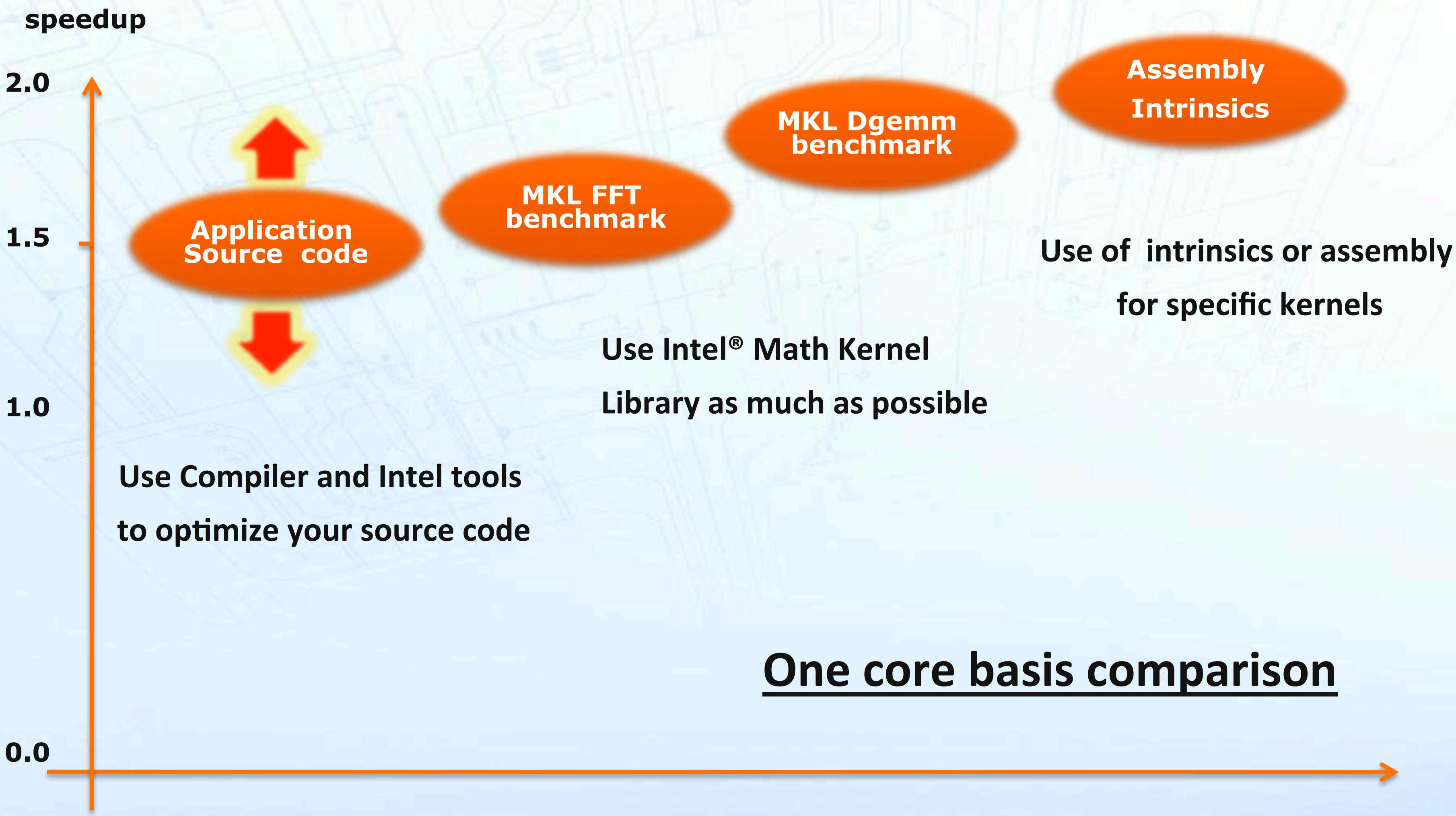
AGENDA

- Vectorization

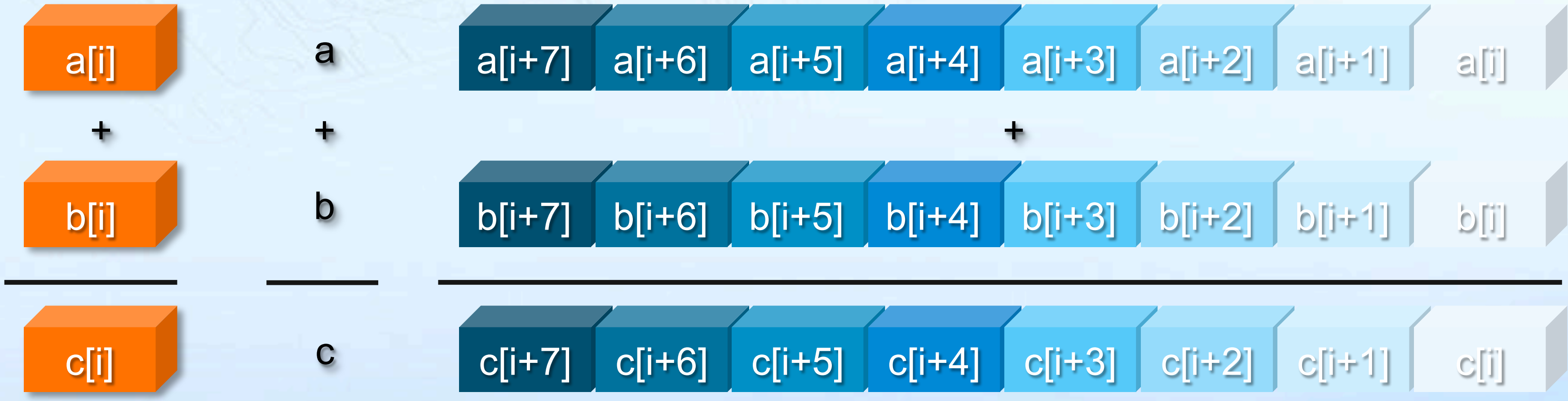
Do not re_invent the wheel



Vectorization of Code

- **Transform sequential code to exploit vector processing capabilities (SIMD)**
  - **Manually by explicit syntax**
  - **Automatically by tools like a compiler**

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



Intel® MKL: Optimized Mathematical Building Blocks

Linear Algebra

- BLAS
- LAPACK
- Sparse Solvers
  - Iterative
  - Pardiso*
- ScaLAPACK

Fast Fourier Transforms

- Multidimensional
- FFTW interfaces
- Cluster FFT

Vector Math

- Trigonometric
- Hyperbolic
- Exponential, Log
- Power / Root

Vector RNGs

- Congruential
- Wichmann-Hill
- Mersenne Twister
- Sobol
- Neiderreiter
- Non-deterministic

Summary Statistics

- Kurtosis
- Variation coefficient
- Order statistics
- Min/max
- Variance-covariance

And More

- Splines
- Interpolation
- Trust Region
- Fast Poisson Solver

Intel® MKL is an integral part of Intel® Composer XE

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

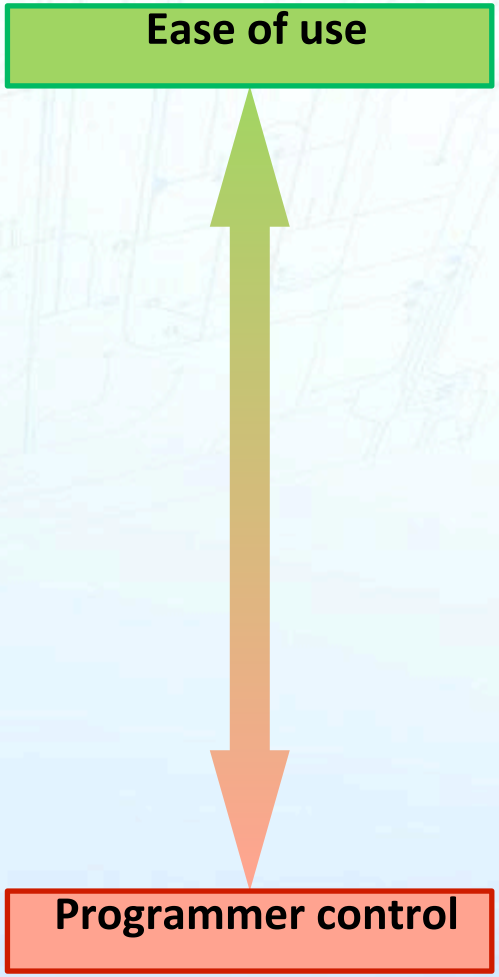
Compiler:
Auto-vectorization hints (`#pragma vector`, …)

Compiler:
Intel® Cilk™ Plus Array Notation Extensions

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`, …)



Control Vectorization !

Provides details on vectorization success & failure:

Linux*, Mac OS* X: **`-vec-report<n>`**, Windows*: **`/Qvec-report<n>`**

n	Diagnostic Messages
0	Tells the vectorizer to report no diagnostic information. Useful for turning off reporting in case it was enabled on command line earlier.
1	Tells the vectorizer to report on vectorized loops. [default if **n** missing]
2	Tells the vectorizer to report on vectorized and non-vectorized loops.
3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences.
4	Tells the vectorizer to report on non-vectorized loops.
5	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.
6*	Tells the vectorizer to use greater detail when reporting on vectorized and non-vectorized loops and any proven or assumed data dependences.

*: First available with Intel® Composer XE 2013

Vectorization Report II

```
35:     subroutine fd( y )
36:     integer :: i
37:     real, dimension(10), intent(inout) :: y
38:     do i=2,10
39:        y(i) = y(i-1) + 1
40:     end do
41:     end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence
of vector dependence.

novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.

novec.f90(38:3-38:3):VEC:MAIN_:  loop was not vectorized:
existence of vector dependence
```

Note:

In case inter-procedural optimization (`-ipo` or `/Qipo`) is activated and compilation and linking are separate compiler invocations, the switch to enable reporting needs to be added to the link step!

Reasons for Vectorization Fails & How to Succeed

- Most frequent reason is **Dependence**:
  Minimize dependencies among iterations by design!
- **Alignment**: Align your arrays/data structures
- **Function calls in loop body**: Use aggressive in-lining (IPO)
- **Complex control flow/conditional branches**:
  Avoid them in loops by creating multiple versions of loops
- **Unsupported loop structure**: Use loop invariant expressions
- **Not inner loop**: Manual loop interchange possible?
- **Mixed data types**: Avoid type conversions
- **Non-unit stride between elements**: Possible to change algorithm to allow linear/consecutive access?
- **Loop body too complex reports**: Try splitting up the loops!
- **Vectorization seems inefficient reports**: Enforce vectorization, benchmark !

IVDEP vs. SIMD Pragma/Directives

Differences between IVDEP & SIMD pragmas/directives:

- `#pragma ivdep` (C/C++) or `!DIR$ IVDEP` (Fortran)
  - Ignore vector dependencies (IVDEP):
    Compiler ignores 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
  - It's an imperative that forces the compiler try everything to vectorize
  - Efficiency heuristic is ignored
  - **Attention: This can break semantically correct code!**
    However, it can <u>vectorize</u> code legally in some cases that wouldn't be possible otherwise!

AGENDA

- **Validation**

Floating Point (FP) Programming Objectives

- **Accuracy**
  - Produce results that are “close” to the correct value
    - ✓ Measured in relative error, possibly in ulp
- **Reproducibility**
  - Produce consistent results
    - ✓ From one run to the next
    - ✓ From one set of build options to another
    - ✓ From one compiler to another
    - ✓ From one platform to another
- **Performance**
  - Produce the most efficient code possible

These options usually conflict!
Judicious use of compiler options lets you control the tradeoffs.
Different compilers have different defaults.

Definition . From Gustafson « reminders »

Precision = Digits available to store a number ("32-bit" or "4 decimal", for example)

Accuracy = Number of valid digits in a result ("to three significant digits", for example)

ULP = Unit of Least Precision.

Precision is not a goal.

Precision is the means, accuracy is the end.

Users are Interested in Consistent Numerical Results

- **Root cause for variations in results**
  - floating-point numbers → order of computation matters!
  - Single precision arithmetic example (a+b)+c != a+(b+c)

$2^{26} - 2^{26} + 1 = 1$	(infinitely precise result)
$(2^{26} - 2^{26}) + 1 = 1$	(correct IEEE single precision result)
$2^{26} - (2^{26} - 1) = 0$	(correct IEEE single precision result)

- **Conditions that affect the order of computations**
  - Different code branches (e.g. SSE2 versus AVX)
  - Memory alignment (scalar or vector code)
  - Dynamic parallel task / thread / rank scheduling

- **Bitwise repeatable/reproducible results**

repeatable	= results the same as last run (same conditions)
reproducible	= results the same as results in other environments
Environments	= OS / architecture / # threads / CPU /

```
4.012345678901111
4.012345678902222
4.012345678902222
4.012345678901111
4.012345678902222
4.012345678901111
4.012345678901111
4.012345678901111
4.012345678902222
4.012345678902222
4.012345678901111
4.012345678902222
4.012345678901111
4.012345678902222
4.012345678902222
4.012345678901111
…
```

The –fp-model switch

- **-fp-model**
  - fast [=1]   allows value-unsafe optimizations (default)
  - fast=2   allows additional approximations (very unsafe)
  - precise   value-safe optimizations only
    (also source, double, extended)
  - except   enable floating point exception semantics
  - strict   precise + except + disable fma +
    don't assume default floating-point environment

- Replaces old switches –mp, -fp-port, etc (don't use!)

- **-fp-model precise -fp-model source**
  - recommended for ANSI/ IEEE standards compliance,   C++ & Fortran
  - "source" is default with "precise" on Intel 64 Linux

Value Safety

ANSI/ IEEE standards compliance C++ & Fortran:

`-fp-model source` or `-fp-model precise`

- Prevents vectorization of reductions
- No use of "fast" division or square root

Ensures 'Value Safety' by disallowing:

x / x ⇔ 1.0	x could be 0.0, ∞, or NaN
x – y ⇔ - (y – x)	If x equals y, x – y is +0.0 while – (y – x) is -0.0
x – x ⇔ 0.0	x could be ∞ or NaN
x * 0.0 ⇔ 0.0	x could be -0.0, ∞, or NaN
x + 0.0 ⇔ x	x could be -0.0
(x + y) + z ⇔ x + (y + z)	General reassociation is not value safe
(x == x) ⇔ true	x could be NaN

Value Safety

Affected Optimizations include:

- Reassociation
- Flush-to-zero
- Expression Evaluation, various mathematical simplifications
- Math library approximations
- Approximate divide and sqrt

[-no]-prec-div /Qprec-div[-]

- Enables[disables] various divide optimizations
  - x / y ⇔ x * (1.0 / y)
  - Approximate divide and reciprocal

[-no]-prec-sqrt /Qprec-sqrt[-]

- Enables[disables] approximate sqrt and reciprocal sqrt

Intel® Math Kernel Library

- Linear algebra, FFTs, sparse solvers, statistical, …
  - Highly optimized, vectorized
  - Threaded internally using OpenMP*
  - Repeated runs may not give identical results

- **C**onditional **B**it**W**ise **R**eproducibility
  - Repeated runs give identical results under certain conditions:
    - Same number of threads
    - OMP_SCHEDULE=static (the default)
    - Same OS and architecture (e.g. Intel 64)
    - Same microarchitecture, or specify a minimum microarchitecture
    - Consistent data alignment
  - Call **mkl_bwr_set**(…)

Conditional Numerical Reproducibility

- Root cause for variations in results
  - With floating-point numbers → order of computation matters!
  - Example $(a+b)+c \neq a+(b+c)$

$2^{-63} + 1 + -1 = 2^{-63}$		(infinitely precise result)
$(2^{-63} + 1) + -1 = 0$		(correct IEEE double precision result)
$2^{-63} + (1 + -1) = 2^{-63}$		(correct IEEE double precision result)

- Intel MKL 11.0 for Xeon includes deterministic scheduling (fixed # of threads) and code path options
  - To get the same results on every Intel processor supporting AVX instructions or later
    - function call: mkl_cbwr_set(MKL_CBWR_AVX)
    - environment variable: set MKL_CBWR_BRANCH="AVX"
- Intel MKL 11.1 removes the data alignment restriction

Get reproducible results despite non-associative floating-point math

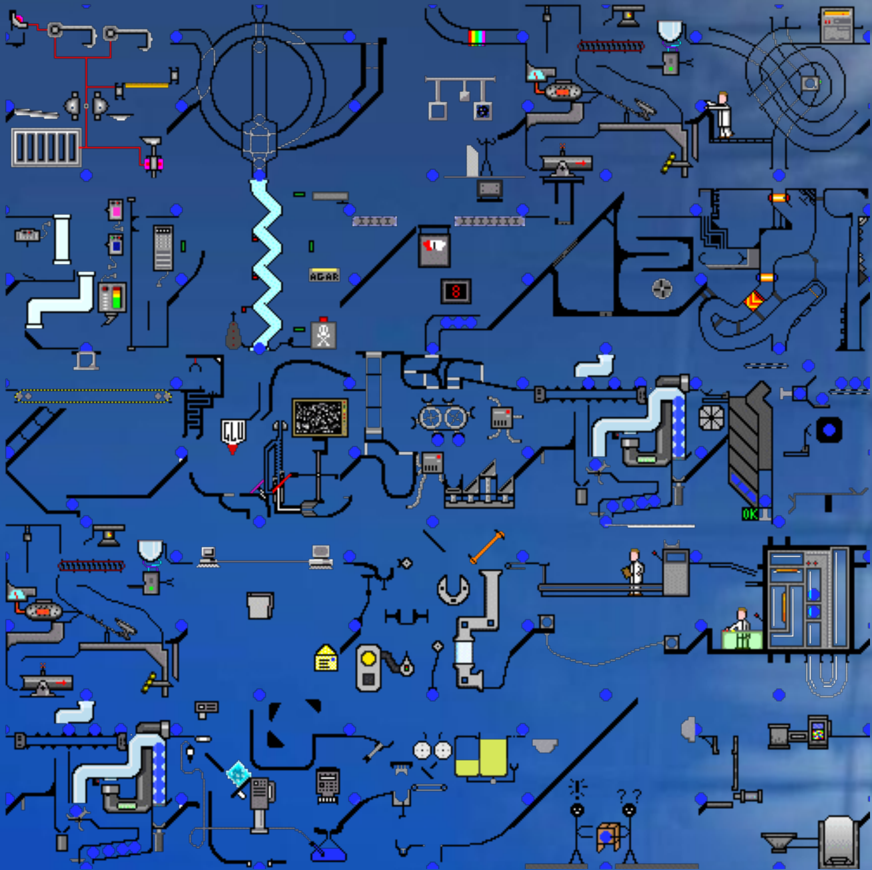
Reproducibility of Reductions in OpenMP*

- Each thread has its own partial sum
  - Breakdown, & hence results, depend on number of threads
  - Partial sums are summed at end of loop
  - Order of partial sums is undefined (OpenMP standard)
    - First come, first served
    - Result may vary from run to run (even for same # of threads)
    - For both gcc and icc
    - Can be more accurate than serial sum
  - For icc, option to define the order of partial sums (tree)
    - Makes results reproducible from run to run
    - export KMP_FORCE_REDUCTION=tree (may change!)
      - ✓ May also help accuracy
      - ✓ Possible slight performance impact, depends on context
      - ✓ Requires static scheduling, fixed number of threads
      - ✓ currently undocumented ("black belt", at your own risk)
      - ✓ See example

References

- [1] Aart Bik: “The Software Vectorization Handbook”
  http://www.intel.com/intelpress/sum_vmmx.htm
- [2] Randy Allen, Ken Kennedy: “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”
- [3] Steven S. Muchnik, “Advanced Compiler Design and Implementation”
- [4] 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 Intel® C++ and Fortran Compiler Documentation, “Floating Point Operations”
- “Consistency of Floating-Point Results using the Intel® Compiler”
  http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/
- Goldberg, David: "What Every Computer Scientist Should Know About Floating-Point Arithmetic“ *Computing Surveys*, March 1991, pg. 203
- the new Intel® BWR features – see this article for more details
- We need your feedback on missing, failing or suboptimal compiler functionality
- Please file a Premier case or post your findings/wishes to the compiler user forum

Questions



"Prediction is very difficult, especially about the future"

by Niels Bohr, Physicist, 1885-1962