PROGRAMMING AND OPTIMIZATION FOR INTEL® ARCHITECTURE

The Hands-On Workshop (HOW) Series

Session 3

Colfax International — *colfaxresearch.com*

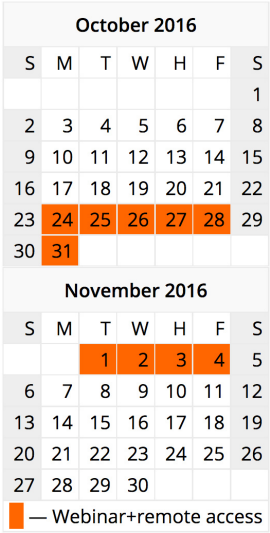
October 2016

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 – Oct 24
  - 02. Programming Intel Xeon Phi (KNC, KNL) – Oct 25
- HOW to Express Parallelism
  - 03. Automatic vectorization – Oct 26
  - 04. Multi-threading with OpenMP – Oct 27
- HOW to Get Performance
  - 05. Comprehensive demo – Oct 28
  - 06. Scalar & vectorization tuning – Oct 31
  - 07. Multi-threading: common issues – Nov 1
  - 08. Multi-threading: memory aspect – Nov 2
  - 09. Memory traffic – Nov 3
- HOW to Scale
  - 10. Distributed Computing: MPI – Nov 4



HOW ONLINE

Course page: colfaxresearch.com/how-16-10

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

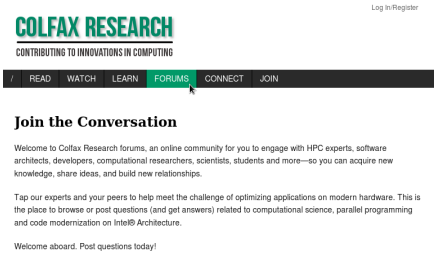
Additional resources:

- More workshops like this one: colfaxresearch.com/training
- Video courses: colfaxresearch.com/video-courses

GET YOUR QUESTIONS ANSWERED

Chat (current):
colfaxresearch.com/how-16-10

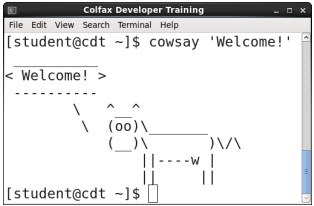
Forums (technical):
colfaxresearch.com/discussion



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. EXPRESSING DATA PARALLELISM

VECTOR INSTRUCTIONS IN INTEL ARCHITECTURE

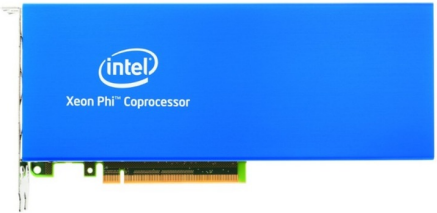
COMPUTING PLATFORMS

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*

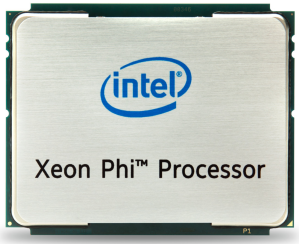
* socket and coprocessor versions

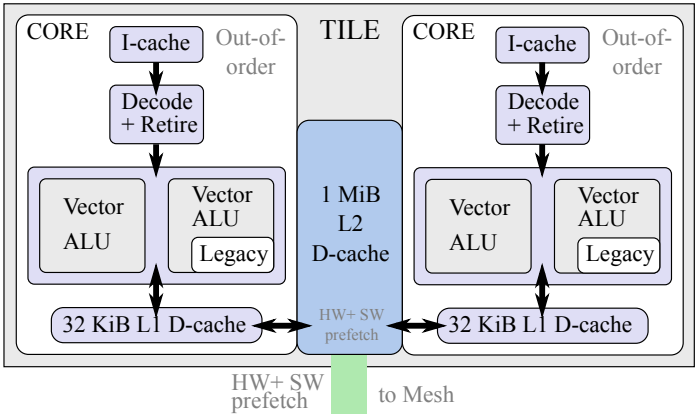
Knights Landing (KNL)

Intel Many Integrated Core (MIC) Architecture

KNL CORES

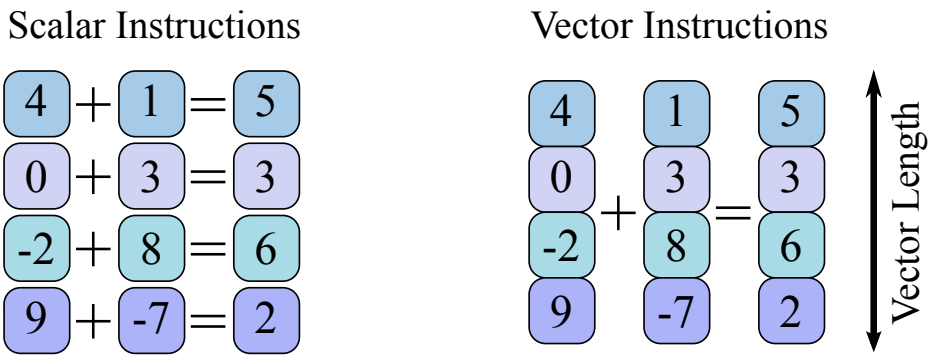
- Even more power in vector units
- Binary compatible with Xeon, but in legacy mode



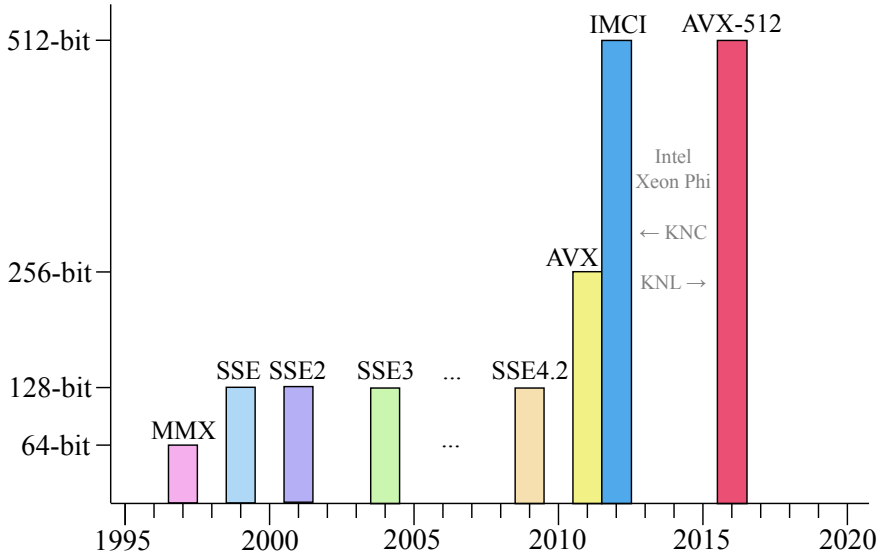


SHORT VECTOR SUPPORT

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

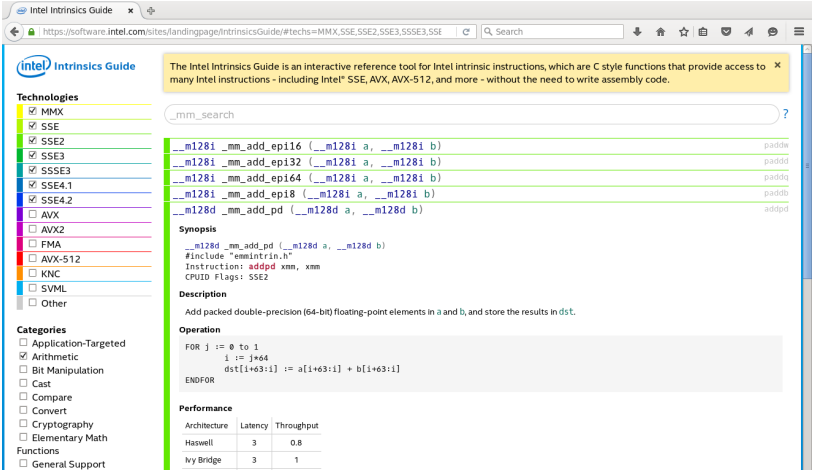


INSTRUCTION SETS IN INTEL ARCHITECTURE



INTEL INTRINSICS GUIDE

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



DETECTING AVAILABLE INSTRUCTIONS

In the OS:

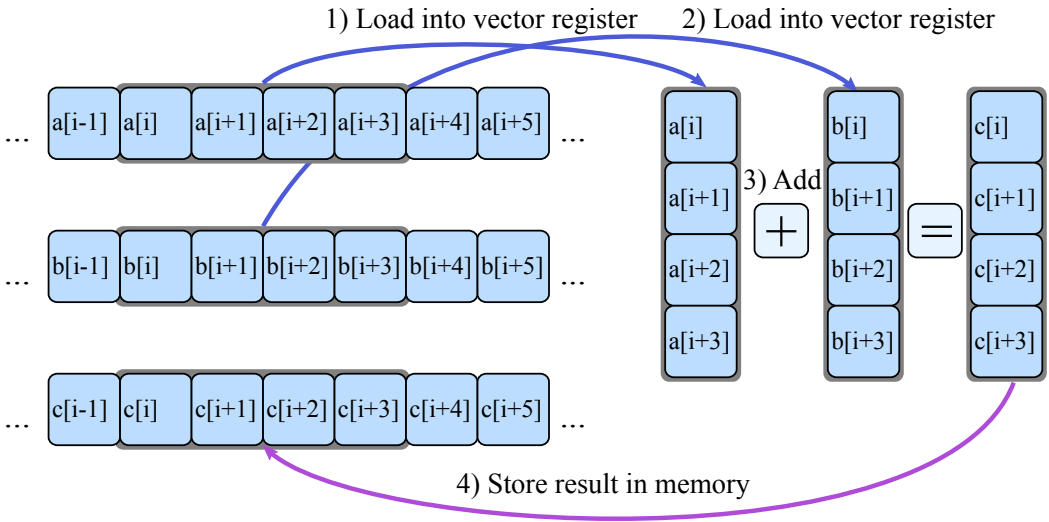
```
[student@cdt ~]% cat /proc/cpuinfo
...
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce
cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 ss ht syscall nx lm constant_tsc
unfair_spinlock pni ssse3 cx16 sse4_1 sse4_2
x2apic 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, INTRINSICS

WORKFLOW OF VECTOR COMPUTATION



EXAMPLE: NUMERICAL INTEGRATION

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

Rectangle method:

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

```
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

```
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

AUTOMATIC VECTORIZATION OF LOOPS

```
#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

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

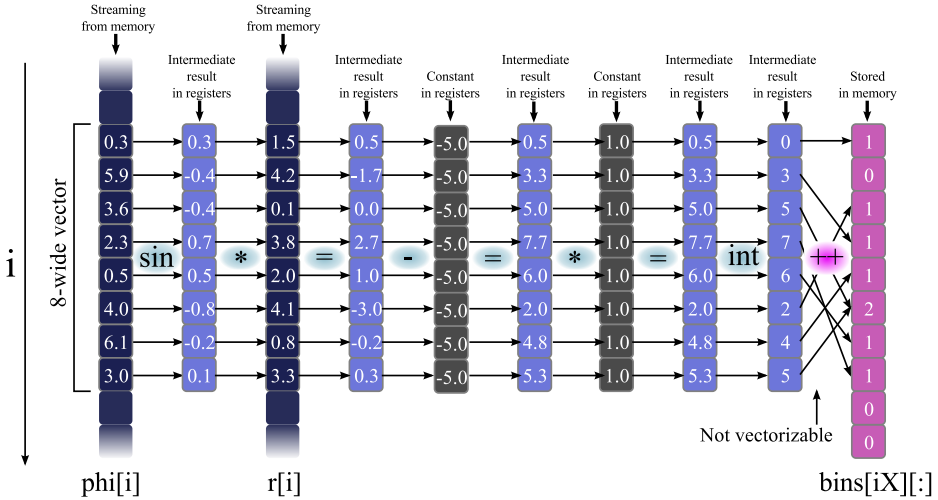
```
1  const int N=128;
2  const int T=4;
3  float A[N*N], B[N*N], C[T*T];
4
5  for (int jj = 0; jj < N; jj+=T) // Tile in j
6    for (int ii = 0; ii < N; ii+=T) // and tile in i
7    // Using pragma simd to vectorize outer loop:
8  #pragma simd
9    for (int k = 0; k < N; ++k) // long loop, vectorize 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 FUNCTIONS

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

```
__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
}
```

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 wheter 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)
<Multiversioned 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:

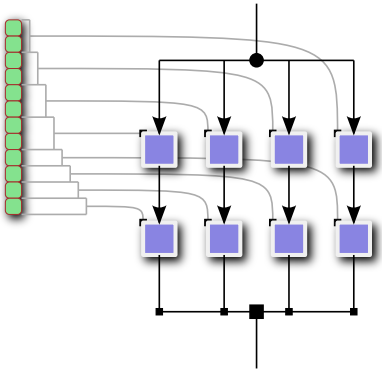
- Vectorization – support for data parallelism in each core
- Automatic vectorization enabled at default optimization level
- Loops, SIMD-enabled functions, array notation
- Argument `-qopt-report` produces a report on vectorization success

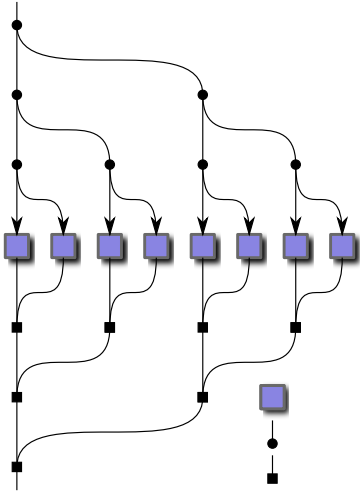
Later in the course – tuning automatic vectorization:

- 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

HOW SERIES: KNIGHTS LANDING

colfaxresearch.com/how-knl/

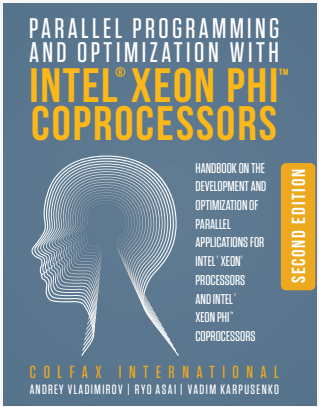
LEARN MORE

TEXTBOOK

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

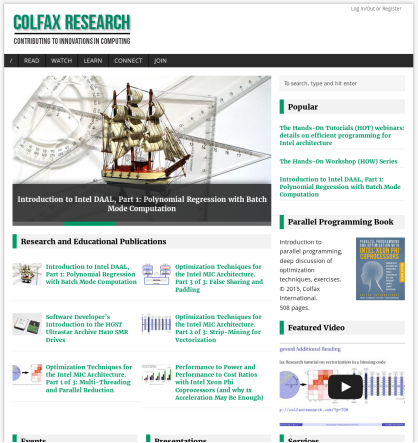
Parallel Programming
and Optimization with
Intel® Xeon Phi™
Coprocessors

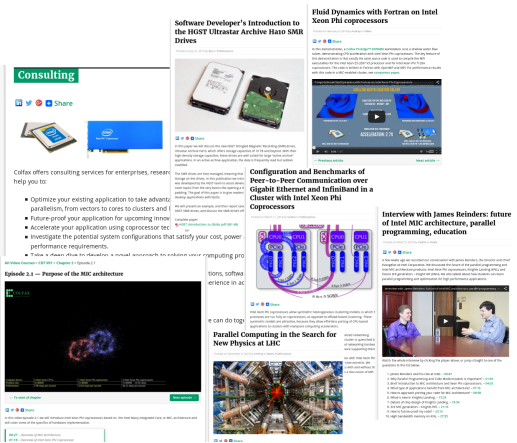
Handbook on the Development and
Optimization of Parallel Applications
for Intel® Xeon® Processors
and Intel® Xeon Phi™ Coprocessors



http://xeonphi.com/book

COLFAX RESEARCH





http://colfaxresearch.com/

LEARN MORE