MODERN CODE PRACTICES AND INTEL® ARCHITECTURE

Part 2 of 3

Colfax International — *colfaxresearch.com*

November 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

- **Part 1**: Multi-threading strategies
  - Minimizing synchronization
  - Avoiding false sharing
  - Exposing parallelism
- **Part 2**: Vectorization tuning
  - Roles of compiler and developer
  - Tuning with directives
  - Data container optimization
  - Language extensions
- **Part 3**: Memory traffic control
  - Maximizing cache utilization
  - Optimizing memory bandwidth
  - Intel Xeon Phi processors: high-bandwidth memory

§2. INTEL ARCHITECTURE

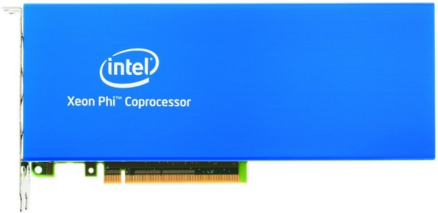
COMPUTING PLATFORMS

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

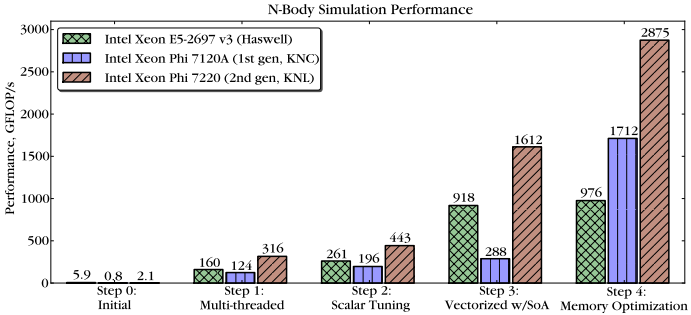
Knights Landing (KNL)

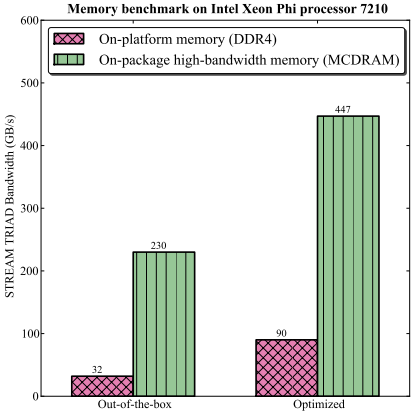
Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

PERFORMANCE OPTIMIZATION

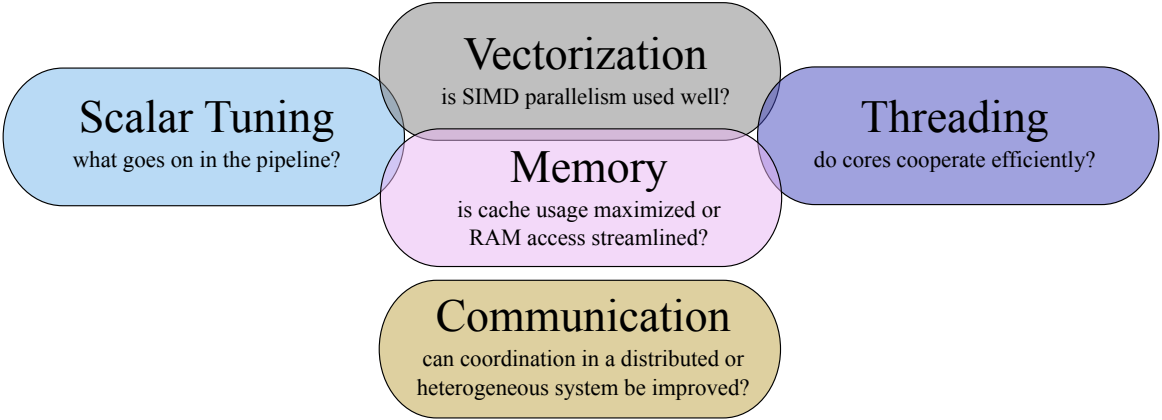
IT TAKES GOOD SOFTWARE TO UNLOCK THE PERFORMANCE!





Details on N-body simulation in Chapter 23 of this book

OPTIMIZATION AREAS

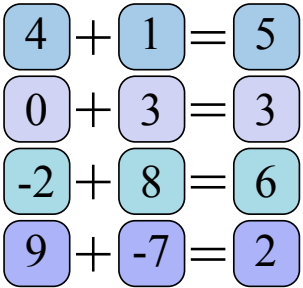


§3. VECTORIZATION

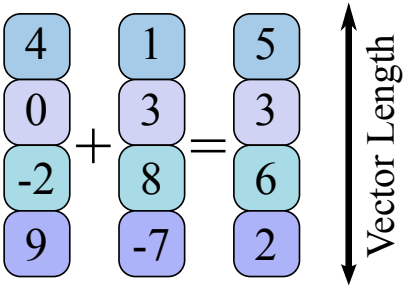
SHORT VECTOR SUPPORT

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

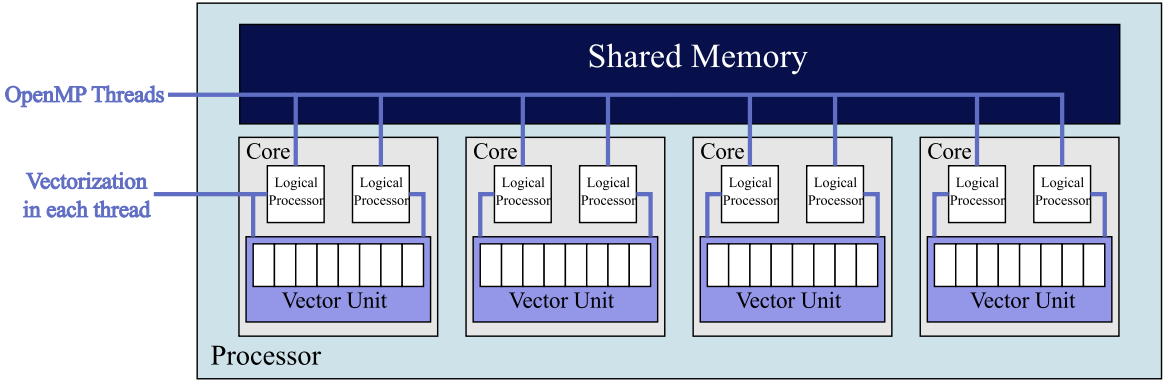
Scalar Instructions



Vector Instructions



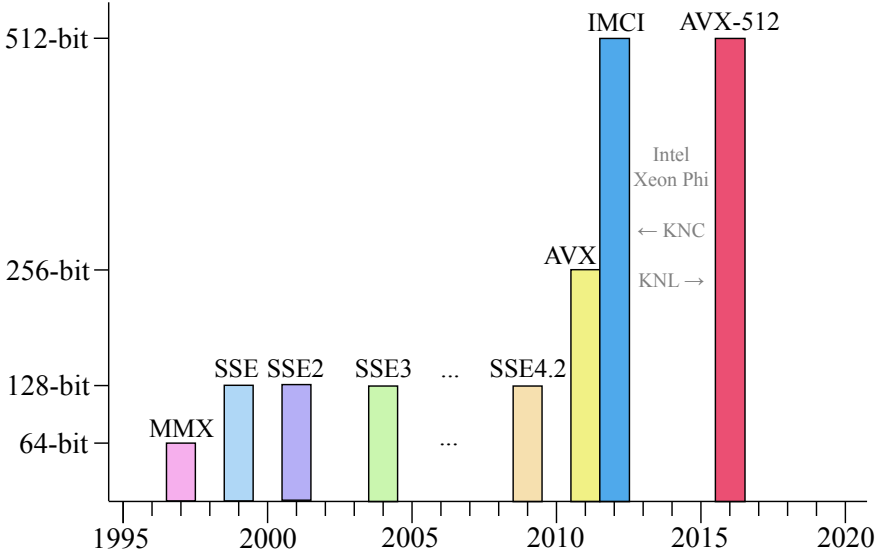
CO-EXISTENCE WITH VECTORS



Utilize cores: run multiple threads/processes (MIMD)

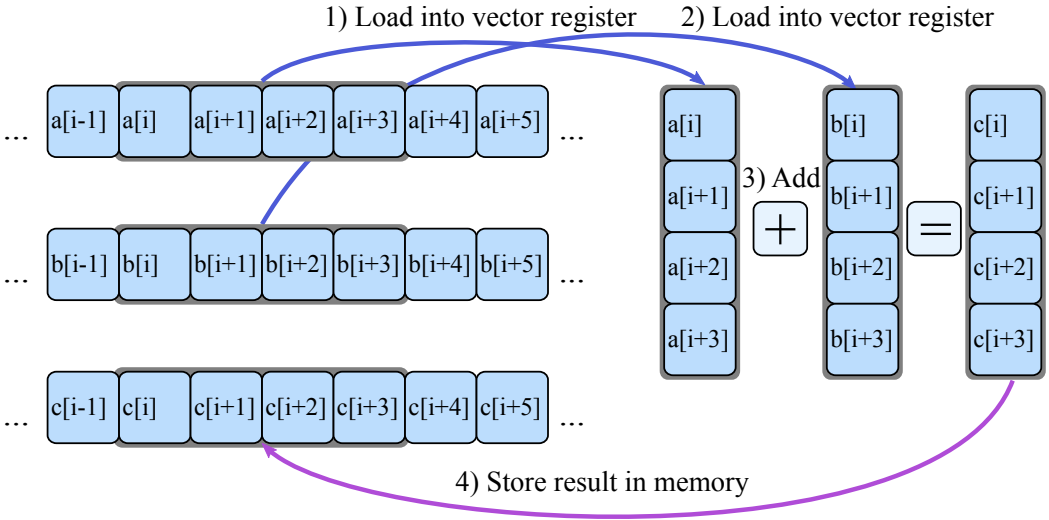
Utilize vectors: each thread (process) issues vector instructions (SIMD)

INSTRUCTION SETS IN INTEL ARCHITECTURE



AUTOMATIC OR EXPLICIT VECTORIZATION

WORKFLOW OF VECTOR COMPUTATION



EXAMPLE: NUMERICAL INTEGRATION

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

Rectangle method:

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

```
float Integrate(const float a,
                const float b,
                const int N) {
  const float dx = (b-a)/float(n);
  float S = 0.0f;
  for (int i = 0; i < n; i++) {
    const float xi = dx*float(i+1);
    S += 1.0f/sqrtf(xi) * dx;
  }
  return S;
}
```

INTEL INTRINSICS GUIDE

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

Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=MMX,SSE,SSE2,SSE3,SSSE3,SSE

Search

Intrinsics Guide

The Intel Intrinsics 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.

Technologies

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

Categories

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

_mm_search

```
__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

IMPLEMENTATION WITH SSE4.2

```
float Integrate(const float a,
                const float b, const int n) {
  __m128 dx = _mm_set1_ps((b - a)/float(n));
  __m128 S  = _mm_set1_ps(0.0f);
  for (int i = 0; i < n; i += 4) {
    __m128i ip1  =
             _mm_set_epi32(i+4, i+3, i+2, i+1);
    __m128  ip1f = _mm_cvtepi32_ps(ip1);
    __m128  xi = _mm_mul_ps(dx, ip1f);
    __m128  fi = _mm_rsqrt_ps(xi);
    __m128  dS = _mm_mul_ps(fi, dx);
    S  = _mm_add_ps(S, dS);
  }
  ConverterType c;
  c.v = S;
  return c.f[0] + c.f[1] + c.f[2] + c.f[3];
}
```

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

```cpp
#include <cstdio>

int main(){
  const int n=8;
  int i;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

  // Initialization
  for (i=0; i<n; i++)
    A[i]=B[i]=i;

  // This loop will be auto-vectorized
  for (i=0; i<n; i++)
    A[i]+=B[i];

  // Output
  for (i=0; i<n; i++)
    printf("%2d %2d %2d\n", i, A[i], B[i]);
}
```

```
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
```

LIMITATIONS ON AUTOMATIC VECTORIZATION

- Number of iterations must be known before start of loop
- Only innermost loops (possible to override)
- No vector dependence allowed
- Functions called from vector loops must be SIMD-enabled

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

LANGUAGE EXTENSIONS

EXTENSIONS FOR ARRAY NOTATION

Array notation is a method for specifying

- slices of arrays (begin, length)

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

- a stride (begin, length, stride)

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

- Multi-dimensional arrays

```
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

```
evenrek[:] = evens[kk  :kTILE:2];
evenimk[:] = evens[kk+1:kTILE:2];
oddrek [:] = odds [kk  :kTILE:2];
oddimk [:] = odds [kk+1:kTILE:2];

evens[kk  :kTILE:2] = evenrek[:] + coslist[:]*oddrek[:] - sinlist[:]*oddimk[:];
evens[kk+1:kTILE:2] = evenimk[:] + sinlist[:]*oddrek[:] + coslist[:]*oddimk[:];

oddmirrek[:] = odds[size-kk  :kTILE:-2];
oddmirimk[:] = odds[size-kk+1:kTILE:-2];

odds[size-kk  :kTILE:-2] =
                      evenrek[:] - coslist[:]*oddrek[:] + sinlist[:]*oddimk[:];
odds[size-kk+1:kTILE:-2] =
                     -evenimk[:] + sinlist[:]*oddrek[:] + coslist[:]*oddimk[:];
// ...
```

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)?

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

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

C

will refuse to automatically vectorize this loop.

SIMD-ENABLED FUNCTIONS MAY BE COMPLEX

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

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

UNIT-STRIDE ACCESS

UNIT-STRIDE ACCESS

Unit-stride access is optimal:

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

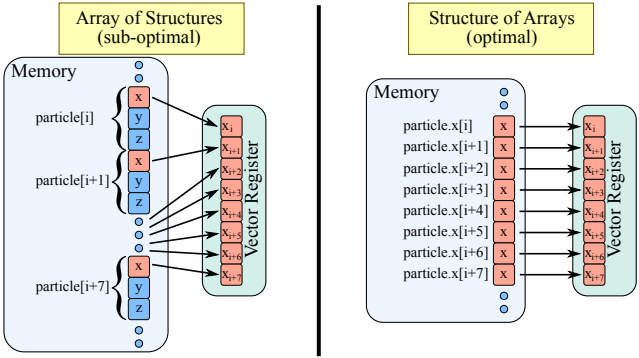
Non-unit stride is slower:

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

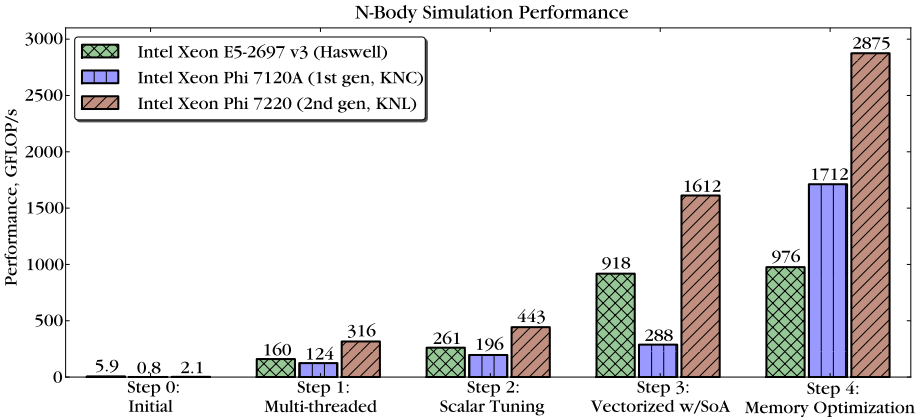
Stochastic access may be vectorized (but not efficient):

```
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:



UNIT-STRIDE ACCESS AT WORK



Details on N-body simulation in Chapter 23 of this book

ALIGNMENT AND PADDING

DATA ALIGNMENT REQUIREMENTS

Array `char* p` is `n`-byte aligned if `((size_t)p%n==0)`.

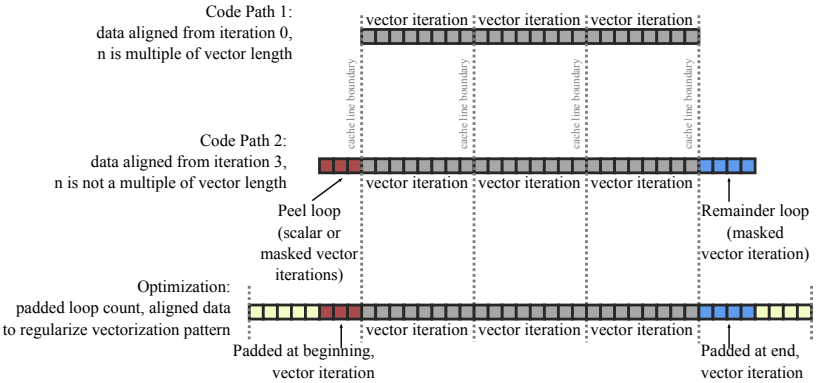
Processor	Operation	Alignment
Xeon (Westmere and earlier)	SSE load, store	16-byte
Xeon (Sandy Bridge and later)	AVX load, store	32-byte (relaxed)
Xeon Phi (1st gen)	IMCI load, store	64-byte (strict)
Xeon Phi (1st gen)	DMA transfer in offload	4096-byte (preferred)
Xeon Phi (2nd gen)	AVX-512 load, store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

COMPILER DIRECTIVES

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]`

§4. LEARN MORE

THE "HOW" SERIES TRAINING

DEEP DIVE

WITH CODE MODERNIZATION EXPERTS

It's free

→ HOWSERIES.COM

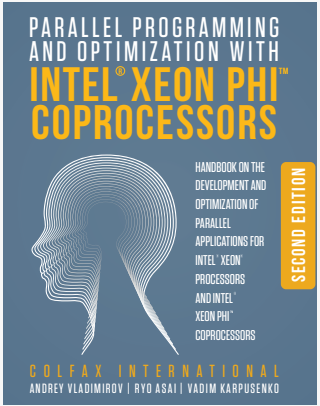
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system

TEXTBOOK

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

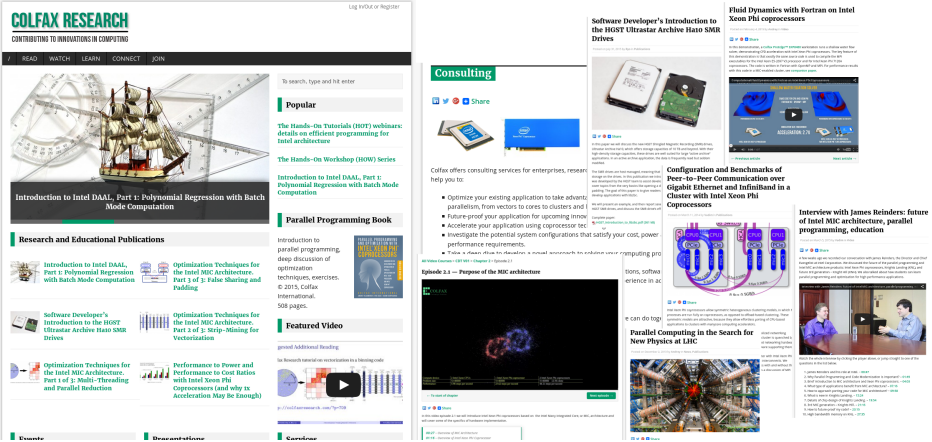
Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors

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



http://xeonphi.com/book

COLFAX RESEARCH



http://colfaxresearch.com/

LEARN MORE

DEVELOPER ACCESS PROGRAM (DAP)

Can't wait to get your hands on Knights Landing?

Find out more at dap.xeonphi.com
or contact us at dap@colfax-intl.com

BOOTABLE INTEL XEON PHI PROCESSORS

- Bootable Host Processor
- RHEL/CentOS/SUSE/Win
- 64 cores × 4 HT, 1.3 GHz
- ≤ 384 GiB DDR4, > 90 GB/s
- 16 GiB HBM, > 400 GB/s
- PCIe bus for networking

dap.xeonphi.com

Servers:

Workstations: