Vectorization for Intel® C++ & Fortran Compiler

Presenter: Georg Zitzlsberger

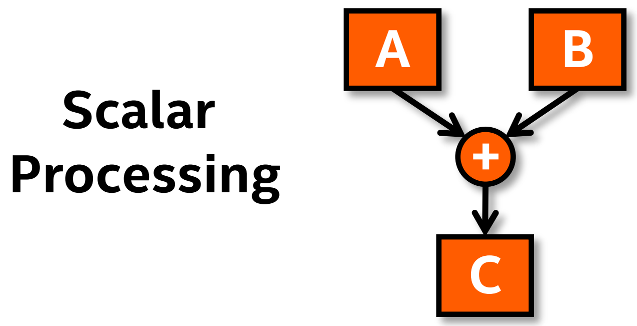
Date: 10-07-2015

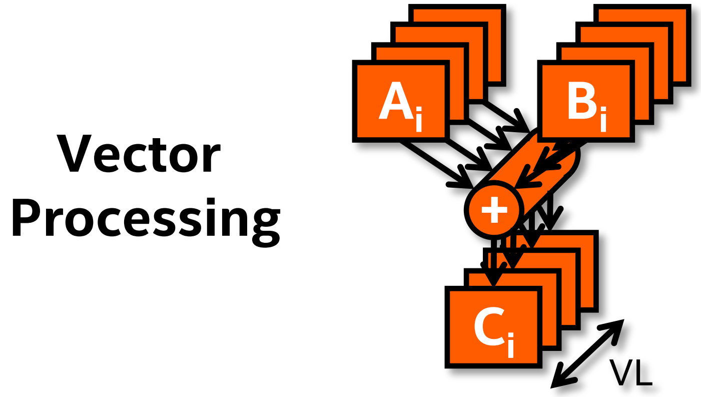
Agenda

- **Introduction to SIMD for Intel® Architecture**
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- Summary

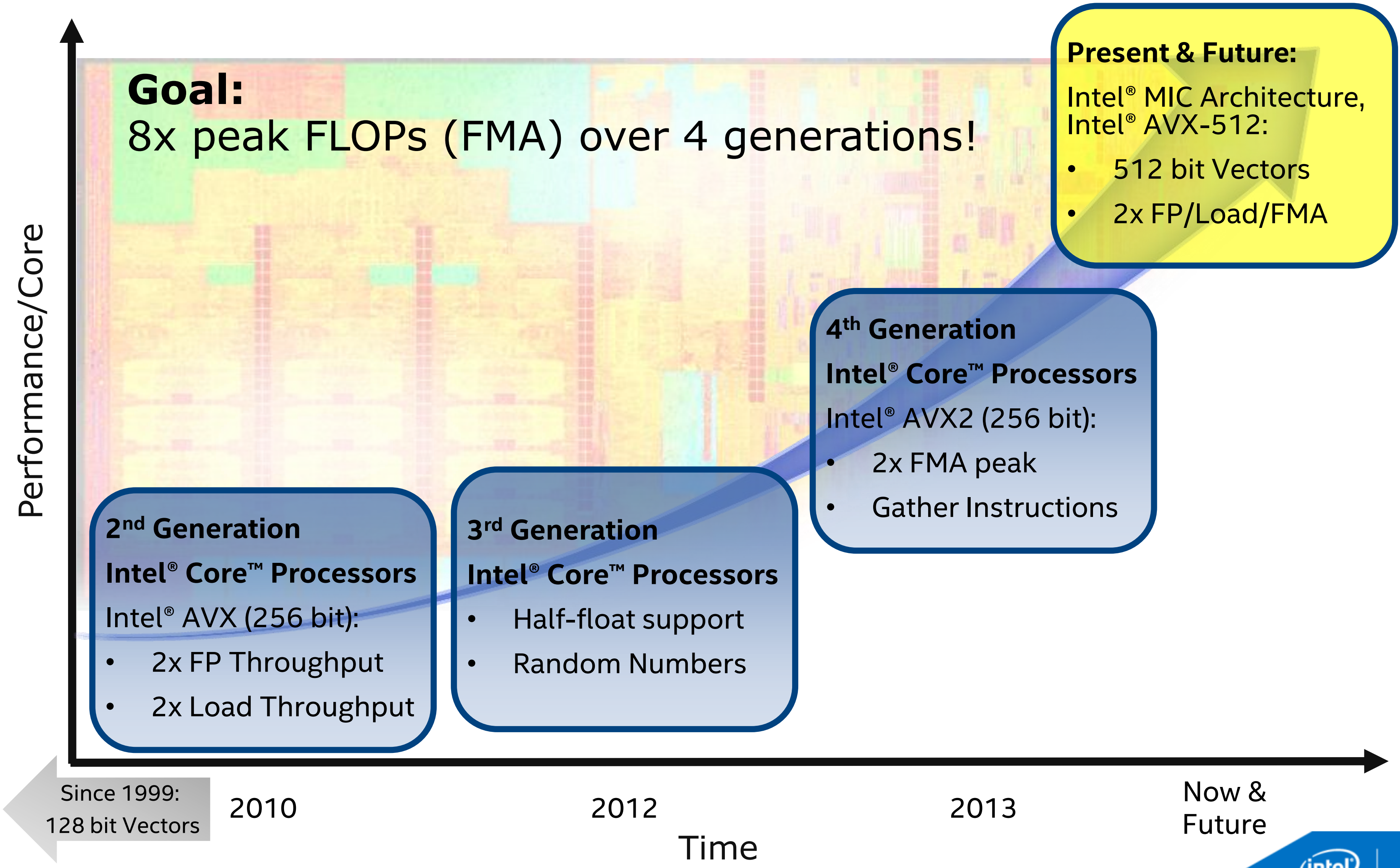
Vectorization

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD):
  - Processing vector with a single operation
  - Provides data level parallelism (DLP)
  - Because of DLP more efficient than scalar processing
- **Vector:**
  - Consists of more than one element
  - Elements are of same scalar data types (e.g. floats, integers, …)
- **Vector length (VL):** Elements of the vector





Evolution of SIMD for Intel Processors



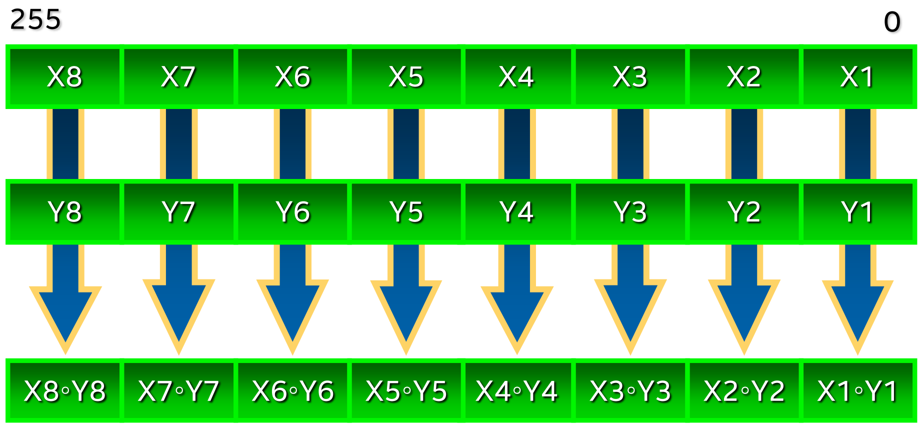
2nd Generation
Intel® Core™ Processors
Intel® AVX (256 bit):
- 2x FP Throughput
- 2x Load Throughput

3rd Generation
Intel® Core™ Processors
- Half-float support
- Random Numbers

4th Generation
Intel® Core™ Processors
Intel® AVX2 (256 bit):
- 2x FMA peak
- Gather Instructions

Present & Future:
Intel® MIC Architecture, Intel® AVX-512:
- 512 bit Vectors
- 2x FP/Load/FMA

SIMD Types for Intel® Architecture II



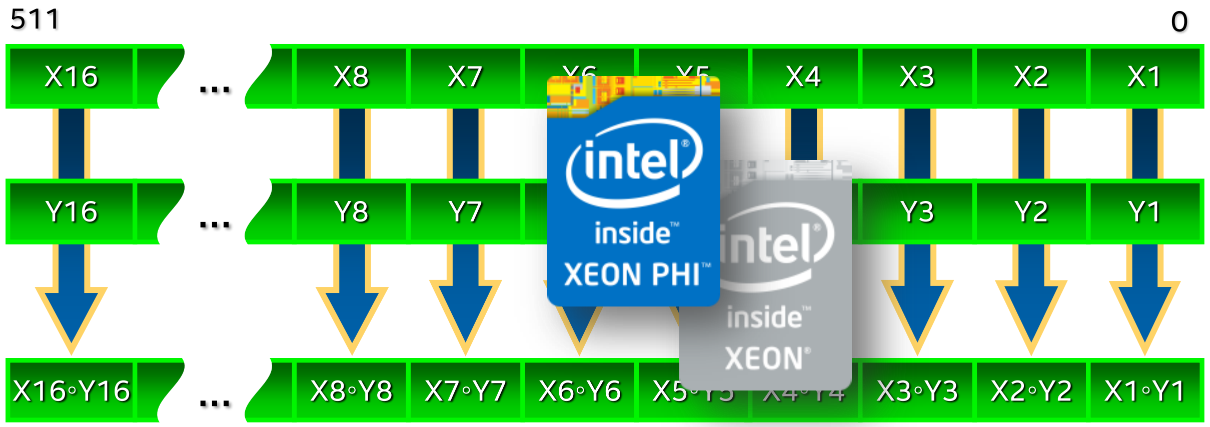
AVX

Vector size: **256 bit**

Data types:

- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 4, 8, 16, 32



511
0
X16 … X8 X7 X6 X5 X4 X3 X2 X1
Y16 … Y8 Y7 Y3 Y2 Y1
X16◦Y16 … X8◦Y8 X7◦Y7 X6◦Y6 X5◦Y5 X4◦Y4 X3◦Y3 X2◦Y2 X1◦Y1
intel inside XEON PHI
intel inside XEON

Intel® AVX-512 &
Intel® MIC Architecture

Vector size: **512 bit**

Data types:

- 8, 16, 32, 64 bit integer
- 32 and 64 bit float

VL: 8, 16, 32, 64

Illustrations: Xi, Yi & results 32 bit integer

AVX Generations

- 2010: Initial version of **Intel® AVX** in 2nd generation Intel® Core™ processors:
  - Double register size of SSE, twice as much vector elements (2x peak FLOP)
  - Support for single- & double-precision FP
  - Load/Store size increased from 128 bit to 256 bit!
- 2012: 3rd generation Intel® Core™ processor improvements:
  - Non-deterministic random number generator
  - Half-precision conversion (from/to single-precision)
- 2013: 4th generation Intel® Core™ processor improvements:
  - **Intel® AVX2** (with integer support)
  - FMA (2x more peak FLOP with same vector length)
  - Gather non adjacent memory locations
- Future: **Intel® AVX-512**

Intel® AVX2

- Basically same as Intel® AVX with following additions:
  - **Doubles** width of **integer vector** instructions to 256 bits
  - Floating point fused multiply add (**FMA**)

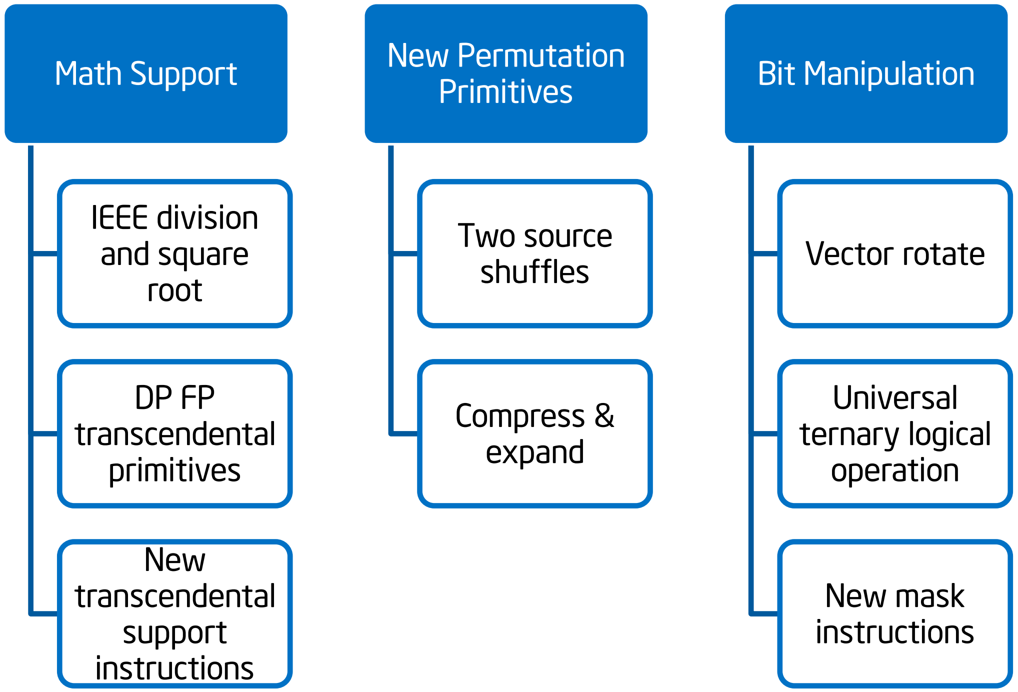
Processor Family	Instruction Set	Single Precision FLOPs Per Clock	Double Precision FLOPs Per Clock	
Pre 2nd generation Intel® Core™ Processors	SSE 4.2	8	4	
2nd and 3rd generation Intel® Core™ Processors	AVX	16	8	2x
4th generation Intel® Core™ Processors	**AVX2**	**32**	**16**	4x

  - Bit Manipulation Instructions (BMI)
  - Gather instructions (scatter for the future)
  - Any-to-any permutes
  - Vector-vector shifts

Intel® AVX-512 Features I

Different versions of Intel® AVX-512:

- Intel® AVX-512 **Foundation**:
  - Extension of AVX known instruction sets including mask registers
  - Available in all products supporting Intel® AVX-512


Math Support
IEEE division and square root
DP FP transcendental primitives
New transcendental support instructions
New Permutation Primitives
Two source shuffles
Compress & expand
Bit Manipulation
Vector rotate
Universal ternary logical operation
New mask instructions

Intel® AVX-512 Features II

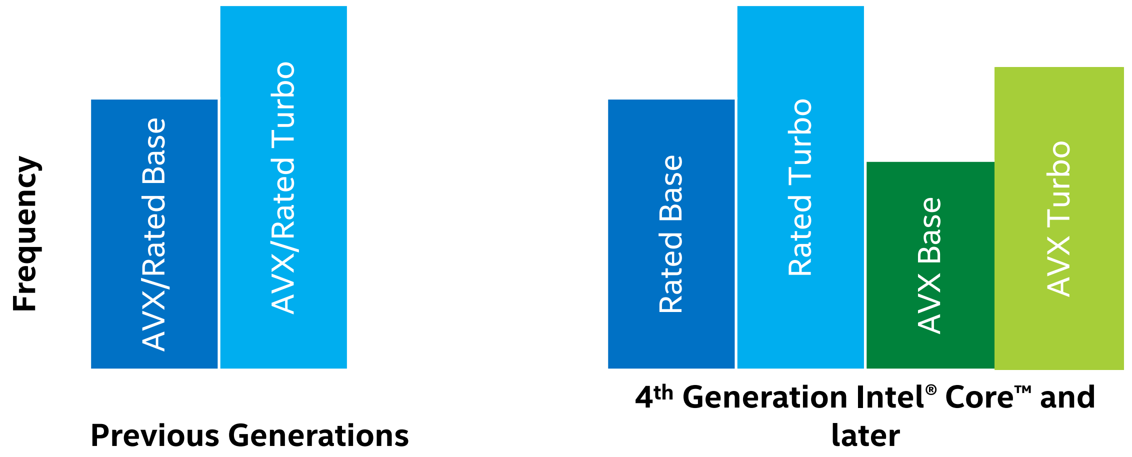
- Intel® AVX-512 **Vector Length Extension**:
  - Freely select the vector length (512 bit, 256 bit and 128 bit)
  - Orthogonal extension but planned for future Intel® Xeon® processors only
- Intel® AVX-512 **Byte/Word** and **Doubleword/Quadword**:
  - Two groups:
    - 8 and 16 bit integers
    - 32 and 64 bit integers & FP
  - Planned for future Intel® Xeon® processors
- Intel® AVX-512 **Conflict Detection**:
  - Check identical values inside a vector (for 32 or 64 bit integers)
  - Used for finding colliding indexes (32 or 64 bit) before a gather-operation-scatter sequence
  - Likely to be available in future for both Intel® Xeon Phi™ coprocessors and Intel® Xeon® processors

Intel® AVX-512 Features III

- Intel® AVX-512 **Exponential & Reciprocal Instructions**:
  - Higher accuracy (28 bit) with HW based sqrt, reciprocal and exp function
  - Likely only for future Intel® Xeon Phi™ coprocessors
- Intel® AVX-512 **Prefetch Instructions**:
  - Manage data streams for higher throughput (incl. gather & scatter)
  - Likely only for future Intel® Xeon Phi™ coprocessors
- More here: https://software.intel.com/en-us/blogs/additional-avx-512-instructions

Intel® Turbo Boost Technology and Intel® AVX*

- Amount of turbo frequency achieved depends on:
  Type of workload, **number** of **active cores**, estimated **current & power consumption**, and processor **temperature**
- Due to workload dependency, separate AVX base & turbo frequencies will be defined for 4th generation Intel® Core™ and Xeon® processors and later



* Intel® AVX refers to Intel® AVX, Intel® AVX2 or Intel® AVX-512

Agenda

- Introduction to SIMD for Intel® Architecture
- **Compiler & Vectorization**
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- Summary

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

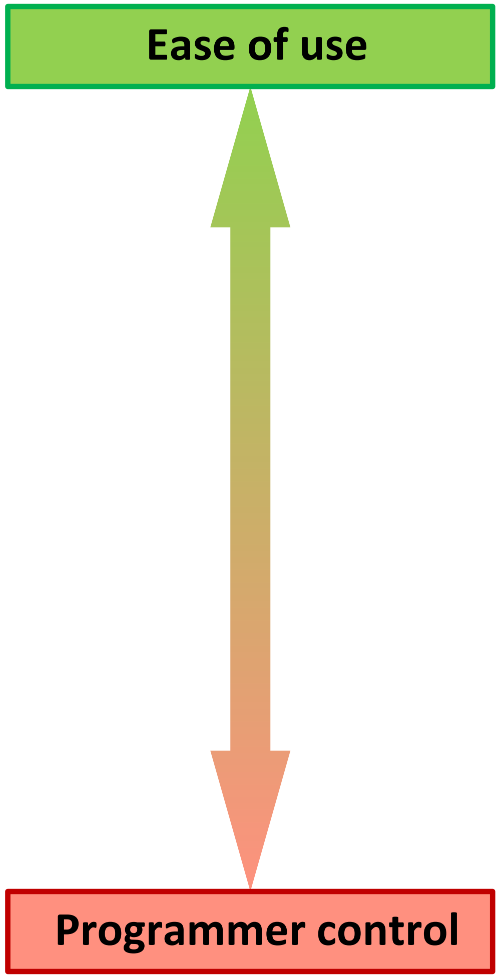
Compiler:
Auto-vectorization hints (`#pragma vector`, …)

Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus

SIMD intrinsic class
(e.g.: `F32vec`, `F64vec`, …)

Vector intrinsic
(e.g.: `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Assembler code
(e.g.: `[v]addps`, `[v]addss`, …)



Auto-vectorization of Intel Compilers

```
void add(A, B, C)
double A[1000]; double B[1000]; double C[1000];
{
  int i;
  for (i = 0; i < 1000; i++)
    C[i] = A[i] + B[i];
}
```

```
subroutine add(A, B, C)
  real*8 A(1000), B(1000), C(1000)
  do i = 1, 1000
      C(i) = A(i) + B(i)
  end do
end
```

Intel® AVX

```
..B1.2:
  vmovupd    (%rsp,%rax,8), %ymm0
  vmovupd    32(%rsp,%rax,8), %ymm2
  vmovupd    64(%rsp,%rax,8), %ymm4
  vmovupd    96(%rsp,%rax,8), %ymm6
  vaddpd     8032(%rsp,%rax,8), %ymm2, %ymm3
  vaddpd     8000(%rsp,%rax,8), %ymm0, %ymm1
  vaddpd     8064(%rsp,%rax,8), %ymm4, %ymm5
  vaddpd     8096(%rsp,%rax,8), %ymm6, %ymm7
  vmovupd    %ymm1, 16000(%rsp,%rax,8)
  vmovupd    %ymm3, 16032(%rsp,%rax,8)
  vmovupd    %ymm5, 16064(%rsp,%rax,8)
  vmovupd    %ymm7, 16096(%rsp,%rax,8)
  addq       $16, %rax
  cmpq       $992, %rax
  jb         ..B1.2
  ...
```

Intel® SSE4.2

```
..B1.2:
  movaps     (%rsp,%rax,8), %xmm0
  movaps     16(%rsp,%rax,8), %xmm1
  movaps     32(%rsp,%rax,8), %xmm2
  movaps     48(%rsp,%rax,8), %xmm3
  addpd      8000(%rsp,%rax,8), %xmm0
  addpd      8016(%rsp,%rax,8), %xmm1
  addpd      8032(%rsp,%rax,8), %xmm2
  addpd      8048(%rsp,%rax,8), %xmm3
  movaps     %xmm0, 16000(%rsp,%rax,8)
  movaps     %xmm1, 16016(%rsp,%rax,8)
  movaps     %xmm2, 16032(%rsp,%rax,8)
  movaps     %xmm3, 16048(%rsp,%rax,8)
  addq       $8, %rax
  cmpq       $1000, %rax
  jb         ..B1.2
  ...
```

SIMD Features I

Support of SIMD extensions for Intel processors:

SIMD Feature	Description
ATOM_SSE4.2	May generate MOVBE instructions for Intel processors (depending on setting of `-minstruction` or `/Qinstruction`). May also generate Intel® SSE4.2, SSE3, SSE2 and SSE instructions for Intel processors. Optimizes for Intel® Atom™ processors that support Intel® SSE4.2 and MOVBE instructions.
SSE4.2	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.
SSE4.1	May generate Intel® SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.
ATOM_SSSE3 deprecated: SSE3_ATOM & SSSE3_ATOM	May generate MOVBE instructions for Intel processors (depending on setting of `-minstruction` or `/Qinstruction`). May also generate Intel® SSE3, SSE2, SSE and Intel SSSE3 instructions for Intel processors. Optimizes for Intel® Atom™ processors that support Intel® SSE3 and MOVBE instructions.
SSSE3	May generate Intel® SSE3, SSE2, SSE and Intel SSSE3.
SSE3	May generate Intel® SSE3, SSE2 and SSE.
SSE2	May generate Intel® SSE2 and SSE.

SIMD Features II

Support of SIMD extensions for Intel processors (cont'd):

SIMD Feature	Description
MIC-AVX512	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Exponential and Reciprocal instructions, Intel® AVX-512 Prefetch instructions for Intel® processors, and the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3 instructions.
CORE-AVX-I	May generate Intel® Advanced Vector Extensions (Intel® AVX), including instructions in 3rd generation Intel® Core™ processors, Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.
AVX	May generate Intel® Advanced Vector Extensions (Intel® AVX), SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.

Basic Vectorization Switches I

- Linux*, OS X*: **`-x<feature>`**, Windows*: **`/Qx<feature>`**
  - Might enable Intel processor specific optimizations
  - Processor-check added to "main" routine:
    Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message

- Linux*, OS X*: **`-ax<features>`**, Windows*: **`/Qax<features>`**
  - Multiple code paths: baseline and optimized/processor-specific
  - Optimized code paths for Intel processors defined by **`<features>`**
  - Multiple SIMD features/paths possible, e.g.: **`-axSSE2,AVX`**
  - Baseline code path defaults to **`–msse2`** (**`/arch:sse2`**)
  - The baseline code path can be modified by **`–m<feature>`** or **`–x<feature>`** (**`/arch:<feature>`** or **`/Qx<feature>`**)

Basic Vectorization Switches II

- Linux*, OS X*: **`-m<feature>`**, Windows*: **`/arch:<feature>`**
  - Neither check nor specific optimizations for Intel processors:
    Application optimized for both Intel and non-Intel processors for selected SIMD feature
  - Missing check can cause application to fail in case extension not available
- Default for Linux*: **`-msse2`**, Windows*: **`/arch:sse2`**:
  - Activated implicitly
  - Implies the need for a target processor with at least Intel® SSE2
- Default for OS X*: **`-msse3`** (IA-32), **`-mssse3`** (Intel® 64)

- For 32 bit compilation, **`-mia32`** (**`/arch:ia32`**) can be used in case target processor does not support Intel® SSE2 (e.g. Intel® Pentium® 3 or older)

Basic Vectorization Switches III

- Special switch for Linux*, OS X*: **`-xHost`**, Windows*: **`/QxHost`**
  - Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
  - Code only executes on processors with same SIMD feature or later as on build host
  - As for **`-x<feature>`** or **`/Qx<feature>`**, if “main” routine is built with **`-xHost`** or **`/QxHost`** the final executable only runs on Intel processors

Control Vectorization I

- Disable vectorization:
  - Globally via switch:
    Linux*, OS X*: **`-no-vec`**, Windows*: **`/Qvec-`**
  - For a single loop:
    C/C++: **`#pragma novector`**, Fortran: **`!DIR$ NOVECTOR`**
  - Compiler still can use some SIMD features
- Using vectorization:
  - Globally via switch (default for optimization level 2 and higher):
    Linux*, OS X*: **`-vec`**, Windows*: **`/Qvec`**
  - Enforce for a single loop (override compiler efficiency heuristic) if semantically correct:
    C/C++: **`#pragma vector always`**, Fortran: **`!DIR$ VECTOR ALWAYS`**
  - Influence efficiency heuristics threshold:
    Linux*, OS X*: **`-vec-threshold[n]`**
    Windows*: **`/Qvec-threshold[[:]n]`**
    `n`: `100` (default; only if profitable) … `0` (always)

Control Vectorization II

- Verify vectorization:
  - Globally:
    Linux*, OS X*: `-opt-repot`, Windows*: `/Qopt-report`
  - Abort compilation if loop cannot be vectorized:
    C/C++: `#pragma vector always assert`
    Fortran: `!DIR$ VECTOR ALWAYS ASSERT`
- Advanced:
  - Ignore vector dependencies (IVDEP):
    C/C++: `#pragma ivdep`
    Fortran: `!DIR$ IVDEP`
  - "Enforce" vectorization:
    C/C++: `#pragma simd` or `#pragma omp simd`
    Fortran: `!DIR$ SIMD` or `!$OMP SIMD`

    When used, vectorization can only be turned off with:
    Linux*, OS X*: `-no-vec -no-simd -qno-openmp-simd`
    Windows*: `/Qvec- /Qsimd- /Qopenmp-simd-`

Agenda

- Introduction to SIMD for Intel® Architecture
- Compiler & Vectorization
- **Validating Vectorization Success**
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- Summary

Validating Vectorization Success I

- **Assembler code inspection (Linux*, OS X*: `-S`, Windows*: `/Fa`):**
  - Most reliable way and gives all details of course
  - Check for scalar/packed or (E)VEX encoded instructions:
    Assembler listing contains source line numbers for easier navigation
- **Using Intel® VTune™ Amplifier:**
  - Different events can be selected to measure use of vector units, e.g.
    `FP_COMP_OPS_EXE.SSE_PACKED_[SINGLE|DOUBLE]`
  - For Intel® MIC Architecture: Use metric **Vectorization Intensity**
- **Difference method:**
  1. Compile and benchmark with `-no-vec -no-simd -qno-openmp-simd` or `/Qvec- /Qsimd- /Qopenmp-simd-`, or on a loop by loop basis via `#pragma novector` or `!DIR$ NOVECTOR`
  2. Compile and benchmark with selected SIMD feature
  3. Compare runtime differences

Validating Vectorization Success II

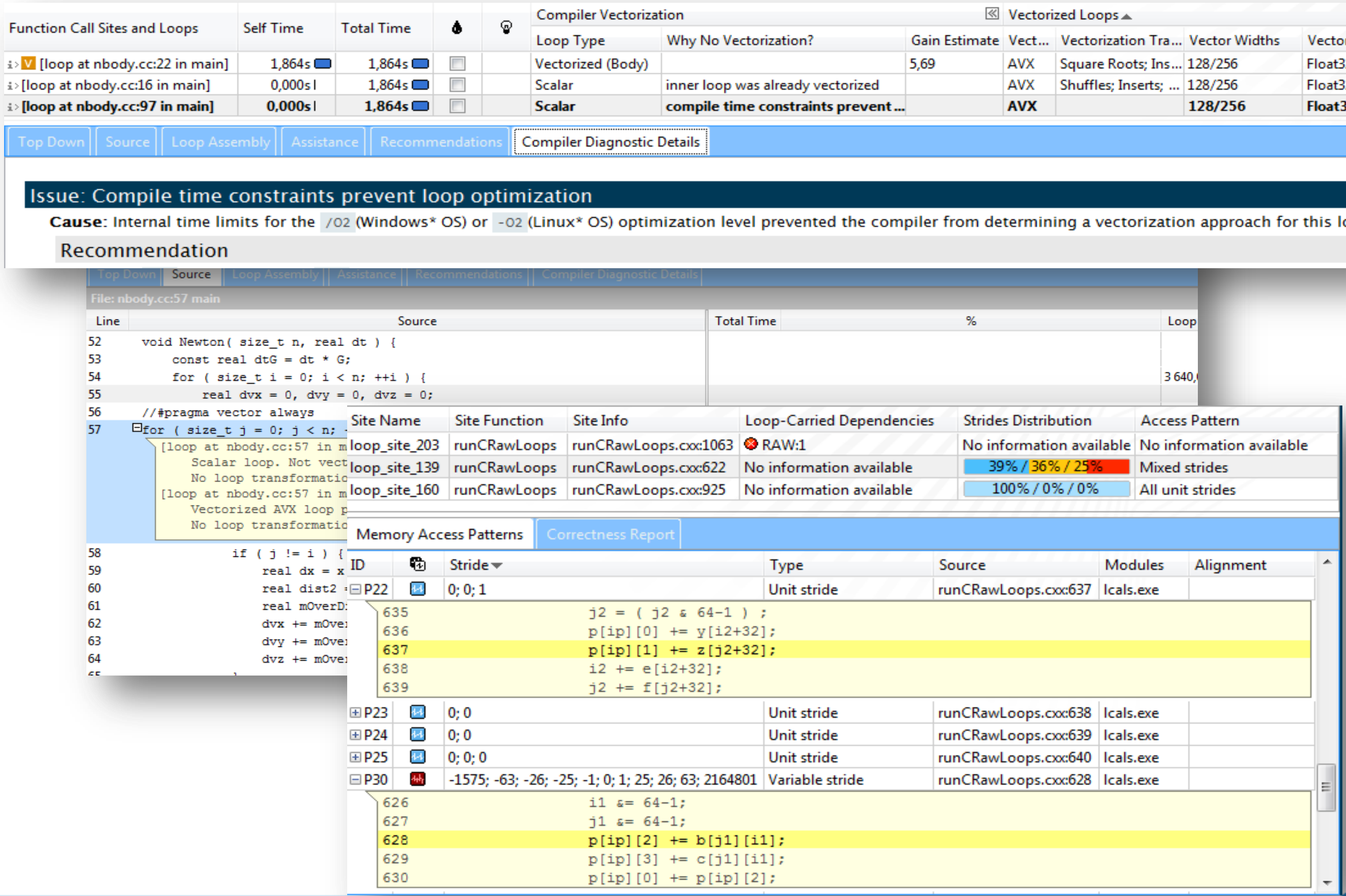
- **Intel® Software Development Emulator:**
  - Emulate (future) Intel® Architecture Instruction Set Extensions (e.g. Intel® AVX-512, Intel® MPX, …)
  - Use the "mix histogramming tool" to check for instructions using vectors
  - Also possible to debug the application while emulated
  - Source: https://software.intel.com/en-us/articles/intel-software-development-emulator
- **Intel® Architecture Code Analyzer:**
  - Statically analyze the data dependency, throughput and latency of code snippets (aka. kernels)
  - Considers ideal front-end, out-of-order engine and memory hierarchy conditions
  - Identifies binding of the kernel instructions to the processor ports & critical path
  - Source: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/

Validating Vectorization Success III

- **Optimization report:**
  - Linux*, OS X*: `-opt-report=<n>`, Windows*: `/Qopt-report:<n>`
    `n`: 0, …, 5 specifies level of detail; 2 is default (more later)
  - Prints optimization report with vectorization analysis
  - Also known as vectorization report for Intel® C++/Fortran Compiler before 15.0:
    Linux*, OS X*: `-vec-report=<n>`, Windows*: `/Qvec-report:<n>`
    Deprecated, don't use anymore – use optimization report instead!
- **Optimization report phase:**
  - Linux*, OS X*: `-opt-report-phase=<p>`,
    Windows*: `/Qopt-report-phase:<p>`
  - `<p>` is `all` by default; use `vec` for just the vectorization report
- **Optimization report file:**
  - Linux*, OS X*: `-opt-report-file=<f>`, Windows*: `/Qopt-report-file:<f>`
  - `<f>` can be `stderr`, `stdout` or a file (default: *.optrpt)

Validating Vectorization Success IV

- **Intel® Advisor XE 2016 (Vectorization Advisor)**



Function Call Sites and Loops	Self Time	Total Time			Loop Type	Why No Vectorization?	Gain Estimate	Vect...	Vectorization Tra...	Vector Widths	Vector ...
V [loop at nbody.cc:22 in main]	1,864s	1,864s			Vectorized (Body)		5,69	AVX	Square Roots; Ins...	128/256	Float32;...
[loop at nbody.cc:16 in main]	0,000s	1,864s			Scalar	inner loop was already vectorized		AVX	Shuffles; Inserts; ...	128/256	Float32;...
[loop at nbody.cc:97 in main]	**0,000s**	**1,864s**			**Scalar**	**compile time constraints prevent ...**		**AVX**		**128/256**	**Float32...**

Top Down | Source | Loop Assembly | Assistance | Recommendations | Compiler Diagnostic Details

Issue: Compile time constraints prevent loop optimization

Cause: Internal time limits for the `/O2` (Windows* OS) or `-O2` (Linux* OS) optimization level prevented the compiler from determining a vectorization approach for this loop.

Recommendation

File: nbody.cc:57 main

```
52      void Newton( size_t n, real dt ) {
53          const real dtG = dt * G;
54          for ( size_t i = 0; i < n; ++i ) {
55              real dvx = 0, dvy = 0, dvz = 0;
56      //#pragma vector always
57      for ( size_t j = 0; j < n; 
          [loop at nbody.cc:57 in m
              Scalar loop. Not vect
              No loop transformatio
          [loop at nbody.cc:57 in m
              Vectorized AVX loop p
              No loop transformatio
58                  if ( j != i ) {
59                      real dx = x
60                      real dist2 
61                      real mOverD
62                      dvx += mOve
63                      dvy += mOve
64                      dvz += mOve
```

Site Name	Site Function	Site Info	Loop-Carried Dependencies	Strides Distribution	Access Pattern
loop_site_203	runCRawLoops	runCRawLoops.cxx:1063	RAW:1	No information available	No information available
loop_site_139	runCRawLoops	runCRawLoops.cxx:622	No information available	39% / 36% / 25%	Mixed strides
loop_site_160	runCRawLoops	runCRawLoops.cxx:925	No information available	100% / 0% / 0%	All unit strides

Memory Access Patterns | Correctness Report

ID		Stride	Type	Source	Modules	Alignment
P22		0; 0; 1	Unit stride	runCRawLoops.cxx:637	lcals.exe	

```
635            j2 = ( j2 & 64-1 ) ;
636            p[ip][0] += y[i2+32];
637            p[ip][1] += z[j2+32];
638            i2 += e[i2+32];
639            j2 += f[j2+32];
```

ID		Stride	Type	Source	Modules	Alignment
P23		0; 0	Unit stride	runCRawLoops.cxx:638	lcals.exe	
P24		0; 0	Unit stride	runCRawLoops.cxx:639	lcals.exe	
P25		0; 0; 0	Unit stride	runCRawLoops.cxx:640	lcals.exe	
P30		-1575; -63; -26; -25; -1; 0; 1; 25; 26; 63; 2164801	Variable stride	runCRawLoops.cxx:628	lcals.exe	

```
626            i1 &= 64-1;
627            j1 &= 64-1;
628            p[ip][2] += b[j1][i1];
629            p[ip][3] += c[j1][i1];
630            p[ip][0] += p[ip][2];
```

Optimization Report Example

Example `novec.f90`:

```
1: subroutine fd(y)
2:   integer :: i
3:   real, dimension(10), intent(inout) :: y
4:   do i=2,10
5:     y(i) = y(i-1) + 1
6:   end do
7: end subroutine fd
```

```
$ ifort novec.f90 -opt-report=5
ifort: remark #10397: optimization reports are generated in *.optrpt
files in the output location

$ cat novec.optrpt
…
LOOP BEGIN at novec.f90(4,5)
   remark #15344: loop was not vectorized: vector dependence prevents
vectorization
   remark #15346: vector dependence: assumed FLOW dependence between y
line 5 and y line 5
   remark #25436: completely unrolled by 9
LOOP END
…
```

Agenda

- Introduction to SIMD for Intel® Architecture
- Compiler & Vectorization
- Validating Vectorization Success
- **Reasons for Vectorization Fails**
- Intel® Cilk™ Plus
- Summary

Reasons for Vectorization Fails I

Most frequent reasons:

- Data dependence
- Alignment
- Unsupported loop structure
- Non-unit stride access
- Function calls/in-lining
- Non-vectorizable Mathematical functions
- Data types
- Control depencence
- Bit masking

…and much more!

Key Theorem for Vectorization

A loop can be vectorized if and only if there is no cyclic dependency chain between the statements of the loop body!

- The theorem takes into account that certain semantic-preserving reordering transformations can be applied (e.g. loop distribution, loop fusion, etc.)
- The theorem assumes an "unlimited" vector length (VL). In cases where VL is limited, loop carried dependencies might be ignored if more than "VL" iterations are required to exist. **Thus in some cases vectorization for SSE or AVX might be still valid, opposed to the theorem!**

Example:
Although we have a cyclic dependency chain, the loop can be vectorized for SSE or AVX in case of VL being max. 3 times the data type size of array **A**.

```
DO I = 1, N
  A(I + 3) = A(I) + C
END DO
```

Disambiguation Hints I

- Disambiguating memory locations of pointers in C99:
  Linux*, OS X*: `-std=c99`, Windows*: `/Qstd=c99`
- Intel® C++ Compiler also allows this for other modes
  (e.g. `-std=c89`, `-std=c++0x`, …), too - **not standardized**, though:
  Linux*, OS X*: `-restrict`, Windows*: `/Qrestrict`
- Declaring pointers with keyword `restrict` asserts compiler that they only reference individually assigned, non-overlapping memory areas
- Also true for any result of pointer arithmetic (e.g. `ptr + 1` or `ptr[1]`)

Examples:

```
void scale(int *a, int *restrict b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}

void mult(int a[][NUM], int b[restrict][NUM])
{ ... }
```

Disambiguation Hints II

Directives:

- `#pragma ivdep` (C/C++) or `!DIR$ IVDEP` (Fortran)
- `#pragma simd` (C/C++) or `!DIR$ SIMD` (Fortran)

For C/C++:

- Assume no aliasing at all (dangerous!):
  Linux*, OS X*: `-fno-alias`, Windows*: `/Oa`
- Assume ISO C Standard aliasing rules:
  Linux*, OS X*: `-ansi-alias`, Windows*: `/Qansi-alias`
  Default with 15.0 and later but not with earlier versions!
- Turns on ANSI aliasing checker, too (thus recommended)
- No aliasing between function arguments:
  Linux*, OS X*: `-fargument-noalias`, Windows*: `/Qalias-args-`
- No aliasing between function arguments and global storage:
  Linux*, OS X*: `-fargument-noalias-global`, Windows*: N/A

Disambiguation Hints III

For Fortran:

- Assume no aliasing at all:
  Linux*, OS X*: `-fno-alias`, Windows*: `/Oa`
- Assume Fortran Standard aliasing rules:
  Linux*, OS X*: `-ansi-alias`, Windows*: `/Qansi-alias`

  Opposed to C/C++ this is default since ever!
- No aliasing of Cray* pointers:
  Linux*, OS X*: `-safe-cray-ptr`, Windows*: `/Qsafe-cray-ptr`

Alignment Hints for C/C++ I

- Aligned heap memory allocation by intrinsic/library call:
  - `void* _mm_malloc(int size, int base)`
  - Linux*, OS X* only:
    `int posix_memaligned(void **p, size_t base, size_t size)`

- `#pragma vector [aligned|unaligned]`
  - Only for Intel Compiler
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - **Use with care:**
    The assertion must be satisfied for all(!) data accesses in the loop!

Alignment Hints for C/C++ II

- Align attribute for variable declarations:
  - Linux*, OS X*, Windows*: `__declspec(align(base)) <var>`
  - Linux*, OS X*: `<var> __attribute__((aligned(base)))`
  - **Portability caveat:**
    `__declspec` is not known for GCC and `__attribute__` not for Microsoft Visual Studio*!

- Hint that start address of an array is aligned (Intel Compiler only):
  `__assume_aligned(<array>, base)`

Alignment Hints for Fortran

- `!DIR$ VECTOR [ALIGNED|UNALIGNED]`
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - **Use with care:**
    The assertion must be satisfied for all(!) data accesses in the loop!
- Hint that an entity in memory is aligned:
  `!DIR$ ASSUME_ALIGNED address1:base [, address2:base] ...`
- Align variables:
  `!DIR$ ATTRIBUTES ALIGN: base :: variable`
- Align data items globally:
  Linux*, OS X*: `-align <a>`, Windows*: `/align:<a>`
  - `<a>` can be `array<n>byte` with `<n>` defining the alignment for arrays
  - Other values for `<a>` are also possible, e.g.: `[no]commons`, `[no]records`, ...

All are Intel® Fortran Compiler only directives and options!

Alignment Impact: Example

Compiled both cases using **-xAVX**:

```
void mult(double* a, double* b, double* c)
{
  int i;
#pragma vector unaligned
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
..B2.2:
  vmovupd    (%rdi,%rax,8), %xmm0
  vmovupd    (%rsi,%rax,8), %xmm1
  vinsertf128 $1, 16(%rsi,%rax,8), %ymm1, %ymm3
  vinsertf128 $1, 16(%rdi,%rax,8), %ymm0, %ymm2
  vmulpd     %ymm3, %ymm2, %ymm4
  vmovupd    %xmm4, (%rdx,%rax,8)
  vextractf128 $1, %ymm4, 16(%rdx,%rax,8)
  addq       $4, %rax
  cmpq       $1000000, %rax
  jb         ..B2.2
```

More efficient if aligned:

```
void mult(double* a, double* b, double* c)
{
  int i;
#pragma vector aligned
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
..B2.2:
  vmovupd    (%rdi,%rax,8), %ymm0
  vmulpd     (%rsi,%rax,8), %ymm0, %ymm1
  vmovntpd   %ymm1, (%rdx,%rax,8)
  addq       $4, %rax
  cmpq       $1000000, %rax
  jb         ..B2.2
```

Non-Unit Stride Access

- Non-consecutive memory locations are being accessed in the loop
- Vectorization works best with contiguous memory accesses
- Vectorization still be possible for non-contiguous memory access, but…
  - Data arrangement operations might be too expensive
    (e.g. access pattern linear/regular)
  - Vectorization report issued when too expensive:
    Loop was not vectorized: vectorization possible but seems inefficient

  For Fortran: Use **CONTIGUOUS** attribute, if possible!

- Examples:

```
for(i = 0; i <= MAX; i++) {
  for(j = 0; j <= MAX; j++) {
    D[i][j] += 1;                // Unit stride
    D[j][i] += 1;                // Non-unit stride but linear
    A[B[j]] += 1;                // Non-unit stride (scatter)
  }
}
```

Vectorizable Mathematical Functions

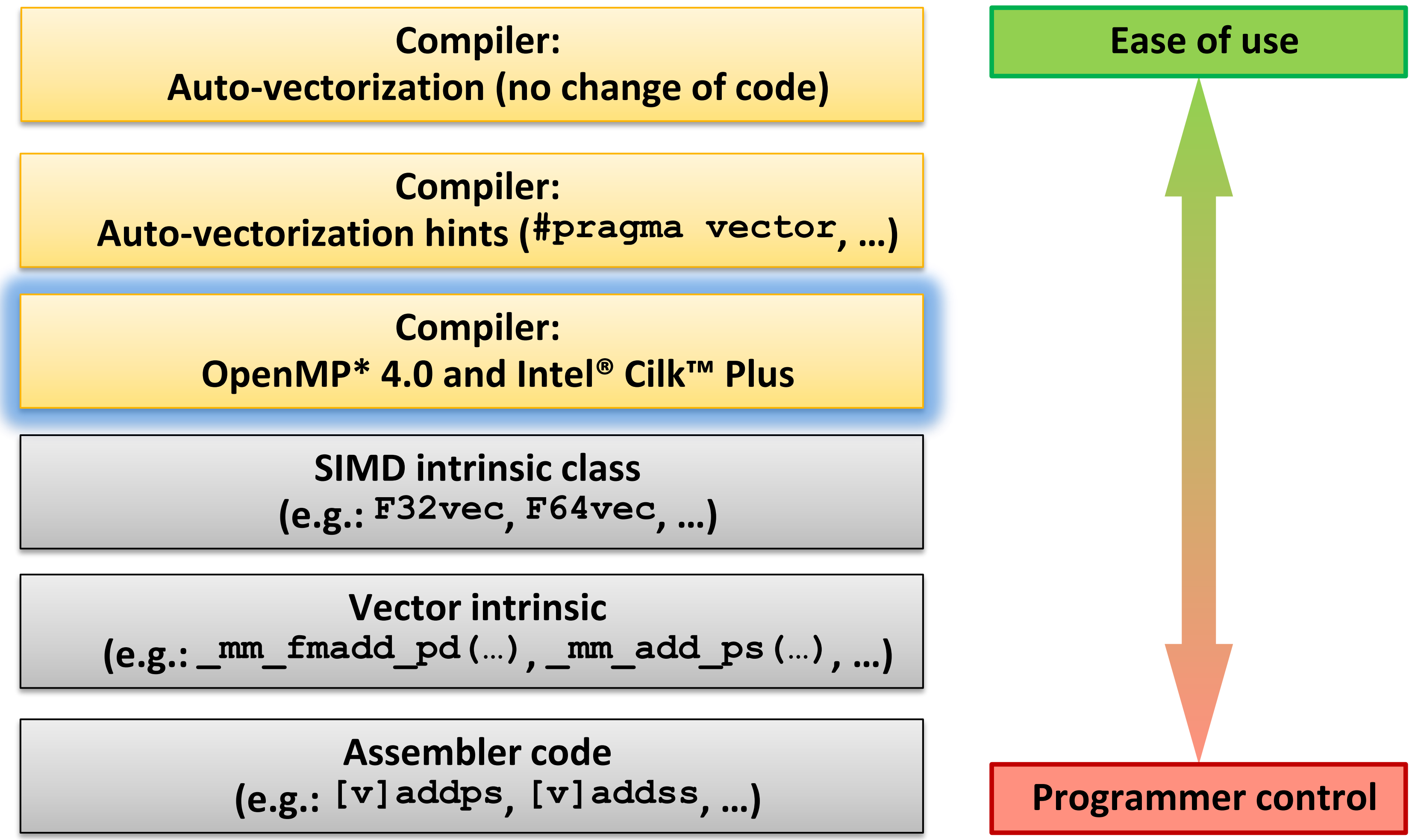
- Calls to most mathematical functions in a loop body can be vectorized using “Short Vector Math Library”:
  - Short Vector Math Library (**libsvml**) provides vectorized implementations of different mathematical functions
  - Optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but which is optimized for throughput
- Routines in **libsvml** can also be called explicitly, using intrinsics (see manual)
- These mathematical functions are currently supported:

acos	acosh	asin	asinh	atan	atan2	atanh	cbrt
ceil	cos	cosh	erf	erfc	erfinv	exp	exp2
fabs	floor	fmax	fmin	log	log10	log2	pow
round	sin	sinh	sqrt	tan	tanh	trunc	

Agenda

- Introduction to SIMD for Intel® Architecture
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- **Intel® Cilk™ Plus**
- Summary

Intel® Cilk™ Plus


Compiler:
Auto-vectorization (no change of code)
Compiler:
Auto-vectorization hints (#pragma vector, …)
Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus
SIMD intrinsic class
(e.g.: F32vec, F64vec, …)
Vector intrinsic
(e.g.: _mm_fmadd_pd(…), _mm_add_ps(…), …)
Assembler code
(e.g.: [v]addps, [v]addss, …)
Ease of use
Programmer control

Intel® Cilk™ Plus

Task Level Parallelism

Simple Keywords

Set of keywords, for expression of task parallelism:

```
cilk_spawn
cilk_sync
cilk_for
```

Reducers (Hyper-objects)

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

Data Level Parallelism

Array Notation

Provide data parallelism for sections of arrays or whole arrays

```
mask[:] = a[:] < b[:] ? -1 : 1;
```

SIMD-enabled Functions

Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters

Runtime system APIs, Environment variables, pragmas

Intel® Cilk™ Plus

Task Level Parallelism

Simple Keywords

Set of keywords, for expression of task parallelism:

`cilk_spawn`

`cilk_sync`

`cilk_for`

Reducers (Hyper-objects)

Reliable access to nonlocal variables without races

`cilk::reducer_opadd<int> sum(3);`

Data Level Parallelism

Array Notation

Provide data parallelism for sections of arrays or whole arrays

`mask[:] = a[:] < b[:] ? -1 : 1;`

SIMD-enabled Functions

Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters

Runtime system APIs, Environment variables, pragmas

Intel® Cilk™ Plus Pragma/Directive I

C/C++: **`#pragma simd [clause [,clause]…]`**

Fortran: **`!DIR$ SIMD [clause [,clause]…]`**

Without any clause, the directive "enforces" vectorization of the loop, ignoring all dependencies (even if they are proved!)

Example:

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

Without SIMD directive, vectorization likely fails since there are too many pointer references to do a run-time check for overlapping (compiler heuristic). The compiler won't create multiple versions here.

Using the directive asserts the compiler that none of the pointers are overlapping.

`#pragma simd` Clauses for C/C++

- **`vectorlength(n1 [,n2] …)`**
  `n1`, **`n2`**, … must be **`2`**, **`4`**, **`8`**, …: The compiler can assume a safe vectorization for a vector length of **`n1`**, **`n2`**, …; alternative: **`vectorlengthfor(type)`**
- **`private(v1, v2, …)`**
  Variables private to each iteration; supersets (extensions):
  - **`firstprivate(…)`**: initial value is broadcast to all private instances
  - **`lastprivate(…)`**: last value is copied out from the last iteration instance
- **`linear(v1:step1, v2:step2, …)`**
  For every iteration of original scalar loop **`v1`** is incremented by **`step1`**, … etc. Therefore it is incremented by **`step1 * VL`** for the vectorized loop.
- **`reduction(operator:v1, v2, …)`**
  Variables **`v1`**, **`v2`**, … etc. are reduction variables for operation **`operator`**
- **`[no]assert`**
  Warning (default: **`noassert`**) or error with failed vectorization

`!DIR$ SIMD` Clauses for Fortran

- **`VECTORLENGTH(n1 [,n2] …)`**
  `n1`, **`n2`**, … must be **`2`**, **`4`**, **`8`**, …: The compiler can assume a safe vectorization for a vector length of **`n1`**, **`n2`**, …
- **`PRIVATE(v1, v2, …)`**
  Variables private to each iteration; supersets (extensions):
  - **`FIRSTPRIVATE(…)`**: initial value is broadcast to all private instances
  - **`LASTPRIVATE(…)`**: last value is copied out from the last iteration instance
- **`LINEAR(v1:step1, v2:step2, …)`**
  For every iteration of original scalar loop **`v1`** is incremented by **`step1`**, … etc. Therefore it is incremented by **`step1 * VL`** for the vectorized loop.
- **`REDUCTION(operator:v1, v2, …)`**
  Variables **`v1`**, **`v2`**, … etc. are reduction variables for operation **`operator`**
- **`[NO]ASSERT`**
  Warning (default: **`NOASSERT`**) or error with failed vectorization

!DIR$ SIMD Example for Fortran

Problem:

"Enforced" vectorization still fails

with the following message:

```
!DIR$ SIMD
do i = 1,n
  if (a(i) .GT. 0) then
    sum2 = sum2 + a(i) * b(i)
  else
    sum2 = sum2 + a(i)
  endif
enddo
```

```
loop was not vectorized: conditional assignment to a scalar
loop was not vectorized with "simd"
```

Solution:

Clarify that scalar is a reduction with operator **+**.

```
!DIR$ SIMD REDUCTION(+:sum2)
do i = 1,n
  if (a(i) .GT. 0) then
    sum2 = sum2 + a(i) * b(i)
  else
    sum2 = sum2 + a(i)
  endif
enddo
```

Attention:

Same as for OpenMP* reduction variables can only be associated to one operator each!

IVDEP vs. SIMD Pragma/Directives

Differences between IVDEP & SIMD pragmas/directives:

- `#pragma ivdep` (C/C++) or `!DIR$ IVDEP` (Fortran)
  - Ignore vector dependencies (IVDEP): Ignore assumed but not proven dependencies for a loop
  - Example:

```
void foo(int *a, int k, int c, int m)
{
#pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

- `#pragma simd` (C/C++) or `!DIR$ SIMD` (Fortran):
  - Aggressive version of IVDEP: Ignores **all** dependencies inside a loop and ignore efficiency heursitic
  - It's an imperative that forces the compiler try everything to vectorize
  - **Attention:** This can break semantically correct code!
    However, it can **vectorize** code legally in some cases that wouldn't be possible otherwise!

Intel® Cilk™ Plus

Task Level Parallelism

Simple Keywords

Set of keywords, for expression of task parallelism:

```
cilk_spawn
cilk_sync
cilk_for
```

Reducers (Hyper-objects)

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

Data Level Parallelism

Array Notation

Provide data parallelism for sections of arrays or whole arrays

```
mask[:] = a[:] < b[:] ? -1 : 1;
```

SIMD-enabled Functions

Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters

Runtime system APIs, Environment variables, pragmas

SIMD-Enabled Functions Syntax

Windows*:

```
__declspec(vector([clause [,clause]…]))
  function definition or declaration
```

Linux*/OS* X:

```
__attribute__((vector([clause [,clause]…])))
  function definition or declaration
```

- C/C++ only
- Intent:
  Express work as scalar operations (kernel) and let compiler create a vector version of it. The size of vectors can be specified at compile time (SSE, AVX, …) which makes it portable!
- **Remember:**
  Both the function definition as well as the function declaration (header file) need to be specified like this!

SIMD-Enabled Functions Clauses

- **`processor(cpuid)`**
  `cpuid` for which (Intel) processor to create a vector version
- **`vectorlength(len)`**
  `len` must be power of 2: Allow as many elements per argument
- **`linear(v1:step1, v2:step2, …)`**
  Defines **`v1`**, **`v2`**, … to be private to SIMD lane and to have linear (**`step1`**, **`step2`**, …) relationship when used in context of a loop
- **`uniform(a1, a2, …)`**
  Arguments **`a1`**, **`a2`**, … etc. are not treated as vectors (constant values across SIMD lanes)
- **`[no]mask`**: SIMD-enabled function called only inside branches (masked) or never (not masked)

Intrinsic also available: **`__intel_simd_lane()`**:
Return the SIMD lane with range: **`[0:vector length - 1]`**

SIMD-Enabled Functions

Write a function for one element and add `__declspec(vector)`:

```
__declspec(vector)
float foo(float a, float b, float c, float d)
{
  return a * b + c * d;
}
```

Call the scalar version:

```
e = foo(a, b, c, d);
```

Call scalar version via SIMD loop:

```
#pragma simd
for(i = 0; i < n; i++) {
  A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

Call it with array notations:

```
A[:] = foo(B[:], C[:], D[:], E[:]);
```

SIMD-Enabled Functions: Invocation

```
__declspec(vector)float my_simdf (float b) { …  }
```

Construct	Example	Semantics

Standard for loop	`for (j = 0; j < N; j++) {`
	`  a[j] = my_simdf(b[j]);`
	`}`
		Single thread, maybe auto-vectorizable

#pragma simd	`#pragma simd`
	`for (j = 0; j < N; j++) {`
	`  a[j] = my_simdf(b[j]);`
	`}`
		Single thread, vectorized; use the appropriate vector version

Array notation	`a[:] = my_simdf(b[:]);`	Single thread, vectorized

OpenMP* 4.0	`#pragma omp parallel for simd`
	`for (j = 0; j < N; j++) {`
	`  a[j] = my_simdf(b[j]);`
	`}`
		Multi-threaded, vectorized

Intel® Cilk™ Plus

Task Level Parallelism

Simple Keywords

Set of keywords, for expression of task parallelism:

`cilk_spawn`

`cilk_sync`

`cilk_for`

Reducers (Hyper-objects)

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

Data Level Parallelism

Array Notation

Provide data parallelism for sections of arrays or whole arrays

```
mask[:] = a[:] < b[:] ? -1 : 1;
```

SIMD-enabled Functions

Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters

Runtime system APIs, Environment variables, pragmas

Array Notation Extension: Syntax I

- An extension to C/C++ only
- Perform operations on sections of arrays in parallel
- Example:

```
for(i = 0; i < …; i++)
  A[i] = B[i] + C[i];
```

$\Rightarrow$

```
A[:] = B[:] + C[:];
```

Not exactly the same: Aliasing is ignored by Array Notations!

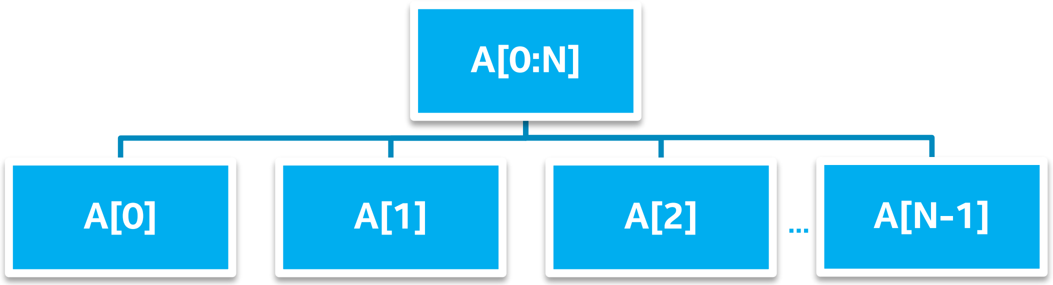
- Well suited for code that:
  - Performs per-element operations on arrays
  - Without an implied order between them (aliasing is ignored)
  - With an intent to execute in vector instructions

Array Notation Extension: Syntax II

- Syntax:

```
A[:]
A[start_index : length]
A[start_index : length : stride]
```

  - Use a ":" for all elements (if size is known)
  - "length" specifies number of elements of subset
  - "stride": distance between elements for subset

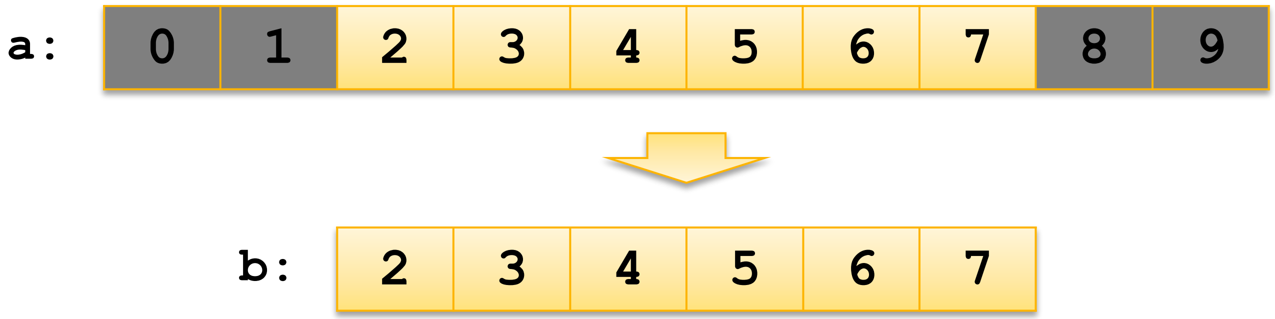


Explicit Data Parallelism Based on C/C++ Arrays

Array Notation Extension: Example I

Accessing a section of an array:

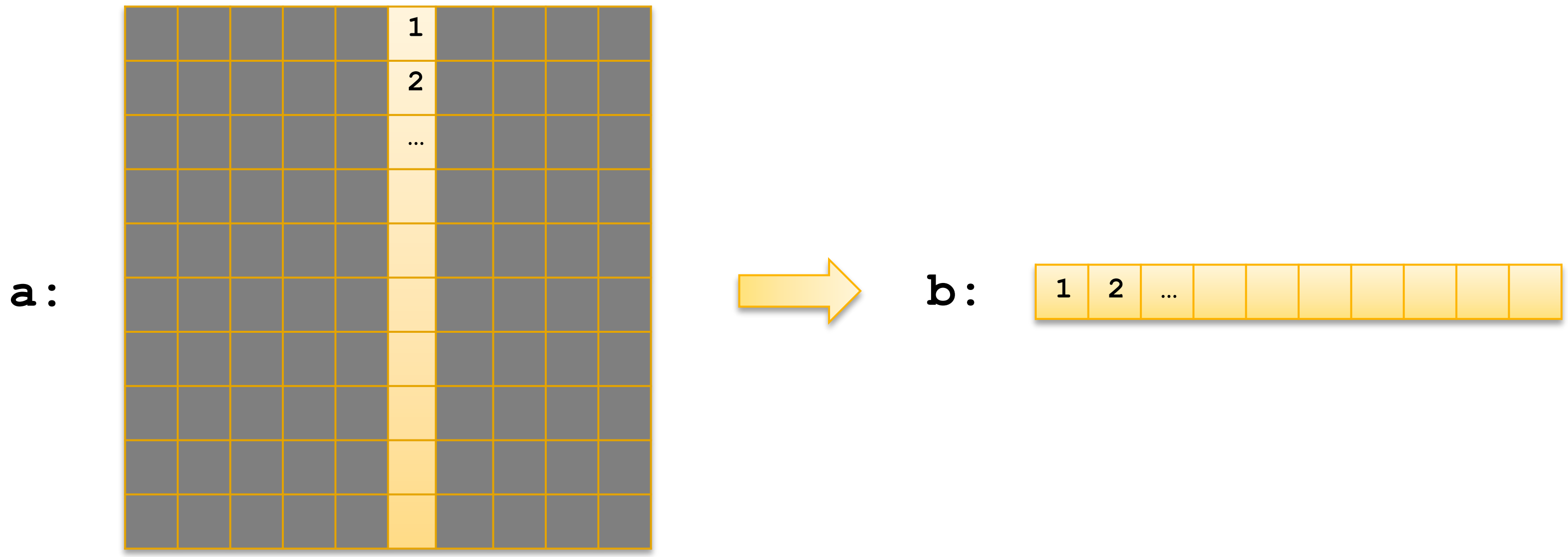
```
float a[10], b[6];
…
// allocate *b
…
b[:] = a[2:6];
…
```



Array Notation Extension: Example II

Section of 2D array:

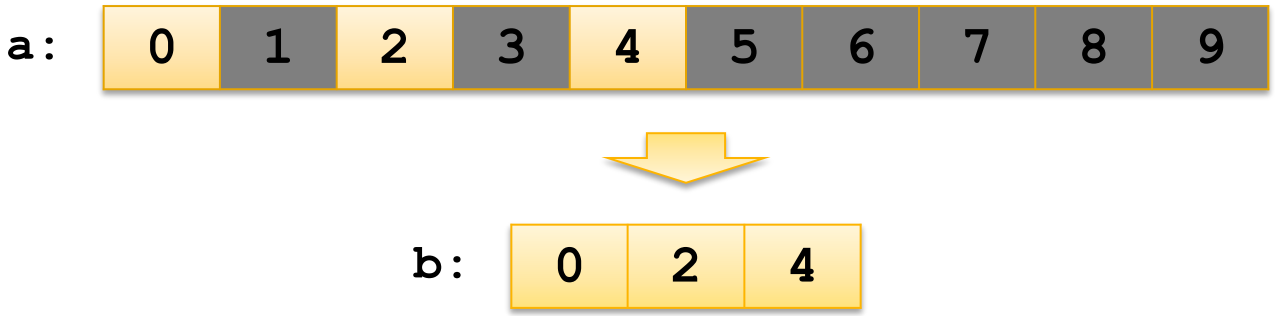
```
float a[10][10], *b;
…
// allocate *b
…
b[0:10] = a[:][5];
…
```



Array Notation Extension: Example III

Strided section of an array:

```
float a[10], *b;
…
// allocate *b
…
b[0:3] = a[0:3:2];
…
```



Array Notation Extension: Operators

Most C/C++ operators are available for array sections:

`+`, `-`, `*`, `/`, `%`, `<`, `==`, `!=`, `>`, `|`, `&`, `^`, `&&`, `||`, `!`, `-` (unary), `+` (unary), `++`, `--`, `+=`, `-=`, `*=`, `/=`, `*` (pointer de-referencing)

Examples:

```
a[:] * b[:]                   // element-wise multiplication
a[3:2][3:2] + b[5:2][5:2]     // matrix addition
a[0:4][1:2] + b[1:2][0:4]     // error, different rank sizes
a[0:4][1:2] + c               // adds scalar c to array section
```

- Operators are implicitly mapped to all elements of the array section operands.
- Operations on different elements can be executed in parallel without any ordering constraints.
- Array operands must have the same **rank** and **size**.
- Scalar operands are automatically expanded.

Array Notation Extension: Reductions

Combine array section elements using a predefined operator, or a user function:

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum is 10
res = __sec_reduce(0, a[:], func);
    // apply function func to all
    // elements in a[], initial value is 0
int func(int arg1, int arg2)
{
   return arg1 + arg2;
}
```

Other reductions (list not exhaustive):

```
__sec_reduce_mul, __sec_reduce_all_zero,
__sec_reduce_all_nonzero, __sec_reduce_any_nonzero,
__sec_reduce_max, __sec_reduce_min,
__sec_reduce_max_ind, __sec_reduce_min_ind
```

Much more! Take a look at the specification:
https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm

Array Notation Extension: Example I

Serial version:

```
float dot_product(unsigned int size, float A[size], float B[size])
{
    int i;
    float dp = 0.0f;
    for (i=0; i<size; i++) {
         dp += A[i] * B[i];
    }
    return dp;
}
```

Array Notation version:

```
float dot_product(unsigned int size, float A[size], float B[size])
{
    // A[:] can also be written as A[0:size]
    return __sec_reduce_add(A[:] * B[:]);
}
```

Intel® Cilk™ Plus

Compilers

The following compilers support Intel® Cilk™ Plus:

- GNU* GCC 4.9:
  - Exception: _cilk_for (Thread Level Parallelism) which will be added with GCC 5.0
  - Enable with `-fcilkplus`
- clang/LLVM 3.5:
  - Not official yet but development branch exists: http://cilkplus.github.io/
  - Enable with `-fcilkplus`
- Intel® C++/Fortran Compiler:
  Beginning with 12.0; newer features added over time (see Release Notes)

Agenda

- Introduction to SIMD for Intel® Architecture
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Intel® Cilk™ Plus
- **Summary**

Summary

- Intel® C++ Compiler and Intel® Fortran Compiler provide sophisticated and flexible support for vectorization
- They also provide a rich set of reporting features that help verifying vectorization and optimization in general
- Directives and compiler switches permit fine-tuning for vectorization
- Vectorization can even be enforced for certain cases where language standards are too restrictive
- Understanding of concepts like dependency and alignment is required to take advantage from SIMD features
- Intel® C++/Fortran Compiler can create multi-version code to address a broad range of processor generations, Intel and non-Intel processors and individually exploiting their feature set

References

- Aart Bik: “The Software Vectorization Handbook”
  http://www.intel.com/intelpress/sum_vmmx.htm
- Randy Allen, Ken Kennedy: “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”
- Steven S. Muchnik, “Advanced Compiler Design and Implementation”
- Intel Software Forums, Knowledge Base, White Papers, Tools Support (see http://software.intel.com)
  Sample Articles:
  - http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/
  - http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
  - http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/

The Software Vectorization Handbook
intel
OPTIMIZING COMPILERS for MODERN ARCHITECTURES
Randy Allen & Ken Kennedy
Advanced COMPILER DESIGN & IMPLEMENTATION
Steven S. Muchnick

Thank you!

Legal Disclaimer & Optimization Notice