Best Practices for Vectorization

Getting ready for Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

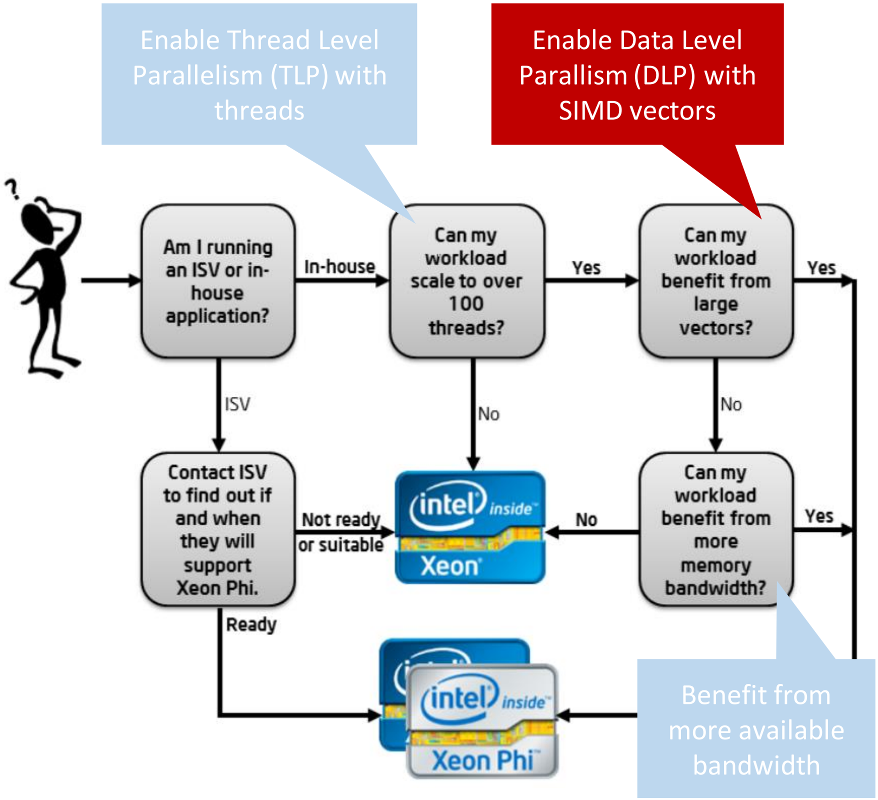
Manel Fernández

Intel HPC Software Workshop Series 2015

May 6th 2015, Munich

The need for SIMD vectorization

Is the Intel® Xeon Phi^{TM} coprocessor right for me?



Enable Thread Level Parallelism (TLP) with threads

Enable Data Level Parallism (DLP) with SIMD vectors

Am I running an ISV or in-house application?

In-house

ISV

Can my workload scale to over 100 threads?

Yes

No

Can my workload benefit from large vectors?

Yes

No

Contact ISV to find out if and when they will support Xeon Phi.

Not ready or suitable

Ready

No

Can my workload benefit from more memory bandwidth?

Yes

Benefit from more available bandwidth

Single thread (ST) performance is limited in today’s CPUs

- Clock frequency constraints
- Difficult to discover “near” Instruction level parallelism (ILP) by hardware

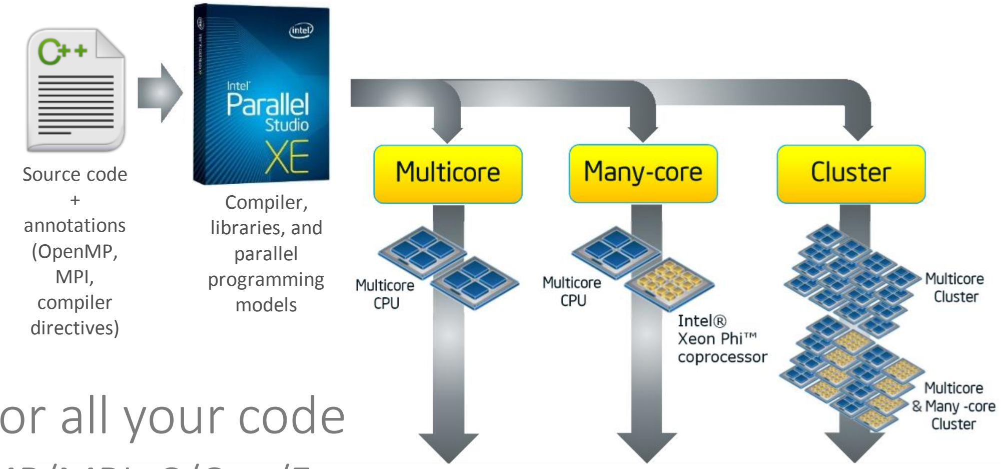
More transistors dedicated to exploit “distant” parallelism

- Task level parallelism (TLP)
  - Improves Multi Thread performance (MT)
- **Data level parallelism (DLP)**
  - Improves Single Thread performance (ST)
  - **Enabled by using SIMD vectors**

“Is the Intel® Xeon Phi^{TM} coprocessor right for me?”, by Eric Gardner - https://software.intel.com/en-us/articles/is-the-intel-xeon-phi-coprocessor-right-for-me

How to enable SIMD vectorization?

Enabling parallelism with Intel® Parallel Studio XE 2015 tool suite



Single programming model for all your code

- Based on standards: OpenMP/MPI, C/C++/Fortran
- Programmers/tools responsibility to expose DLP/TLP parallelism

Exposing TLP/DLP in your application will benefit today and future Intel® Xeon® processors and Intel® Xeon Phi^{TM} coprocessors

- Including SIMD vectorization on future Intel® AVX-512 products

Single Instruction Multiple Data (SIMD)

Technique for exploiting DLP on a single thread

- Operate on more than one element at a time
- Might decrease instruction counts significantly

Elements are stored on SIMD registers or *vectors*

Code needs to be *vectorized*

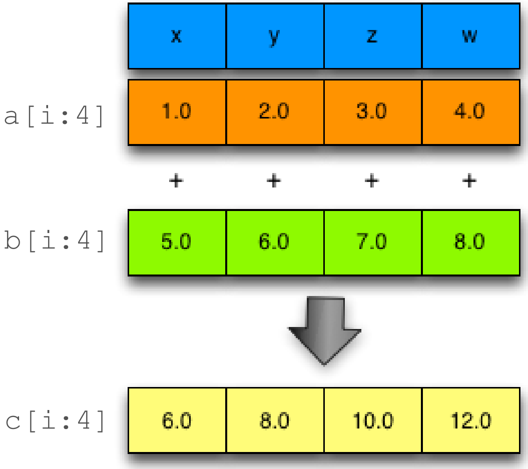
- Vectorization usually on *inner* loops
- Main and *remainder* loops are generated

Scalar loop

```
for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

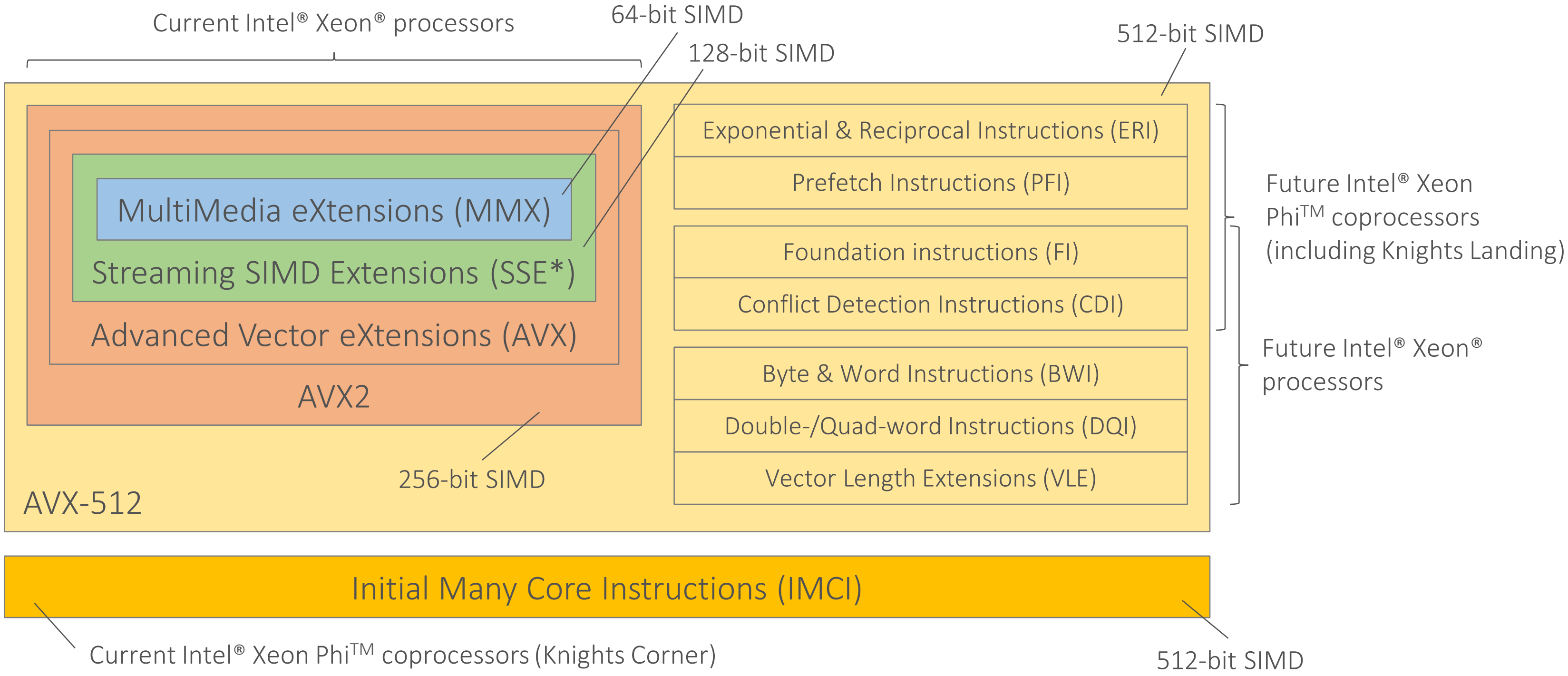
SIMD loop (4 elements)

```
for (int i = 0; i < N; i += 4)
    c[i:4] = a[i:4] + b[i:4];
```



	x	y	z	w
a[i:4]	1.0	2.0	3.0	4.0
	+	+	+	+
b[i:4]	5.0	6.0	7.0	8.0
c[i:4]	6.0	8.0	10.0	12.0

Past, present, and future of Intel SIMD types



Current Intel® Xeon® processors
64-bit SIMD
128-bit SIMD
512-bit SIMD
MultiMedia eXtensions (MMX)
Streaming SIMD Extensions (SSE*)
Advanced Vector eXtensions (AVX)
AVX2
256-bit SIMD
AVX-512
Exponential & Reciprocal Instructions (ERI)
Prefetch Instructions (PFI)
Foundation instructions (FI)
Conflict Detection Instructions (CDI)
Byte & Word Instructions (BWI)
Double-/Quad-word Instructions (DQI)
Vector Length Extensions (VLE)
Future Intel® Xeon Phi™ coprocessors (including Knights Landing)
Future Intel® Xeon® processors
Initial Many Core Instructions (IMCI)
Current Intel® Xeon Phi™ coprocessors (Knights Corner)
512-bit SIMD

For more information about Intel® AVX-512 instructions, check out James Reinders' initial and updated post for this topic.

Intel® AVX2/IMCI/AVX-512 differences

	Intel® Initial Many Core Instructions **IMCI**	Intel® Advanced Vector Extensions 2 **AVX2**	Intel® Advanced Vector Extensions 512 **AVX-512**
Introduction	2012	2013	2015
Products	Knights Corner	Haswell, Broadwell	Knights Landing, future Intel® Xeon® and Xeon® Phi™ products
Register file	SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x 16-bit mask registers	SP/DP/int32/int64 data types 16 x 256-bit SIMD registers No mask registers (instr. blending)	SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x (up to) 64-bit mask
ISA features	Not compatible with AVX*/SSE* No unaligned data support Embedded broadcast/cvt/swizzle MVEX encoding	Fully compatible with AVX/SSE* Unaligned data support (penalty) VEX encoding	Fully compatible with AVX*/SSE* Unaligned data support (penalty) Embedded broadcast/rounding EVEX encoding
Instruction features	Fused multiply-and-add (FMA) Partial gather/scatter Transcendental support	Fused multiply-and-add (FMA) Full gather	Fused multiply-and-add (FMA) Full gather/scatter Transcendental support (ERI only) Conflict detection instructions PFI/BWI/DQI/VLE (if applies)

Intel® AVX-512 is a major step in unifying the instruction set of Intel® MIC and Intel® Xeon® architecture

Side effects of SIMD vectorization

Scalar loop

```
float a[1024], b[1024], c[1024];
    …
for (int i = 0; i < 1024; i++)
    c[i] = a[i] + b[i];
```

Assumptions

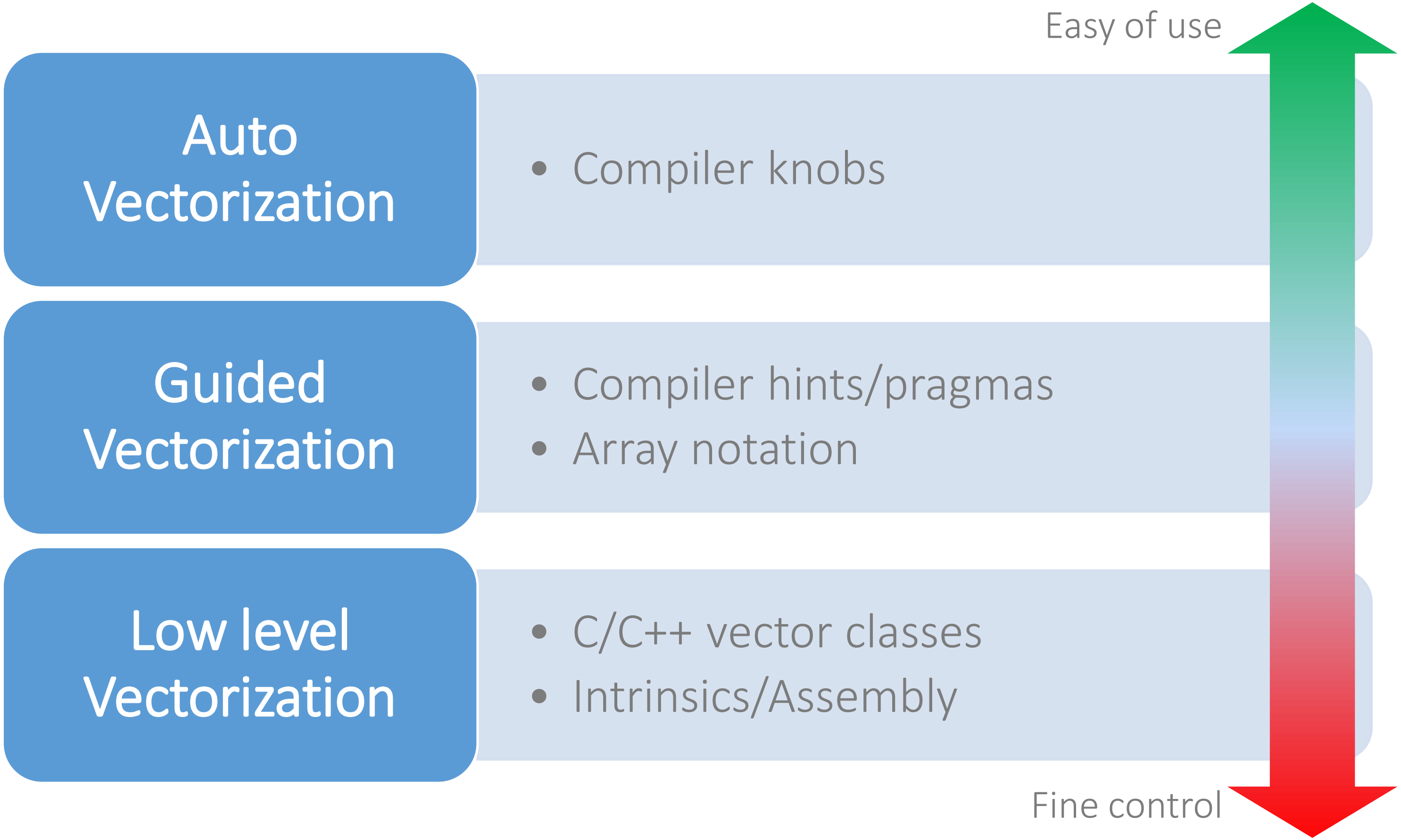
- 64-byte cache lines
- 32-byte (AVX2) and 64-byte (IMCI/AVX-512) SIMD registers
- 4-byte SP elements (float)
- No hardware prefetcher, no ld+op instructions
- Arrays are not cached

#Instructions	Scalar	AVX2 (256-bit)	IMCI AVX-512 (512-bit)
Loads (hit) to a[], b[]	960 + 960	64 + 64	0
Loads (miss) to a[], b[]	64 + 64	64 + 64	64 + 64
SP adds	1024	128	64
Stores to c[]	1024	128	64
Total (Reduction)	4096 (x1)	512 (x8)	256 (x16)

Observations

- Significant instruction count reduction (up to *vector-length*)
  - IPC decreases, but so does execution time as well
  - Usually translated into speedup
- Compute-bound codes turn into memory-bound codes
  - If code already was memory bound, no benefits at all (other than energy reduction)

Vectorization on Intel® compilers



Easy of use

Auto Vectorization

- Compiler knobs

Guided Vectorization

- Compiler hints/pragmas
- Array notation

Low level Vectorization

- C/C++ vector classes
- Intrinsics/Assembly

Fine control

Auto vectorization

Relies on the compiler for vectorization

- No source code changes
- Enabled with `-vec` compiler knob (default in `-O2` and `-O3` modes)

Option	Description
`-O0`	Disables all optimizations.
`-O1`	Enables optimizations for speed which are know to not cause code size increase.
`-O2/-O` (default)	Enables intra-file interprocedural optimizations for speed, including: • Vectorization • Loop unrolling
`-O3`	Performs O2 optimizations and enables more aggressive loop transformations such as: • Loop fusion • Block unroll-and-jam • Collapsing IF statements This option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. However, it might incur in slower code, numerical stability issues, and compilation time increase.

Compiler smart enough to apply loop transformations

- It will allow to vectorize more loops

Vectorization: target architecture options

On which architecture do we want to run our program?

Option	Description
`-mmic`	Builds an application that runs natively on Intel® MIC Architecture.
`-xfeature` `-xHost`	Tells the compiler which processor features it may target, referring to which instruction sets and optimizations it may generate (not available for Intel® Xeon Phi™ architecture). Values for *feature* are: • **`COMMON-AVX512`** (includes AVX512 FI and CDI instructions) • **`MIC-AVX512`** (includes AVX512 FI, CDI, PFI, and ERI instructions) • **`CORE-AVX512`** (includes AVX512 FI, CDI, BWI, DQI, and VLE instructions) • **`CORE-AVX2`** • `CORE-AVX-I` (including RDRND instruction) • `AVX` • `SSE4.2, SSE4.1` • `ATOM_SSE4.2, ATOM_SSSE3` (including MOVBE instruction) • `SSSE3`, `SSE3, SSE2` When using **`-xHost`**, the compiler will generate instructions for the highest instruction set available on the compilation host processor.
`-axfeature`	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. Values for *feature* are the same described for `-xfeature` option. Multiple features/paths possible, e.g.: `-axSSE2,AVX`. It also generates a baseline code path for the default case.

Vectorized code will be different depending on the chosen target architecture

Auto vectorization: not all loops will vectorize

Data dependencies between iterations

- Proven Read-after-Write data (i.e., loop carried) dependencies
- Assumed data dependencies
  - Aggressive optimizations (e.g., IPO) might help

RaW dependency

```
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

Vectorization won’t be efficient

- Compiler estimates how better the vectorized version will be
- Affected by data alignment, data layout, etc.

Inefficient vectorization

```
for (int i = 0; i < N; i++)
    a[c[i]] = b[d[i]];
```

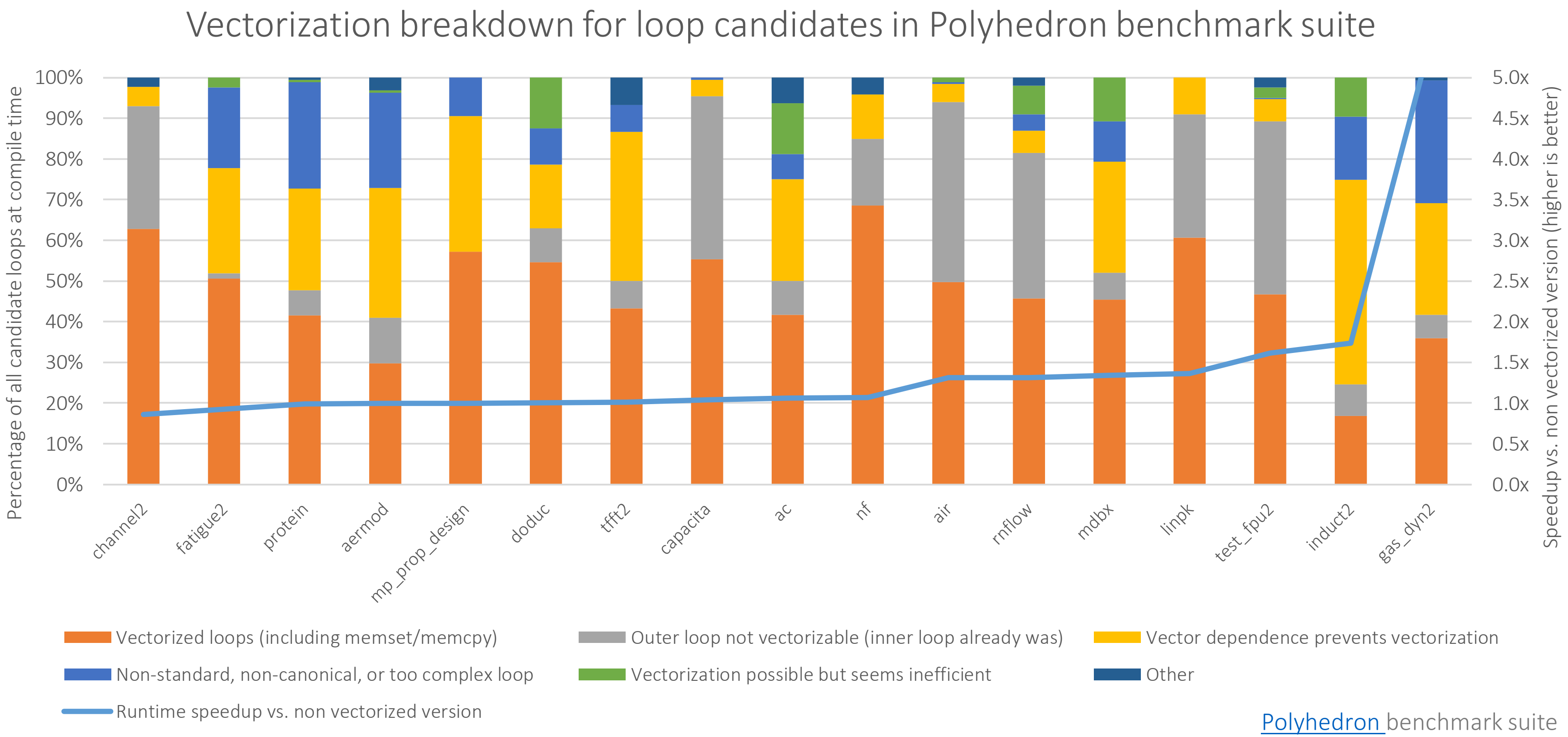
Unsupported loop structure

- While-loop, for-loop with unknown number of iterations
- Complex loops, unsupported data types, etc.
- (Some) function calls within loop bodies
  - Not the case for SVML functions

Function call within loop body

```
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```

Auto vectorization on Intel® compilers



Polyhedron benchmark suite

Intel® Xeon Phi™ 7120A, 61 cores x 4 threads

Intel® Fortran Compiler 15.0.1.14 [`-O3 -fp-model fast=2 -align array64byte -ipo -mmic`]

Validating vectorization success

Generate compiler report about optimizations

`-qopt-report[=n]`	Generate report (level [1..5], default 2)
`-qopt-report-file=<fname>`	Optimization report file (stderr, stdout also valid)
`-qopt-report-phase=<phase>`	Info about opt. phase:

`loop`	Loop nest optimizations
`par`	Auto-parallelization
`vec`	**Vectorization**
`openmp`	OpenMP
`offload`	Offload
`ipo`	Interprocedural optimizations
`pgo`	Profile Guided optimizations
`cg`	Code generation optimizations
`tcollect`	Trace analyzer (MPI) collection
`all`	All optimizations (default)

```
LOOP BEGIN at gas_dyn2.f90(193,11) inlined into gas_dyn2.f90(4326,31)
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 53
   remark #15477: vector loop cost: 14.870
   remark #15478: estimated potential speedup: 2.520
   remark #15479: lightweight vector operations: 19
   remark #15481: heavy-overhead vector operations: 1
   remark #15488: --- end vector loop cost summary ---
   remark #25456: Number of Array Refs Scalar Replaced In Loop: 1
   remark #25015: Estimate of max trip count of loop=4
LOOP END
```

Vectorized loop

```
LOOP BEGIN at gas_dyn2.f90(2346,15)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between IOLD line 376 and IOLD line 354
   remark #25015: Estimate of max trip count of loop=3000001
LOOP END
```

Non-vectorized loop

Guided vectorization: disambiguation hints

Get rid of assumed vector dependencies

Assume function arguments won’t be aliased

- C/C++: Compile with `-fargument-noalias`

C99 “restrict” keyword for pointers

- Compile with `-restrict` otherwise

```
void v_add(float *restrict c,
           float *restrict a,
           float *restrict b)
{
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Ignore assumed vector dependencies (compiler directive)

- C/C++: `#pragma ivdep`
- Fortran: `!dir$ ivdep`

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

Some Intel® compiler directives

Directive	Description
`distribute, distribute_point`	Instructs the compiler to prefer loop distribution at the location indicated.
`inline`	Instructs the compiler to inline the calls in question.
`ivdep`	Instructs the compiler to ignore assumed vector dependencies.
`loop_count`	Indicates the loop count is likely to be an integer.
`optimization_level`	Enables control of optimization for a specific function.
`parallel/noparallel`	Facilitates auto-parallelization of an immediately following loop; using keyword `always` forces the compiler to auto-parallelize; `noparallel` pragma prevents auto-parallelization.
`[no]unroll`	Instructs the compiler the number of times to unroll/not to unroll a loop
`[no]unroll_and_jam`	Prevents or instructs the compiler to partially unroll higher loops and jam the resulting loops back together.
`unused`	Describes variables that are unused (warnings not generated).
`[no]vector`	Specifies whether the loop should be vectorised. In case of forcing vectorization that should be according to the given clauses.

Guided vectorization: #pragma simd

Force loop vectorization ignoring **all** dependencies

- Additional clauses for specify reductions, etc.

SIMD loop

```
void v_add(float *c, float *a, float *b)
{
#pragma simd
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

SIMD function

```
__declspec(vector)
void v_add(float c, float a, float b)
{
    c = a + b;
}
    …
for (int i = 0; i < N; i++)
    v_add(C[i], A[i], B[i]);
```

Also supported in OpenMP

- Almost same functionality/syntax
  - Use `#pragma omp simd [clauses]` for SIMD loops
  - Use `#pragma omp declare simd [clauses]` for SIMD functions
- See OpenMP 4.0 specification for more information

Intel® compiler directives for vectorization

Directive	Clause	Description
vector	always	Force vectorization even when it might be not efficient.
	[un]aligned	Use [un]aligned data movement instructions for all array vector references.
	[non]temporal(*var1*[,…])	Do or do not generate non-temporal (streaming) stores for the given array variables. On Intel® MIC architecture, generates a cache-line-evict instruction when the store is known to be aligned.
	[no]vecreminder	Do (not) vectorize the remainder loop when the mail loop is vectorized.
	[no]mask_readwrite	Enables/disables memory speculation causing the generation of [non-]masked loads and stores within conditions.

simd	vectorlength(*n1*[,…])
	vectorlengthfor(*dtype*)
		Assume safe vectorization for the given vector length values or data type.

	private(*var1*[,…])
	firstprivate(*var1*[,…])
	lastprivate(*var1*[,…])
		Which variables are private to each iteration; *firstprivate*, initial value is broadcasted to all private instances; *lastprivate*, last value is copied out from the last instance.

	linear(*var1*:*step1*[,…])	Letting know the compiler that *var1* is incremented by *step1* on every iteration of the original loop.
	reduction(*oper*:*var1*[,…])	Which variables are reduction variables with a given operator.
	[no]assert	Warning or error when vectorization fails.
	[no]vecremainder	Do (not) vectorize the remainder loop when the mail loop is vectorized.

Explicit vectorization with array notation

Express high-level vector parallel array operations

- Valid notation in Fortran since Fortran 90
- Supported in C/C++ by Intel® compiler (Cilk^{TM} Plus) and GCC 4.9
  - Enabled by default on Intel® compiler, use `-fcilkplus` option on GCC
- No additional modifications to source code
- Most arithmetic and logic operations already overloaded
- Also built-in reducers for array sections

Vectorization becomes explicit

- C/C++ syntax: *`array-expression[lower-bound:length[:stride]]`*

Samples

```
a[:]        // All elements
a[2:6]      // Elements 2 to 7
a[:][5]     // Column 5
a[0:3:2]    // Elements 0,2,4
```

SIMD function invoked with array notation

```
__declspec(vector)
void v_add(float c, float a, float b)
{
    c = a + b;
}
    …
v_add(C[:], A[:], B[:]);
```

Improving vectorization: data layout

Vectorization more efficient with unit strides

- Non-unit strides will generate gather/scatter
- Unit strides also better for data locality
- Compiler might refuse to vectorize

AoS vs SoA

- Layout your data as Structure of Arrays (SoA)

Traverse matrices in the right direction

- C/C++: `a[i][:]`, Fortran: `a(:,i)`
- Loop interchange might help
  - Usually the compiler is smart enough to apply it
  - Check compiler optimization report

Array of Structures vs Structure of Arrays

```
// Array of Structures (AoS)
struct coordinate {
    float x, y, z;
} crd[N];
    …
for (int i = 0; i < N; i++)
    … = … f(crd[i].x, crd[i],y, crd[i].z);
```

Consecutive elements in memory →

x0 y0 z0 x1 y1 z1 … x(n-1) y(n-1) z(n-1)

```
// Structure of Arrays (AoS)
struct coordinate {
    float x[N], y[N], z[N];
} crd;
    …
for (int i = 0; i < N; i++)
    … = … f(crd.x[i], crd.y[i], crd.z[i]);
```

Consecutive elements in memory →

x0 x1 … x(n-1) y0 y1 … y(n-1) z0 z1 … z(n-1)

Improving vectorization: data alignment

Unaligned accesses might cause significant performance degradation

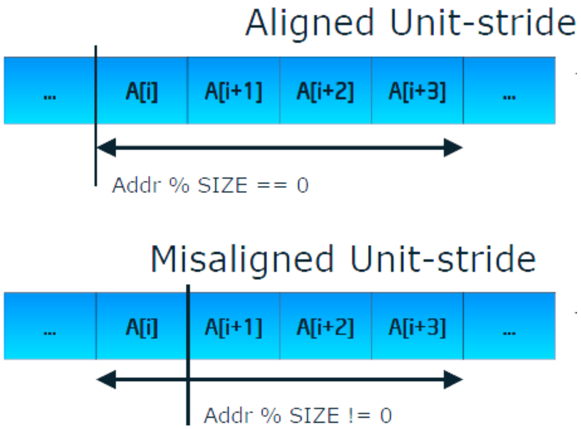
- Two instructions on current Intel® Xeon Phi^{TM} coprocessor
- Might cause “false sharing” problems
  - Consumer/producer thread on the same cache line

Alignment is generally unknown at compile time

- Every vector access is potentially an unaligned access
  - Vector access size = cache line size (64-byte)
- Compiler might “peel” a few loop iterations
  - In general, only one array can be aligned, though

When possible, we have to

- Align our data
- Tell the compiler data is aligned
  - Might not be always the case



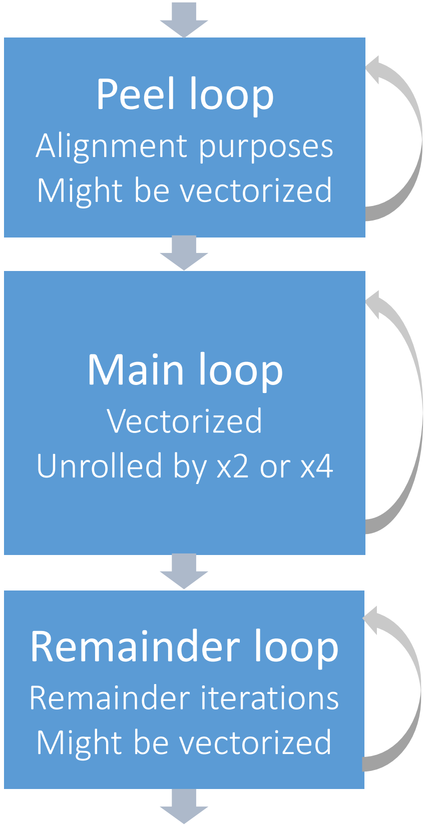
Improving vectorization: data alignment (cont’d)

How to…	Language	Syntax	Semantics
…align data	C/C++	void* _mm_malloc(int size, int n)	Allocate memory on heap aligned to *n* byte boundary.
	C/C++	int posix_memalign (void **p, size_t n, size_t size)	
	C/C++	__declspec(align(*n*)) *array*	Alignment for variable declarations.
	Fortran (not in common section)	!dir$ attributes align:*n*::*array*	
	Fortran (compiler option)	-align*n*byte	
…tell the compiler about it	C/C++	#pragma vector aligned	Vectorize assuming all array data accessed are aligned (may cause fault otherwise).
	Fortran	!dir$ vector aligned	
	C/C++	__assume_aligned(*array*, *n*)	Compiler may assume array is aligned to *n* byte boundary.
	Fortran	!dir$ assume_aligned *array*:*n*	

n=64 for Intel® Xeon Phi™ coprocessors, *n*=32 for AVX, *n*=16 for SSE

Padding might be necessary to guarantee aligned access to matrices

Vectorization with multi-version loops



Peel loop

Alignment purposes

Might be vectorized

Main loop

Vectorized

Unrolled by x2 or x4

Remainder loop

Remainder iterations

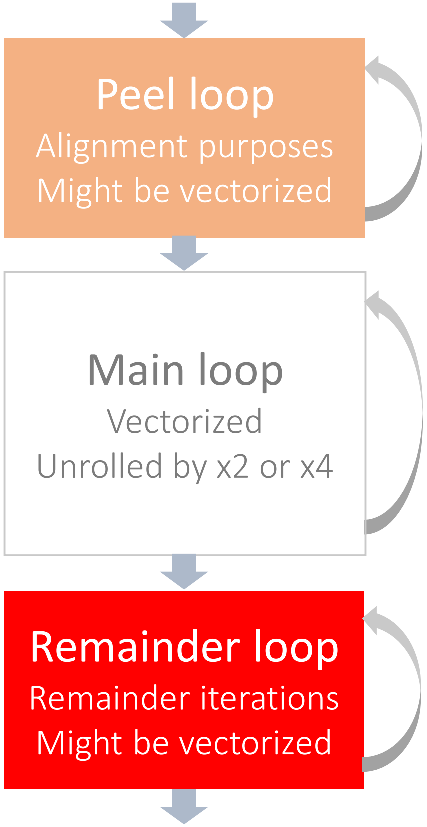
Might be vectorized

```
LOOP BEGIN at gas_dyn2.f90(2330,26)
<Peeled>
   remark #15389: vectorization support: reference AMAC1U has unaligned access
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at gas_dyn2.f90(2330,26)
   remark #25084: Preprocess Loopnests: Moving Out Store
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 8
   remark #15477: vector loop cost: 0.620
   remark #15478: estimated potential speedup: 15.890
   remark #15479: lightweight vector operations: 5
   remark #15488: --- end vector loop cost summary ---
   remark #25018: Total number of lines prefetched=4
   remark #25019: Number of spatial prefetches=4, dist=8
   remark #25021: Number of initial-value prefetches=6
LOOP END

LOOP BEGIN at gas_dyn2.f90(2330,26)
<Remainder>
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Improving vectorization: trip count hints



Vectorization can be seen as aggressive unrolling

- Main loop usually unrolled by x2 or x4
- Peel and remainder loop are vectorized with masks
- If trip count is low, vectorization might not be efficient
  - Remainder loop becomes the hotspot

Take a look at remainder loops

- Specify loop trip counts for efficient vectorization
  - `#pragma loop_count (n1,[ n2…])`
  - `#pragma loop_count min(n1), max(n2), avg(n3)`
- Consider padding (Intel® Xeon Phi^{TM} only)
  - Otherwise, remainder loops using gather/scatter loops
  - `-qopt-assume-safe-padding` to avoid it

Other considerations

Loop tiling/blocking to improve data locality

- Square tiles so elements can be reused

Use streaming loads/stores to save bandwidth

- `#pragma vector [non]temporal(list)`
- `-qopt-streaming-stores=[always|never|auto]`
- `-qopt-streaming-cache-evict[=n]` (Intel® MIC only)

Tune software prefetcher

- `-qopt-prefetch[=n]`
- `-qprefetch-distance=n1[,n2]` (Intel® MIC only)
- `#pragma [no]prefetch [clauses]` (Intel® MIC only)

Low level (explicit) vectorization

A.k.a “ninja programming”

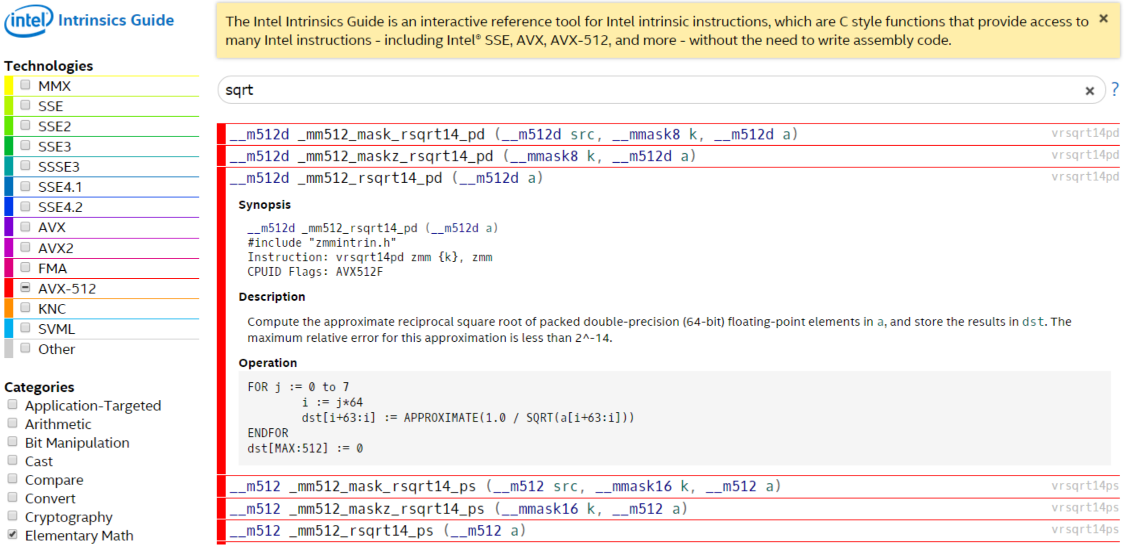
Vectorization relies on the programmer with some help from the compiler

Might be convenient for low level performance tuning of critical hotspots

Not portable among different SIMD architectures

SIMD C++ class	Intrinsics	Assembly

`#include <fvec.h>`
 	
`F32vec4 a,b,c;`
`a = b +c;`
	`#include <xmmintrin.h>`
	
	`__m128 a,b,c;`
	`a = _mm_add_ps(b,c);`
		`__m128 a,b,c;`
		`__asm {`
		`    movaps xmm0,b`
		`    movaps xmm1,c`
		`    addps xmm0,xmm1`
		`    movaps a, xmm0`
		`}`



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

sqrt × ?

```
__m512d _mm512_mask_rsqrt14_pd (__m512d src, __mmask8 k, __m512d a)      vrsqrt14pd
__m512d _mm512_maskz_rsqrt14_pd (__mmask8 k, __m512d a)                   vrsqrt14pd
__m512d _mm512_rsqrt14_pd (__m512d a)                                     vrsqrt14pd
```

Synopsis

```
__m512d _mm512_rsqrt14_pd (__m512d a)
#include "zmmintrin.h"
Instruction: vrsqrt14pd zmm {k}, zmm
CPUID Flags: AVX512F
```

Description

Compute the approximate reciprocal square root of packed double-precision (64-bit) floating-point elements in a, and store the results in dst. The maximum relative error for this approximation is less than 2^-14.

Operation

```
FOR j := 0 to 7
        i := j*64
        dst[i+63:i] := APPROXIMATE(1.0 / SQRT(a[i+63:i]))
ENDFOR
dst[MAX:512] := 0
```

```
__m512 _mm512_mask_rsqrt14_ps (__m512 src, __mmask16 k, __m512 a)         vrsqrt14ps
__m512 _mm512_maskz_rsqrt14_ps (__mmask16 k, __m512 a)                    vrsqrt14ps
__m512 _mm512_rsqrt14_ps (__m512 a)                                       vrsqrt14ps
```

How to get ready for Intel® AVX-512?

BKM: Start optimizing your application today for current generation of Intel® Xeon® processors and Intel® Xeon^{TM} Phi coprocessors

Tune your AVX-512 kernels on non-existing silicon

- Compile with latest compiler toolchains
  - Intel® compiler (v15.0): `-xCOMMON-AVX512`, `-xMIC-AVX512`, `-xCORE-AVX512`
  - GNU compiler (v4.9): `-mavx512f`, `-mavx512cd`, `-mavx512er`, `-mavx512pf`
- Run Intel® Software Development emulator (SDE)
  - Emulate (future) Intel® Architecture Instruction Set Extensions (e.g. Intel® MPX, …)
  - Tools available for detailed analysis
    - Instruction type histogram
    - Pointer/misalignment checker
  - Also possible to debug the application while emulated

Summary

Programmers are mostly responsible of exposing DLP (SIMD) parallelism

Intel® compilers provide sophisticated/flexible support for vectorization

- Auto, guided (assisted), and low-level (explicit) vectorization
- Based on OpenMP standards and specific directives
- Easily portable across different Intel® SIMD architectures

Fine-tuning of generated code is key to achieve the best performance

- Check whether code is actually vectorized
- Data layout, alignment, remainder loops, etc.

Get ready for Intel® AVX-512 by optimizing your application today on current generation of Intel® Xeon® processors and Intel® Xeon^{TM} Phi coprocessors

Online resources

Intel® Xeon Phi$^{\text{TM}}$

Developer portal	Programming guides, tools, trainings, case studies, etc.
Solutions catalog	Existing Intel® Xeon Phi$^{\text{TM}}$ solutions for known codes

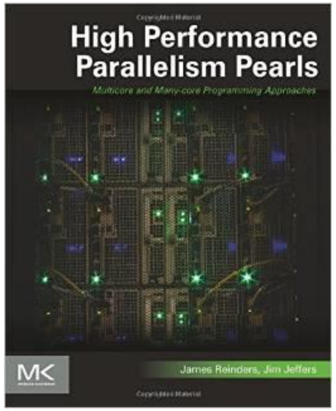
Intel® software development tools, performance tuning, etc.

Documentation library	All available documentation about Intel software
Learning lab	Learning material with Intel® Parallel Studio XE
Performance	Resources about performance tuning on Intel hardware
Forums	Public discussions about Intel SIMD, threading, ISAs, etc.

Other resources (white papers, benchmarks, case studies, etc.)

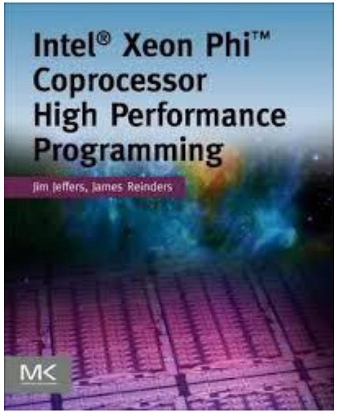
Go parallel	BKMs for Intel multi- and many-core architectures
Colfax research	Publications and material on parallel programming
Bayncore labs	Research and development activities (WIP)

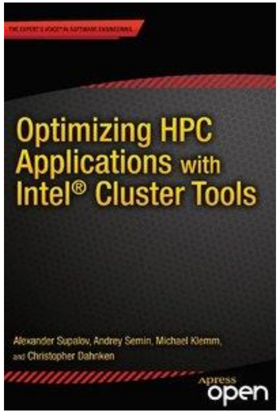
Recommended books


High Performance Parallelism Pearls
Multicore and Many-core Programming Approaches
MK
James Reinders, Jim Jeffers

High performance parallelism pearls: multi-core and many-core approaches, by James Reinders and Jim Jeffers, Morgan Kaufmann, 2014

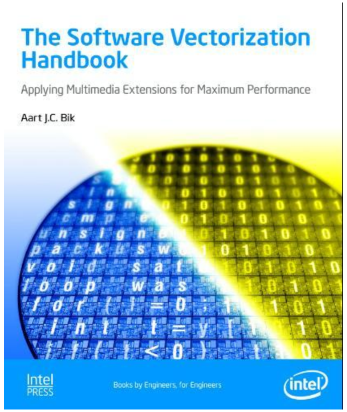
Intel® Xeon Phi^{TM} coprocessor high-performance programming, by Jim Jeffers and James Reinders, Morgan Kaufmann, 2013

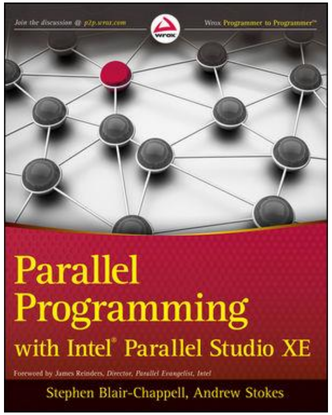

Intel® Xeon Phi™ Coprocessor High Performance Programming
Jim Jeffers, James Reinders
MK


Optimizing HPC Applications with Intel® Cluster Tools
Alexander Supalov, Andrey Semin, Michael Klemm, and Christopher Dahnken
Apress open

Optimizing HPC applications with Intel® cluster tools, by Alexander Supalov et al, Apress, 2014

The software optimization handbook, by Aart Bik, Intel® press, 2004


The Software Vectorization Handbook
Applying Multimedia Extensions for Maximum Performance
Aart J.C. Bik
Intel PRESS
Books by Engineers, for Engineers
intel


Wrox Programmer to Programmer™
Parallel Programming with Intel® Parallel Studio XE
Stephen Blair-Chappell, Andrew Stokes

Parallel programming with Intel® Parallel Studio XE, by Stephen Blair-Chappell and Andrew Stokes, Wrox press, 2012