COLFAX
Customized Solutions

Programming and Optimization for Intel® Architecture

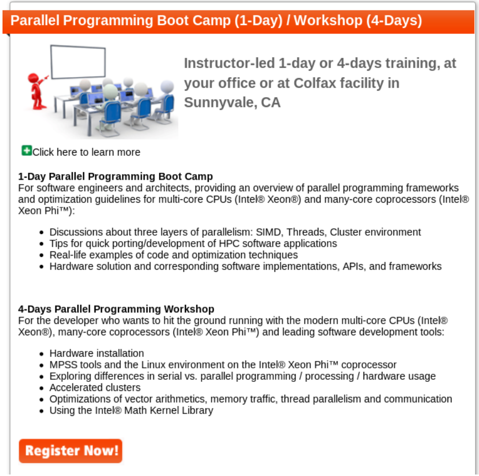
The Hands-On Workshop (HOW) Series

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.



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:

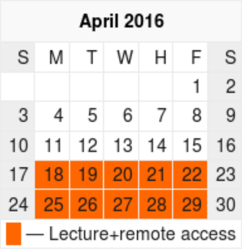
- Hardware installation
- MPSS tools and the Linux environment on the Intel® Xeon Phi™ coprocessor
- Exploring differences in serial vs. parallel programming / processing / hardware usage
- Accelerated clusters
- Optimizations of vector arithmetics, memory traffic, thread parallelism and communication
- Using the Intel® Math Kernel Library

colfaxresearch.com/how-series

Disclaimer

Course Roadmap

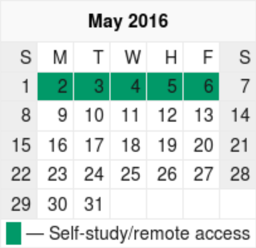
1. Why Intel Parallel Architectures?
    - Parallelism and specialization – April 18
    - Programming model continuity – April 18
2. Programming models for Xeon Phi coprocessors
    - Native programming – April 18
    - Offload programming – April 19
3. Expressing Parallelism
    - Introduction to vectorization – 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

Developer’s Guide to Knights Landing

colfaxresearch.com/knl-webinar/

§2. Expressing Data Parallelism

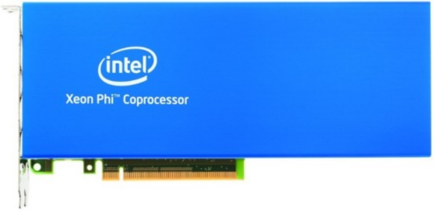
Vector Instructions in Intel Architecture

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

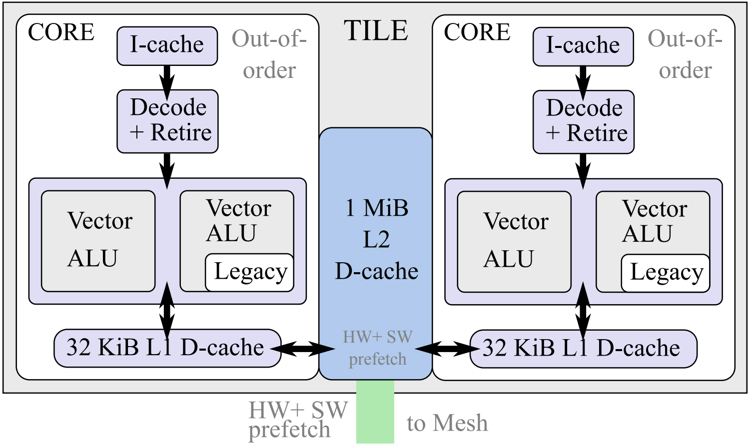
Upcoming: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

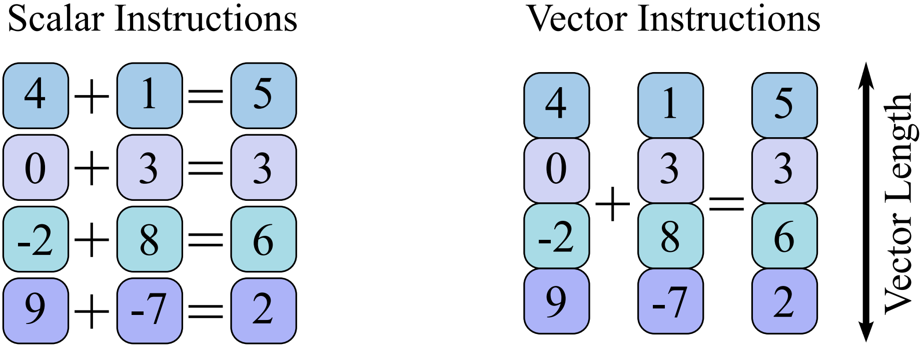
KNL Cores

- 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 Architectures

Instruction Set	Year and Intel Processor	Vector registers	Packed Data Types
MMX	1997, Pentium	64-bit	8-, 16- and 32-bit integers
SSE	1999, Pentium III	128-bit	32-bit single precision FP
SSE2	2001, Pentium 4	128-bit	8 to 64-bit integers; SP & DP FP
SSE3–SSE4.2	2004 – 2009	128-bit	(additional instructions)
AVX	2011, Sandy Bridge	256-bit	single and double precision FP
AVX2	2013, Haswell	256-bit	integers, additional instructions
IMCI	2012, Knights Corner	512-bit	32- and 64-bit integers; single & double precision FP
AVX-512	Knights Landing	512-bit	32- and 64-bit integers; single & double precision FP

Features of the IMCI Instruction Set

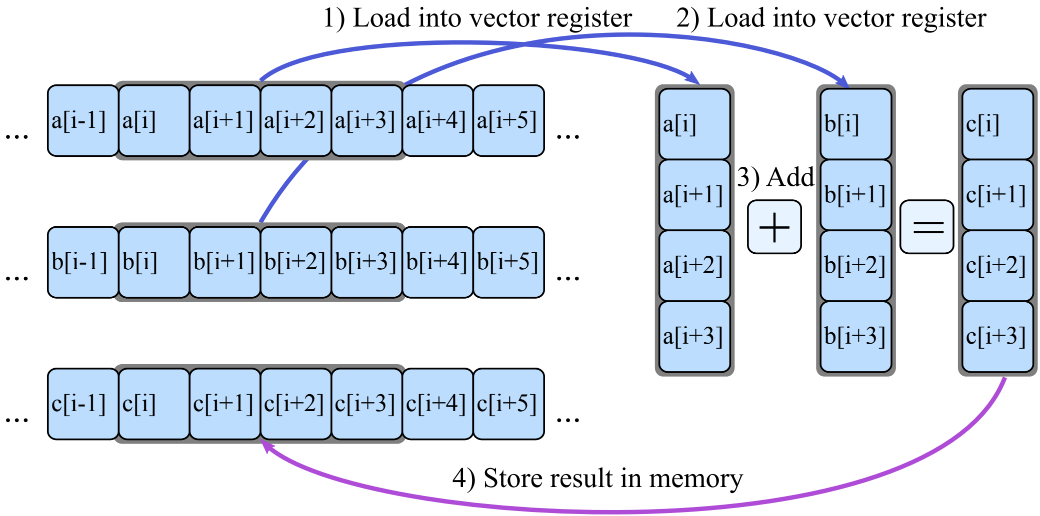
Knight's Corner uses Initial Many Core Instruction (IMCI) set.
512-bit wide registers: can hold 8 DP or 16 SP values.

IMCI Supports:

- Initialization, Load and Store, Gather and Scatter
- Arithmetic Instructions: Binary Operators, Transcendental Functions, etc.
- Comparison
- Conversion and type cast
- Bitwise instructions: `NOT, AND, OR, XOR, XAND`
- Reduction and minimum/maximum instructions
- Vector mask instructions

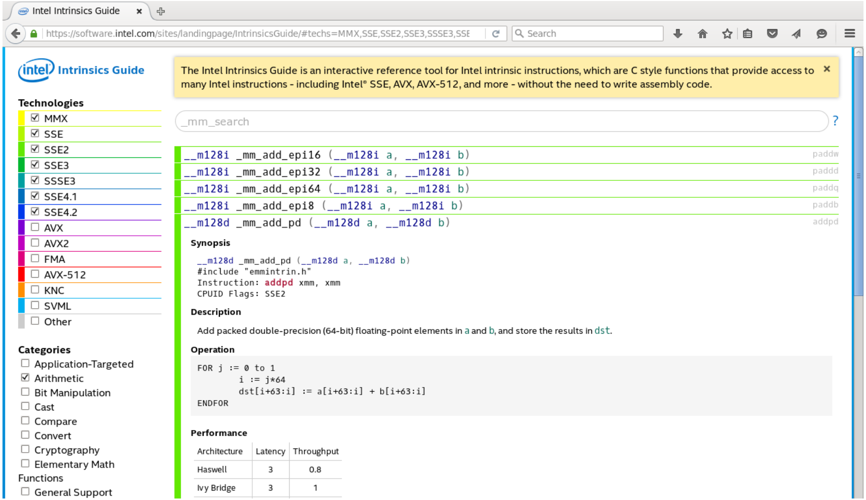
Explicit Vectorization, Intrinsics

Workflow of Vector Computation



Intel Intrinsics Guide

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



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

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):

```
// Intel compiler
// preprocessor macros:

#ifdef __SSE__
// ...SSE code path
#endif

#ifdef __SSE4_2__
// ...SSE code path
#endif

#ifdef __AVX__
// ...AVX code path
#endif
```

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;
}
```

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 <stdio.h>

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 -qopt-report-phase:vec
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

```
const int N=128;
const int T=4;
float A[N*N], B[N*N], C[T*T];

for (int jj = 0; jj < N; jj+=T) // Tile in j
  for (int ii = 0; ii < N; ii+=T) // and tile in i
  // Using pragma simd to vectorize outer loop:
#pragma simd
  for (int k = 0; k < N; ++k) // long loop, vectorize it
    for (int i = 0; i < T; i++) { // Loop between ii and ii+T
      // Instead of a loop between jj and jj+T, unrolling that loop:
      C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
      C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
      C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
      C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
}
```

Auto-Vectorized Loops May Be Complex (Example 1)

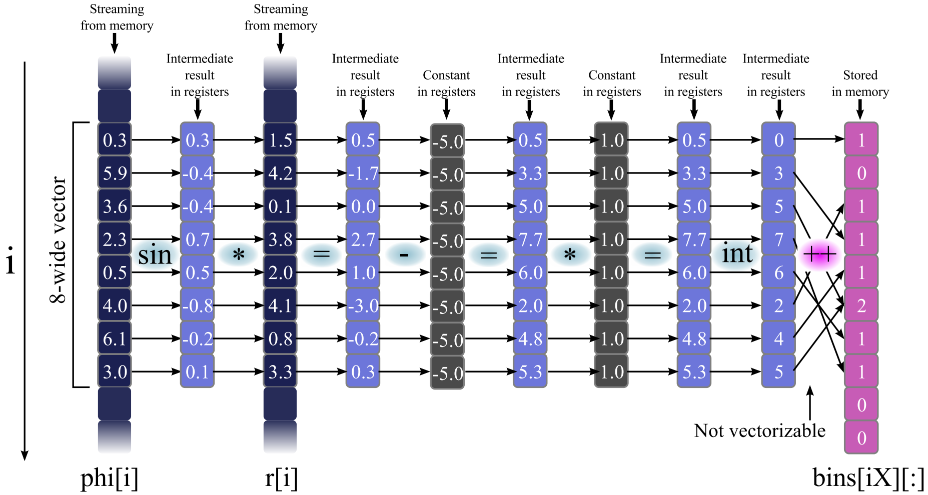
```
  for (int i = ii; i < ii + tileSize; i++) { // Target for auto-vectorization

    // Newton's law of universal gravity
    const float dx = particle.x[j] - particle.x[i]; // x[j] is a const
    const float dy = particle.y[j] - particle.y[i]; // x[i] makes SIMD vector
    const float dz = particle.z[j] - particle.z[i];
    const float rr = 1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
    const float drPowerN32  = rr*rr*rr;

    // Calculate the net force
    Fx[i-ii] += dx * drPowerN32;
    Fy[i-ii] += dy * drPowerN32;
    Fz[i-ii] += dz * drPowerN32;
  }
```

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)

```
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 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)?

```
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]);
}
```

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

Assumed Vector Dependence

- True vector dependence makes vectorization impossible:

```
float *a, *b;
for (int i = 1; i < n; i++)
  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:

```
void mycopy(int n,
            float* a, float* b) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

```
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`

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

```
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`

```
#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]`).

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.