Best practices for SIMD vectorization
Parallelism at single-thread core level

Manel Fernández

Intel HPC Software Workshop Series 2016
HPC Code Modernization for Intel® Xeon and Xeon Phi™
February 17th 2016, Barcelona
Exploiting the parallel universe
Three levels of parallelism supported by Intel hardware

Task Level Parallelism (TLP)
- Multi thread/task (MT) performance
- Exposed by programming models
- Execute tens/hundreds/thousands task concurrently

Data Level Parallelism (DLP)
- Single thread (ST) performance
- Exposed by tools and programming models
- Operate on 4/8/16 elements at a time

Instruction Level Parallelism (ILP)
- Single thread (ST) performance
- Automatically exposed by HW/tools
- Effectively limited to a few instructions

Programmers responsibility to expose DLP/TLP
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

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

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

Scalar loop

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

SIMD loop (4 elements)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>6.0</td>
<td>10.0</td>
</tr>
<tr>
<td>6.0</td>
<td>8.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>
```
Past, present, and future of Intel SIMD types

Current Intel® Xeon® processors
- 64-bit SIMD
- 128-bit SIMD
- 512-bit SIMD

Advanced Vector eXtensions (AVX)
- 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)

MultiMedia eXtensions (MMX)
- Initial Many Core Instructions (IMCI)

Streaming SIMD Extensions (SSE*)

AVX2
- 256-bit SIMD

AVX-512
- 512-bit SIMD

Current Intel® Xeon Phi™ coprocessors (Knights Corner)

Future Intel® Xeon Phi™ coprocessors (including Knights Landing)

Future Intel® Xeon® processors
- 64-bit SIMD
- 128-bit SIMD
- 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

<table>
<thead>
<tr>
<th></th>
<th>Intel® Initial Many Core Instructions</th>
<th>Intel® Advanced Vector Extensions 2</th>
<th>Intel® Advanced Vector Extensions 512</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IMCI</strong></td>
<td><strong>AVX2</strong></td>
<td><strong>AVX-512</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Introduction</strong></td>
<td>2012</td>
<td>2013</td>
<td>2015-2016</td>
</tr>
<tr>
<td><strong>Products</strong></td>
<td>Knights Corner</td>
<td>Haswell, Broadwell</td>
<td>Knights Landing, future Intel® Xeon® and Xeon® Phi™ products</td>
</tr>
<tr>
<td><strong>Register file</strong></td>
<td>SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x 16-bit mask registers</td>
<td>SP/DP/int32/int64 data types 16 x 256-bit SIMD registers No mask registers (instr. blending)</td>
<td>SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x (up to) 64-bit mask</td>
</tr>
<tr>
<td><strong>ISA features</strong></td>
<td>Not compatible with AVX*/SSE* No unaligned data support Embedded broadcast/cvt/swizzle MVEX encoding</td>
<td>Fully compatible with AVX/SSE* Unaligned data support (penalty) VEX encoding</td>
<td>Fully compatible with AVX*/SSE* Fast unaligned data support Embedded broadcast/rounding EVEX encoding</td>
</tr>
<tr>
<td><strong>Instruction features</strong></td>
<td>Fused multiply-and-add (FMA) Partial gather/scatter Transcendental support</td>
<td>Fused multiply-and-add (FMA) Full gather</td>
<td>Fused multiply-and-add (FMA) Full gather/scatter Transcendental support (ERI only) Conflict detection instructions PFI/BWI/DQI/VLE (if applies)</td>
</tr>
</tbody>
</table>

Intel® AVX-512 is a major step in unifying the instruction set of Intel® MIC and Intel® Xeon® architecture.
Side effects of SIMD vectorization

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

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)

<table>
<thead>
<tr>
<th>#Instructions</th>
<th>Scalar</th>
<th>AVX2 (256-bit)</th>
<th>IMCI AVX-512 (512-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads (hit) to a[], b[]</td>
<td>960 + 960</td>
<td>64 + 64</td>
<td>0</td>
</tr>
<tr>
<td>Loads (miss) to a[], b[]</td>
<td>64 + 64</td>
<td>64 + 64</td>
<td>64 + 64</td>
</tr>
<tr>
<td>SP adds</td>
<td>1024</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>Stores to c[]</td>
<td>1024</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>Total (Reduction)</td>
<td>4096 (x1)</td>
<td>512 (x8)</td>
<td>256 (x16)</td>
</tr>
</tbody>
</table>
Vectorization on Intel compilers

- Auto Vectorization
  - Libraries
  - Compiler knobs

- Guided Vectorization
  - Compiler hints/pragmas
  - Array notation

- Low level Vectorization
  - C/C++ vector classes
  - Intrinsics/Assembly
Rely on Intel® performance libraries
Highly efficient SIMD implementation of common functions for multiple Intel® processors

INTEL® INTEGRATED PERFORMANCE PRIMITIVES (INTEL® IPP)
A library of optimized building blocks for media and data applications. Take advantage of the unique capabilities of Intel processor families using optimized low-level APIs with significant emphasis on signal processing and certain media-focused applications, with cross-OS support and an internal dispatcher capable of selecting the prime optimization path.

INTEL® MATH KERNEL LIBRARY (INTEL® MKL)
The fastest and most used math library for Intel® and compatible processors. Harness the power of today’s processors—with increasing core counts, wider vector units, and more varied architectures. Includes highly vectorized and threaded linear algebra, fast Fourier Transforms, vector math, and statistics functions. Through a single API call, these functions automatically scale for future processor architectures by selecting the best code path for each.

INTEL® DATA ANALYTICS ACCELERATION LIBRARY (INTEL® DAAL)
Crunch more big data on the same node with Intel® DAAL for C++ and Java. The library provides highly optimized algorithmic building blocks to speed big data analytics performance on platforms from edge devices to servers. It encompasses data analysis stages (preprocessing, transformation, analysis, modeling, and decision making) for offline, streaming, and distributed analytics usages. Tight integration with popular data platforms (including Hadoop® and Spark®) enables highly efficient data access.

All libraries available at no cost with Community Licensing (Intel® support not included)
Auto vectorization
For C/C++ and Fortran

Relies on the compiler for vectorization of inner loops
- No source code changes
- Enabled with -vec compiler knob (default in -O2 and -O3 optimization levels)

<table>
<thead>
<tr>
<th>Opt. level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-00</td>
<td>Disables all optimizations.</td>
</tr>
<tr>
<td>-01</td>
<td>Enables optimizations for speed which are know to not cause code size increase.</td>
</tr>
</tbody>
</table>
| -02 / -0  (default) | Enables intra-file interprocedural optimizations for speed, including:  
  • Vectorization  
  • Loop unrolling |
| -03        | Performs -02 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. |
Auto vectorization strengths

Compiler is smart enough to vectorize not only simple loops
  • Loops over STL containers (including C++11 auto and ranged based loops)
  • Most reductions are recognized

```c++
double math_over_stl_vec (std::vector<double>& x)
{
    double sum{ 0.0 };  
    for (auto ix : x) 
        sum += sqrt(sin(ix)*sin(ix)+cos(ix)*cos(ix));
    return sum;
}
```

Transformations can be automatically applied to enable vectorization
  • Loop transformations (e.g., interchange, multi-versioning, if-conversion, etc).
  • Function inlining of user or vectorized library functions (e.g., SVML functions)
## Target architecture compiler options

On which architecture do we want to run our program?

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--mmic</code></td>
<td>Builds an application that runs natively on Intel® MIC Architecture.</td>
</tr>
<tr>
<td><code>--xfeature</code></td>
<td>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 <code>feature</code> are:</td>
</tr>
<tr>
<td><code>-xHost</code></td>
<td></td>
</tr>
<tr>
<td><code>--axfeature</code></td>
<td>Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. Values for <code>feature</code> are the same described for <code>--xfeature</code> option. Multiple features/paths possible, e.g.: <code>--aXSSE2</code>, AVX. It also generates a baseline code path for the default case.</td>
</tr>
</tbody>
</table>

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

Vectorization won’t be efficient
  • Compiler estimates how better the vectorized version will be
  • Affected by data alignment, data layout, etc.

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

---

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

**RaW dependency**

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

**Inefficient vectorization**

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

**Function call within loop body**
Auto vectorization on Intel compilers

Vectorization breakdown for loop candidates in Polyhedron benchmark suite

- Vectorized loops (including memset/memcpy)
- Non-standard, non-canonical, or too complex loop
- Outer loop not vectorizable (inner loop already was)
- Vectorization possible but seems inefficient
- Other

Polyhedron benchmark suite
Intel® Xeon Phi™ 7120A, 61 cores x 4 threads
Intel® Fortran Compiler 15.0.1.14 [-03 -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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>Loop nest optimizations</td>
</tr>
<tr>
<td>par</td>
<td>Auto-parallelization</td>
</tr>
<tr>
<td>vec</td>
<td>Vectorization</td>
</tr>
<tr>
<td>openmp</td>
<td>OpenMP</td>
</tr>
<tr>
<td>offload</td>
<td>Offload</td>
</tr>
<tr>
<td>ipo</td>
<td>Interprocedural optimizations</td>
</tr>
<tr>
<td>pgo</td>
<td>Profile Guided optimizations</td>
</tr>
<tr>
<td>cg</td>
<td>Code generation optimizations</td>
</tr>
<tr>
<td>tcollect</td>
<td>Trace analyzer (MPI) collection</td>
</tr>
<tr>
<td>all</td>
<td>All optimizations (default)</td>
</tr>
</tbody>
</table>

```plaintext
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
```

```plaintext
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
```

Vectorized loop

Non-vectorized loop
Guiding 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
- Or compile with `-restrict` knob

Ignore assumed vector dependencies with Intel-specific compiler directive
- **C/C++**: `#pragma ivdep`
- **Fortran**: `!dir$ ivdep`
## Some Intel-specific compiler directives

For **C/C++** and **Fortran**

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

C/C++ (also part of Cilk™ Plus)

- Enforcing **loop vectorization** ignoring all dependencies
  - `#pragma simd` in front of vectorizable loop
  - `_Simd` keyword right after `for/cilk_for` loop keyword
- Declaring **vectorized functions**
  - `__attribute__((vector))` / `__declspec(vector)` on Linux/Windows

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

Fortran

- `!dir$ simd`, `!dir$ attributes vector`

All directive idioms accept additional clauses (e.g., define reductions, etc.)
OpenMP SIMD directives

OpenMP 4.0 with similar functionality/syntax

- `#pragma omp simd` / `!$omp simd` for **SIMD loops**
- `#pragma omp declare simd` / `!$omp declare simd` for **SIMD functions**

```c
float sum = 0.0f;
float *p = a;
int step = 4;
#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
    sum += *p;
    p += step;
}
```

**SIMD loop**

```c
#pragma omp declare simd uniform(x0, max_iterations) linear(y0:1) simdlen(8)
color_t fractal::calc_one_pixel(int x0, int y0, int max_iterations)
{
    ...
}
```

**SIMD function**
## SIMD directives for loop vectorization

<table>
<thead>
<tr>
<th>Directive</th>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simd (OpenMP)</td>
<td><code>aligned(list[:step])</code></td>
<td>Specifies that list items have a given alignment (architecture alignment by default).</td>
</tr>
<tr>
<td>simd (OpenMP)</td>
<td><code>linear(list[:step])</code></td>
<td>The variable value is in relationship with the iteration number, ( x_i = x_{orig} + i \times \text{step} ).</td>
</tr>
<tr>
<td>simd (OpenMP)</td>
<td><code>safelen(length)</code></td>
<td>Maximum number of iterations that can run concurrently without breaking a dependence. In practice, maximum vector length.</td>
</tr>
<tr>
<td>simd (OpenMP)</td>
<td><code>collapse(n)</code> <code>firstprivate(list)</code> <code>lastprivate(list)</code> <code>private(list)</code> <code>reduction(oper:var1[,...])</code></td>
<td>Same as OpenMP existing clauses for work sharing.</td>
</tr>
<tr>
<td>simd (Cilk Plus)</td>
<td><code>vectorlength(n1[,...])</code> <code>vectorlengthfor(dtype)</code></td>
<td>Assume safe vectorization for the given vector length values or data type.</td>
</tr>
<tr>
<td>simd (Cilk Plus)</td>
<td><code>private(var1[,...])</code> <code>firstprivate(var1[,...])</code> <code>lastprivate(var1[,...])</code></td>
<td>Which variables are private to each iteration; <code>firstprivate</code>, initial value is broadcasted to all private instances; <code>lastprivate</code>, last value is copied out from the last instance.</td>
</tr>
<tr>
<td>simd (Cilk Plus)</td>
<td><code>linear(var1:step1[,...])</code></td>
<td>Letting know the compiler that <code>var1</code> is incremented by <code>step1</code> on every iteration of the original loop.</td>
</tr>
<tr>
<td>simd (Cilk Plus)</td>
<td><code>reduction(oper:var1[,...])</code></td>
<td>Which variables are reduction variables with a given operator.</td>
</tr>
<tr>
<td>simd (Cilk Plus)</td>
<td><code>[no]assert</code></td>
<td>Warning or error when vectorization fails.</td>
</tr>
<tr>
<td>simd (Cilk Plus)</td>
<td><code>[no]vecremainder</code></td>
<td>Do (not) vectorize the remainder loop when the main loop is vectorized.</td>
</tr>
</tbody>
</table>
Explicit vectorization with array notation
High level vector parallel operations on multi-dimensional array sections

**C/C++** (part of Cilk™ Plus)
- Syntax: `array-expression[[lower-bound]:[length][:stride]]`
- Most arithmetic and logic operations already overloaded
- Also built-in reducers for array sections
  - Syntax: `__sec_reduce_op(array-section)`, being `op` one of `max`, `min`, `add`, etc.

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

**Fortran**
- Syntax: `array-expression([lower-bound]:[upper-bound][:stride])`
- Valid notation in Fortran since Fortran 90
  - Most Fortran intrinsic functions naturally work on these expressions

```fortran
__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

Layout your data as Structure of Arrays (SoA)

- As opposite to Array of Structures (AoS)

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

```
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);
```

```
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]);
```
Intel® SIMD Data Layout Templates (SDLT)

**SDLT** stores data SOA internally while exposing AOS view to the program

- C++11 template library (no Intel compiler specific)
- Targets effective SIMD vectorization for performance
- SDLT's 1D-containers nearly identical as `std::vector`

```cpp
#include <sdlt/primitive.h>
#include <sdlt/soa1d_container.h>

struct Point3s { float x, y, z; }
SDLT_PRIMITIVE(Point3s, x, y, z)
sdlt::soa1d_container<Point3s> inputDataSet(count);
auto inputData = inputDataSet.const_access();
sdlt::soa1d_container<Point3s> outputDataSet(count);
auto outputData = outputDataSet.access();

#pragma omp simd
for(int i=0; i < count; ++i) {
    Point3s inputElement = inputData[i];
    Point3s result = …
    outputData[i] = result;
}
```
Improving vectorization: **data alignment**

Unaligned accesses might cause significant performance degradation
- Two instructions on current Intel® Xeon Phi™ 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)

<table>
<thead>
<tr>
<th>How to…</th>
<th>Language</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>…align data</td>
<td>C/C++</td>
<td><code>void* _mm_malloc(int size, int n)</code></td>
<td>Allocate memory on heap aligned to n byte boundary.</td>
</tr>
<tr>
<td></td>
<td>C/C++</td>
<td><code>int posix_memalign(void **p, size_t n, size_t size)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C/C++</td>
<td><code>__declspec(align(n)) array</code> (Windows) <code>__attribute__(align(n)) array</code> (Linux)</td>
<td>Alignment for variable declarations.</td>
</tr>
<tr>
<td></td>
<td>C++11</td>
<td>`alignas(expression</td>
<td>type)`</td>
</tr>
<tr>
<td></td>
<td>Fortran (not in common section)</td>
<td><code>!dir$ attributes align:n::array</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fortran (compiler option)</td>
<td><code>-alignnbyte</code></td>
<td></td>
</tr>
<tr>
<td>…tell the compiler about it</td>
<td>C/C++</td>
<td><code>#pragma vector aligned</code></td>
<td>Vectorize assuming all array data accessed are aligned (may cause fault otherwise).</td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
<td><code>!dir$ vector aligned</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C/C++</td>
<td><code>__assume_aligned(array, n)</code></td>
<td>Compiler may assume array is aligned to n byte boundary.</td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
<td><code>!dir$ assume_aligned array:n</code></td>
<td></td>
</tr>
</tbody>
</table>

Padding might be necessary to guarantee aligned access to matrices

\(n=64\) for Intel® Xeon Phi™ coprocessors, \(n=32\) for AVX, \(n=16\) for SSE
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)
<Peel>

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 #15481: --- begin 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/!dir$ loop_count (n1,[n2...])`
  • `#pragma/!dir$ loop_count min(n1),max(n2),avg(n3)`
• Consider `safe padding` option (Intel® Xeon Phi™ only)
  • Otherwise, remainder loops using gather/scatter loops
  • `-qopt-assume-safe-padding` to avoid it
Other considerations

Consider outer loop vectorization with SIMD directives
  • Inner loop might not have enough iterations

Loop tiling/blocking to improve data locality
  • Vectorization usually increases memory bandwidth requirements
  • Square tiles so elements can be reused

Use streaming stores to save bandwidth
  • `#pragma/!dir$ 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/!dir$ [no]prefetch [clauses]` (Intel® MIC only)
Low level (explicit) vectorization
A.k.a “ninja programming” (C/C++ only)

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++ classes

```
#include <fvec.h>
F32vec4 a, b, c;
a = b + c;
```

Intrinsics

```
#include <xmmintrin.h>
__m128 a, b, c;
__asm {
    movaps xmm0, b
    movaps xmm1, c
    addps xmm0, xmm1
    movaps a, xmm0
}
```

Inline assembly

```
__asm {
    movaps xmm0, b
    movaps xmm1, c
    addps xmm0, xmm1
    movaps a, xmm0
}
```
Survey analysis

- See what prevents vectorization
- Detect vectorization issues
- Source/assembly integration
- Optimization reports
- Automatic recommendations

Trip-count analysis

- How many iterations in a loop
- Quantify peel/main/remainder

Deeper analyses

- Correctness analysis to see if a loop can be safely vectorized
- Memory access pattern (MAP) to figure out actual vectorization stride

Complete tutorial in latest Intel’s magazine “The Parallel Universe” (Issue 22)
How to get ready for Intel® AVX-512?

Start optimizing your application today for current generation of Intel® Xeon® processors and Intel® Xeon™ Phi coprocessors - and/or -

Compile with latest compiler toolchains
• Intel compiler (v15.0+): -xCOMMON-AVX512, -xMIC-AVX512, -xCORE-AVX512
• GNU compiler (v4.9+): -mavx512f, -mavx512cd, -mavx512er, -mavx512pf

Tune your AVX-512 kernels on non-existing silicon
• Run your kernels on top of 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 being emulated
Summary

Programmers are mostly responsible of exposing DLP (SIMD) parallelism
• Key to fully exploit ST performance on Intel® multi-and many-core architectures

Intel compilers provide sophisticated/flexible support for vectorization
• Auto, guided (assisted), and low-level (explicit) vectorization
• Based on standards (OpenMP, Cilk™ Plus) and Intel-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 and how
  • Data layout, alignment, remainder loops, etc.
  • Use Intel® Parallel Studio XE 2016 to help vectorize your code

Get ready for “future” Intel® AVX-512 by optimizing your application on today’s Intel® Xeon® processors and Intel® Xeon™ Phi coprocessors
Online resources

Intel® software development tools, performance tuning, etc.

- **Documentation library**
  All available documentation about Intel software
- **HPC webinars**
  Free technical webinars about HPC on Intel platforms
- **Modern code**
  Intel resources about code modernization
- **Forums**
  Public discussions about Intel SIMD, threading, ISAs, etc.

Intel® Xeon Phi™ resources

- **Developer portal**
  Programming guides, tools, trainings, case studies, etc.
- **Solutions catalog**
  Existing Intel® Xeon Phi™ solutions for known codes

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


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

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