compare
 Convert
 Cryptography
 Elementary Math
 Functions
 General Support

`__m128i __mm_add_epi16 (__m128i a, __m128i b)` paddw
`__m128i __mm_add_epi32 (__m128i a, __m128i b)` paddd
`__m128i __mm_add_epi64 (__m128i a, __m128i b)` paddq
`__m128i __mm_add_epi8 (__m128i a, __m128i b)` paddb
`__m128d __mm_add_pd (__m128d a, __m128d b)` addpd

Synopsis

```
__m128d __mm_add_pd (__m128d a, __m128d b)
#include "emmintrin.h"
Instruction: addpd xmm, xmm
CPUID Flags: SSE2
```

Description

Add packed double-precision (64-bit) floating-point elements in *a* and *b*, and store the results in *dst*.

Operation

```
FOR j := 0 to 1
  i := j*64
  dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
```

Performance

Architecture	Latency	Throughput
Haswell	3	0.8
Ivy Bridge	3	1

Detecting Available Instructions

In the OS:

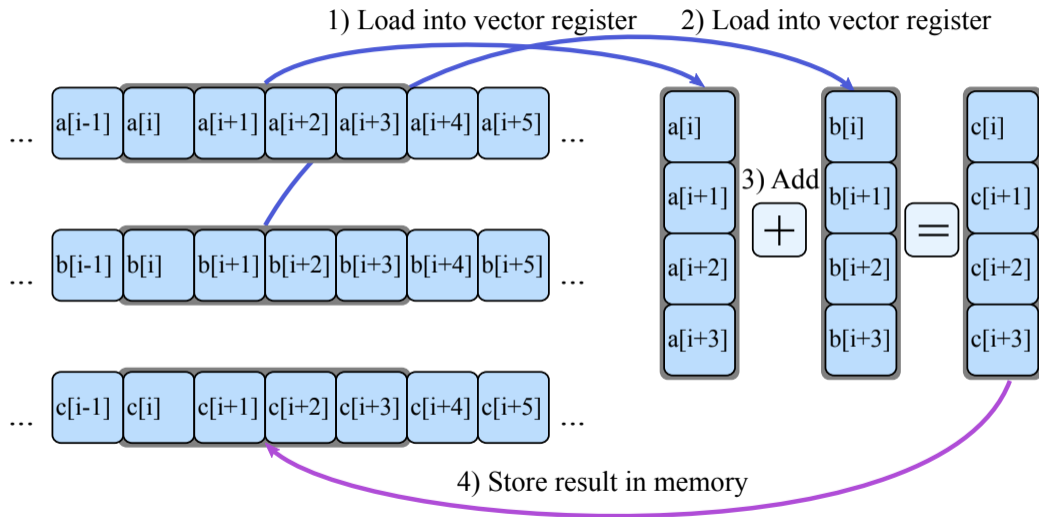
```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception    : yes
cpuid level      : 11
wp               : yes
flags            : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic popcnt aes hypervisor lahf_lm fsgsbase
bogomips        : 5985.17
clflush size     : 64
cache_alignment : 64
address sizes    : 46 bits physical, 48 bits virtual
...
```

In code (see also):

```
1 // Intel compiler
2 // preprocessor macros:
3
4 #ifdef __SSE__
5 // ...SSE code path
6 #endif
7
8 #ifdef __SSE4_2__
9 // ...SSE code path
10 #endif
11
12 #ifdef __AVX__
13 // ...AVX code path
14 #endif
```

Explicit Vectorization, Intrinsic

Workflow of Vector Computation



Example: Numerical Integration

$$I(a, b) = \int_a^b \frac{1}{\sqrt{x}} dx$$

Rectangle method:

$$\Delta x = \frac{b-a}{n},$$

$$x_i = (i+1)\Delta x,$$

$$I(a, b) = \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x).$$

```

1 float Integrate(const float a,
2                 const float b,
3                 const int N) {
4     const float dx = (b-a)/float(n);
5     float S = 0.0f;
6     for (int i = 0; i < n; i++) {
7         const float xi = dx*float(i+1);
8         S += 1.0f/sqrtf(xi) * dx;
9     }
10    return S;
11 }

```

Implementation with SSE4.2

```

1 float Integrate(const float a,
2                 const float b, const int n) {
3     __m128 dx = _mm_set1_ps((b - a)/float(n));
4     __m128 S  = _mm_set1_ps(0.0f);
5     for (int i = 0; i < n; i += 4) {
6         __m128i ip1 =
7             _mm_set_epi32(i+4, i+3, i+2, i+1);
8         __m128 ip1f = _mm_cvtepi32_ps(ip1);
9         __m128 xi = _mm_mul_ps(dx, ip1f);
10        __m128 fi = _mm_rsqrt_ps(xi);
11        __m128 dS = _mm_mul_ps(fi, dx);
12        S = _mm_add_ps(S, dS);
13    }
14    ConverterType c;
15    c.v = S;
16    return c.f[0] + c.f[1] + c.f[2] + c.f[3];
17 }

```

That is fine, *but...*

- Assuming n is a multiple of 4
- Only for SSE4.2 (circa 2011)
- No memory access. If we had some, peeling may be needed

Automatic Vectorization of Loops

Automatic Vectorization of Loops

```

1  #include <stdio.h>
2
3  int main(){
4      const int n=8;
5      int i;
6      int A[n] __attribute__((aligned(64)));
7      int B[n] __attribute__((aligned(64)));
8
9      // Initialization
10     for (i=0; i<n; i++)
11         A[i]=B[i]=i;
12
13     // This loop will be auto-vectorized
14     for (i=0; i<n; i++)
15         A[i]+=B[i];
16
17     // Output
18     for (i=0; i<n; i++)
19         printf("%2d %2d %2d\n", i, A[i], B[i]);
20 }

```

```

vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(14,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(14,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(14,3)]
LOOP END
...
vega@lyra% ./a.out
0 0 0
1 2 1
2 4 2
3 6 3
4 8 4
5 10 5
6 12 6
7 14 7

```

What Can Be Automatically Vectorized

Limitations:

- Only `for`-loops can be auto-vectorized. Number of iterations must be known at a runtime and/or compilation time
- By default, compiler targets the innermost loop for vectorization
- Memory access in the loop must have regular pattern, ideally with unit stride

What Cannot be Automatically Vectorized

Non-standard loops that cannot be automatically vectorized:

- loops with irregular memory access pattern
- calculations with vector dependence
- `while`-loops, `for`-loops with undetermined number of iterations
- outer loops (unless `#pragma simd` overrides this restriction)
- loops with complex branches (i.e., `if`-conditions)
- anything else that cannot be, or is very difficult to vectorize.

Vectorize more loops: `#pragma simd`

Statement `#pragma simd` is used to “enforce vectorization of loops”, which includes:

- Loops with SIMD-enabled functions (see below)
- Second innermost loops
- Failed vectorization due to compiler decision
- Loops where guidance is required (vector length, reduction, etc.)

See compiler reference on `#pragma simd` for more information.

Example for #pragma simd

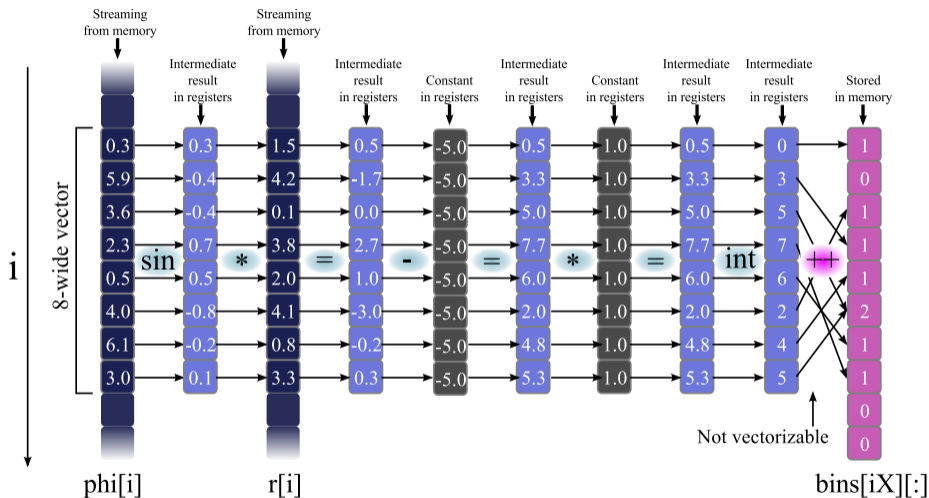
```
1  const int N=128;
2  const int T=4;
3  float A[N*N], B[N*N], C[T*T];
4
5  for (int jj = 0; jj < N; jj+=T) // Tile in j
6    for (int ii = 0; ii < N; ii+=T) // and tile in i
7      // Using pragma simd to vectorize outer loop:
8      #pragma simd
9      for (int k = 0; k < N; ++k) // long loop, vectorize it
10     for (int i = 0; i < T; i++) { // Loop between ii and ii+T
11       // Instead of a loop between jj and jj+T, unrolling that loop:
12       C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
13       C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
14       C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
15       C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
16     }
```

Auto-Vectorized Loops May Be Complex (Example 1)

```
1  for (int i = ii; i < ii + tileSize; i++) { // Target for auto-vectorization
2
3      // Newton's law of universal gravity
4      const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
5      const float dy = particle.y[j] - particle.y[i]; // x[i] makes SIMD vector
6      const float dz = particle.z[j] - particle.z[i];
7      const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
8      const float drPowerN32 = rr*rr*rr;
9
10     // Calculate the net force
11     Fx[i-ii] += dx * drPowerN32;
12     Fy[i-ii] += dy * drPowerN32;
13     Fz[i-ii] += dz * drPowerN32;
14 }
```

See also [this presentation](#)

Auto-Vectorized Loops May Be Complex (Example 2)



See [this paper](#) for more details

Array Notation

Extensions for Array Notation

Array notation is a method for specifying

- slices of arrays (begin, length)

```
1 A[0:16] += B[32:16]; // B[32]...B[47] added to A[0]...A[15]
```

- a stride (begin, length, stride)

```
1 A[0:16:2] += B[32:16:4]; // B[32],B[36]...B[92] added A[0],A[2]...A[30]
```

- Multi-dimensional arrays

```
1 A[:, :] += B[:, :]; // Add B to A; arrays are of the same shape
```

Better than strided loops (e.g., [this paper](#)).

Expressions with Array Notation May Be Complex

Example from <http://xeonphi.com/papers/efft>

```

1 evenrek[:] = evens[kk :kTILE:2];
2 evenimk[:] = evens[kk+1:kTILE:2];
3 oddrek [:] = odds [kk :kTILE:2];
4 oddimk [:] = odds [kk+1:kTILE:2];
5
6 evens[kk :kTILE:2] = evenrek[:] + coslist[:] * oddrek[:] - sinlist[:] * oddimk[:];
7 evens[kk+1:kTILE:2] = evenimk[:] + sinlist[:] * oddrek[:] + coslist[:] * oddimk[:];
8
9 oddmirrek[:] = odds[size-kk :kTILE:-2];
10 oddmirimk[:] = odds[size-kk+1:kTILE:-2];
11
12 odds[size-kk :kTILE:-2] =
13     evenrek[:] - coslist[:] * oddrek[:] + sinlist[:] * oddimk[:];
14 odds[size-kk+1:kTILE:-2] =
15     -evenimk[:] + sinlist[:] * oddrek[:] + coslist[:] * oddimk[:];
16 // ...

```

SIMD-Enabled Function

SIMD-Enabled Functions

(formerly “elemental functions”)

What if the implementation of a function is in a separate source code file (e.g., a library function)?

```
1 float my_simple_add(float x1, float x2){  
2     return x1 + x2;  
3 }
```

```
1 // ...in a separate source file:  
2 for (int i = 0; i < N, ++i) {  
3     output[i] = my_simple_add(inputa[i], inputb[i]);  
4 }
```

Compiler will refuse to automatically vectorize this loop.

SIMD-enabled Functions May Be Complex

Example from <http://xeonphi.com/papers/simd-lib>

```
1  __attribute__((vector)) float MyErfElemental(const float inx){
2      // Computes analytic approximation of the error function
3      const float x = fabsf(inx); // Take absolute value (in each vector lane)
4      const float p = 0.3275911f; // Constant parameter across vector lanes
5      const float t = 1.0f/(1.0f+p*x); // Expression in each vector lanes
6      const float l2e = 1.442695040f; // log2f(expf(1.0f))
7      const float e = exp2f(-x*x*l2e); // Transcendental in each vector lane
8      float res = -1.453152027f + 1.061405429f*t; // Computing a polynomial
9      res = 1.421413741f + t*res; // in each vector lane
10     res = -0.284496736f + t*res;
11     res = 0.254829592f + t*res;
12     res *= e;
13     res = 1.0f - t*res; // Analytic approximation in each vector lane
14     return copysignf(res, inx); // Copy sign in each vector lane
15 }
```

Helping the Compiler

Targeting a Specific Instruction Set

`-x[code]` instructs the compiler to target specific processor features, including instruction sets and optimizations.

code	Target architecture
MIC-AVX512	Intel Xeon Phi processors (KNL)
CORE-AVX512	Fugure Intel Xeon processors
CORE-AVX2	Intel Xeon processor E3 v3 family
CORE-AVX-I	Intel Xeon processor E3 v2, E5 v2 and E7 v2 family
AVX	Intel Xeon processor E3 and E5 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family
host	architecture on which the code is compiled

Assumed Vector Dependence

- True vector dependence makes vectorization impossible:

```

1 float *a, *b;
2 for (int i = 1; i < n; i++)
3     a[i] += b[i]*a[i-1]; // dependence on the previous element

```

- *Assumed vector dependence*: when compiler cannot determine whether vector dependence exists, auto-vectorization fails:

```

1 void mycopy(int n,
2             float* a, float* b) {
3     for (int i = 0; i < n; i++)
4         a[i] = b[i];
5 }

```

```

vega@lyra% icpc -c vdep.cc -qopt-report \
> -qopt-report-phase:vec
vega@lyra% cat vdep.optrpt
...
remark #15304: loop was not
vectorized: non-vectorizable loop
instance from multiversioning
...

```

Ignoring Assumed Vector Dependence

To ignore assumed vector dependence

```
#pragma ivdep
```

```
1 void mycopy(int n,  
2           float* a, float* b) {  
3   #pragma ivdep  
4   for (int i = 0; i < n; i++)  
5     a[i] = b[i];  
6 }
```

```
vega@lyra% icpc -c vdep.cc -qopt-report \  
> -qopt-report-phase:vec  
vega@lyra% cat vdep.optrpt  
...  
LOOP BEGIN at vdep.cc(4,1)  
<Multiversed v2>  
remark #15300: LOOP WAS VECTORIZED  
LOOP END
```

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`

```
1 #pragma ivdep
2   for (int i = 0; i < n; i++)
3     // ...
```

```
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]`).

Vectorization Pragmas, Keywords and Compiler Arguments

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `__assume_aligned` keyword
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`
- `-qopt-report -qopt-report-phase:vec`
- `-O[n]`
- `-x[code]`

Loop Was Vectorized, What Now?

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.

Review and What's Next

Discussed today:

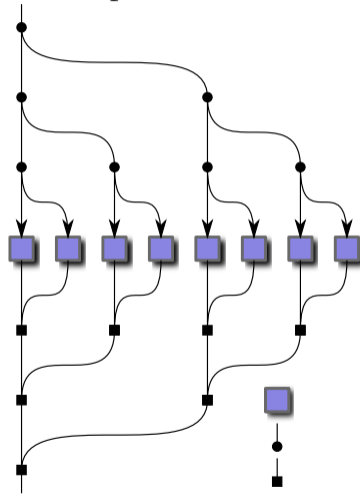
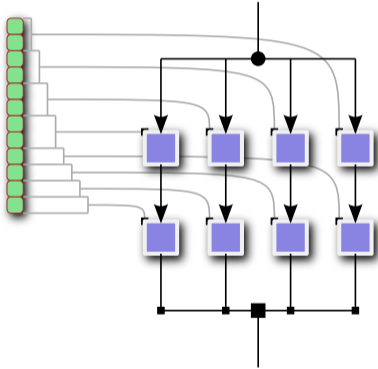
- Vectorization – support for data parallelism in each core
- Automatic vectorization enabled at default optimization level
- Argument `-qopt-report` produces a report on vectorization success
- SIMD-enabled functions, array notation and compiler hints

Later in the course – tuning automatic vectorization:

- alignment
- unit-stride data structures
- vectorization pattern regularization
- programming techniques for exposing automatic vectorization
- compiler hints.

What's Next

Next session: expressing thread parallelism with OpenMP.



[Learn More](#)

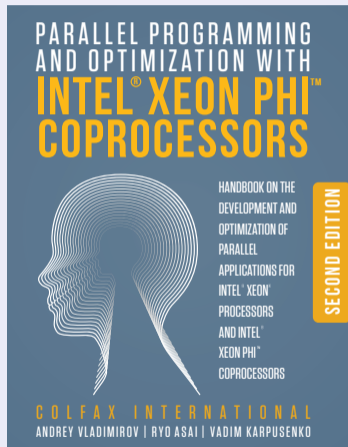
Textbook

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

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](#)



[Optimization Techniques for the Intel MIC Architecture, Part 3 of 3: Strip-Mining for Vectorization](#)



[Software Developer's Introduction to the HGST Ultrastar Archive HaaS SMR Drives](#)



[Optimization Techniques for the Intel MIC Architecture, Part 2 of 3: Strip-Mining for Vectorization](#)



[Optimization Techniques for the Intel MIC Architecture, Part 1 of 3: Multi-Threading and Parallel Reduction](#)



[Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors \(and why TX Acceleration May be Enough\)](#)

Featured Video

[Advanced Addressing](#)

[Intel Research National Lab visualization for a floating point](#)

[http://colfaxresearch.com/?p=768](#)

Events

[Presentations](#)

[Courses](#)

Consulting

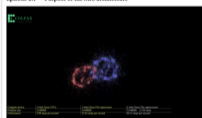


Colfax offers consulting services for enterprises, research and education to help you to:

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



[http://colfaxresearch.com/?p=768](#)

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



[http://colfaxresearch.com/?p=768](#)

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



[http://colfaxresearch.com/?p=768](#)

Parallel Computing in the Search for New Physics at LHC



[http://colfaxresearch.com/?p=768](#)

Fluid Dynamics with Fortran on Intel Xeon Phi coprocessors



[http://colfaxresearch.com/?p=768](#)

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



[http://colfaxresearch.com/?p=768](#)

<http://colfaxresearch.com/>