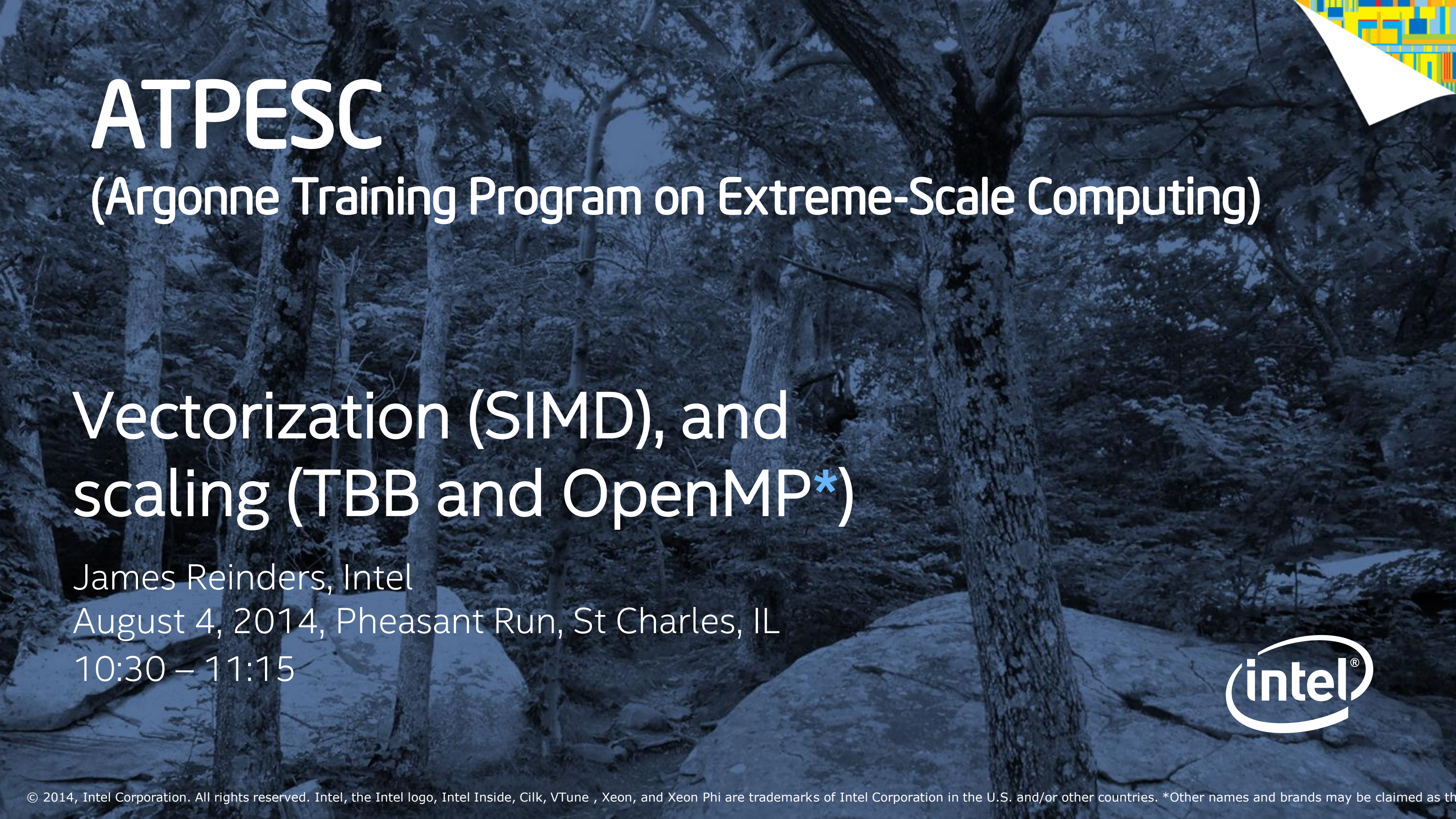
We resume @ 10:30am

ATPESC

(Argonne Training Program on Extreme-Scale Computing)

Vectorization (SIMD), and scaling (TBB and OpenMP*)

James Reinders, Intel
August 4, 2014, Pheasant Run, St Charles, IL
10:30 – 11:15



ATPESC

(Argonne Training Program on Extreme-Scale Computing)

Vectorization (SIMD), and scaling (TBB and OpenMP*)

James Reinders, Intel

August 4, 2014, Pheasant Run, St Charles, IL
10:30 – 11:15

Using A Single Vector Lane Can Inhibit Performance

What Maximum Performance Can Look Like With Permission To Use All Lanes And Resources

Modernized Software Delivers Significant Performance Advantages

Following up on “data parallelism is KEY”

- dive into the topic of vectorization
- explicit vectorization in OpenMP 4.0
- consider a few other programming considerations along the way

Summary

We need to embrace *explicit* vectorization in our programming.

How many of us here today…

have ever worried about vectorization for your application?

Shouldn’t we solve with better tools?

What is vectorization?

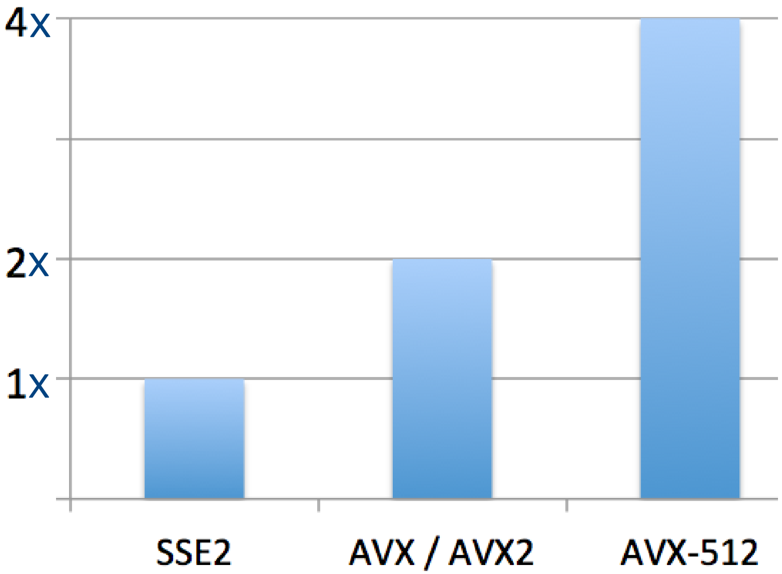
Could we just ignore it?

Vectors Instructions (SIMD instructions) Make things Faster

(that’s the premise)

Up to 4x Performance

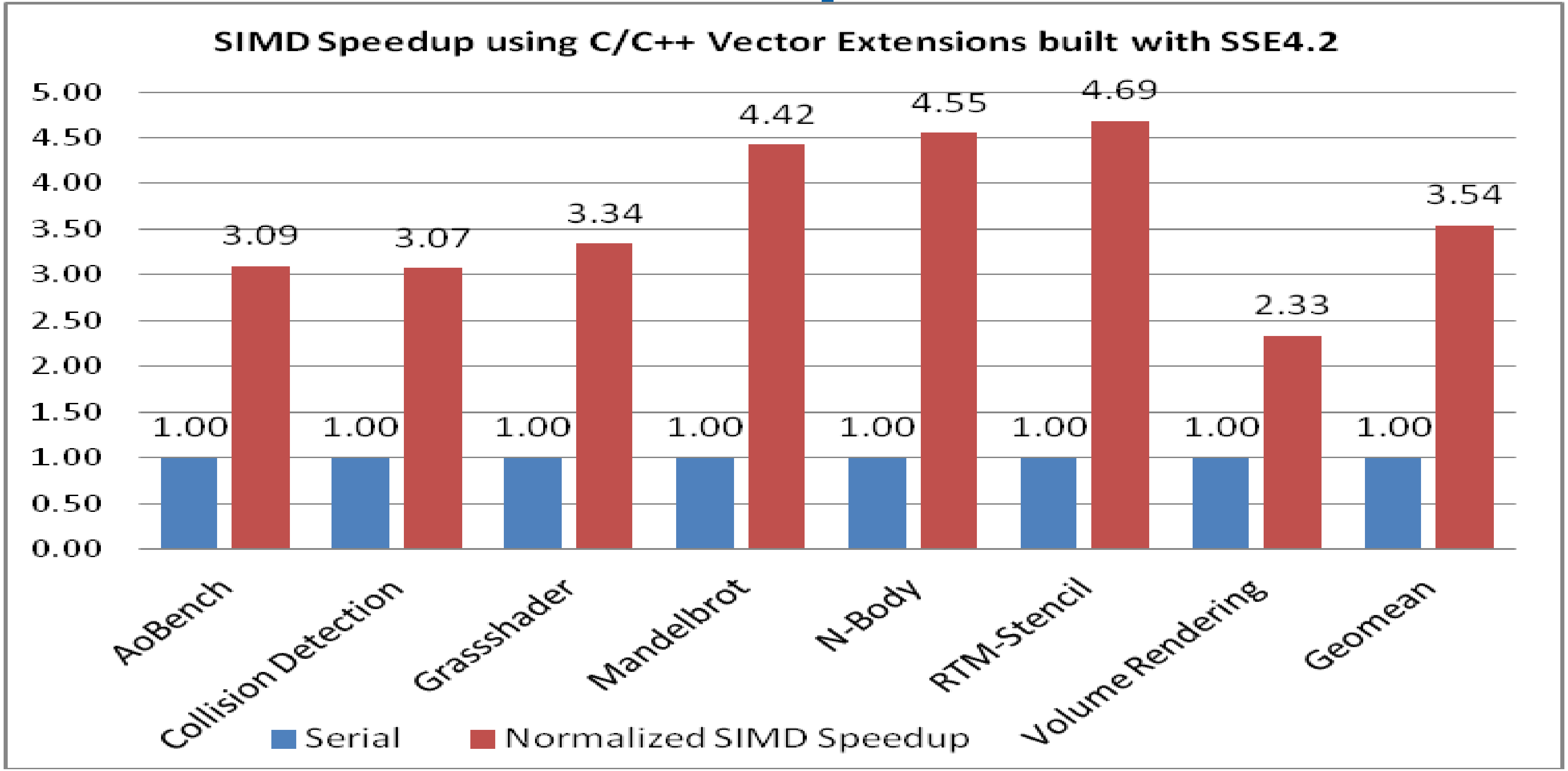
with Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Support



- Significant leap to 512-bit SIMD support for processors
- Intel® Compilers and Intel® Math Kernel Library include AVX-512 support
- Strong compatibility with AVX
- Added EVEX prefix enables additional functionality
- Appears first in future Intel® Xeon Phi™ coprocessor, code named Knights Landing

Higher performance for the most demanding computational tasks

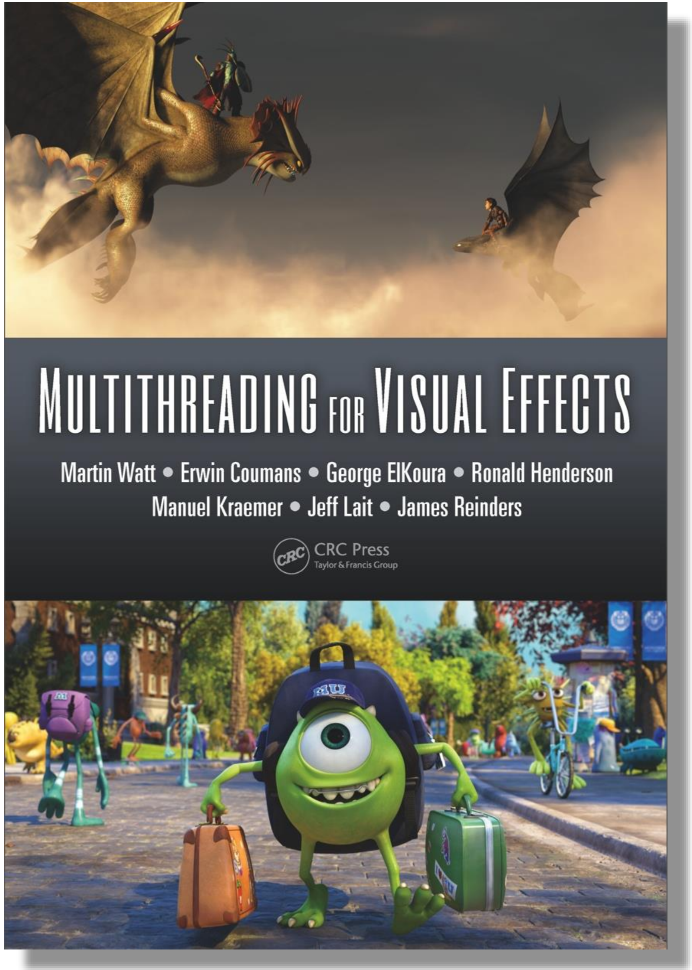
Performance with Explicit Vectorization



Configuration: Intel® Core™ i7 CPU X980 system (6 cores with Hyper-Threading On), running at 3.33GHz, with 4.0GB RAM, 12M smart cache, 64-bit Windows Server 2008 R2 Enterprise SP1. **For more information go to** http://www.intel.com/performance

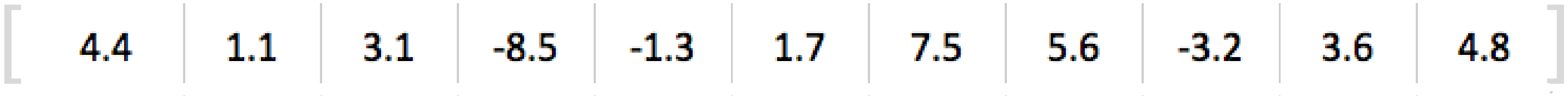
Parallel first

Vectorize second


MULTITHREADING FOR VISUAL EFFECTS
Martin Watt • Erwin Coumans • George ElKoura • Ronald Henderson
Manuel Kraemer • Jeff Lait • James Reinders
CRC Press
Taylor & Francis Group

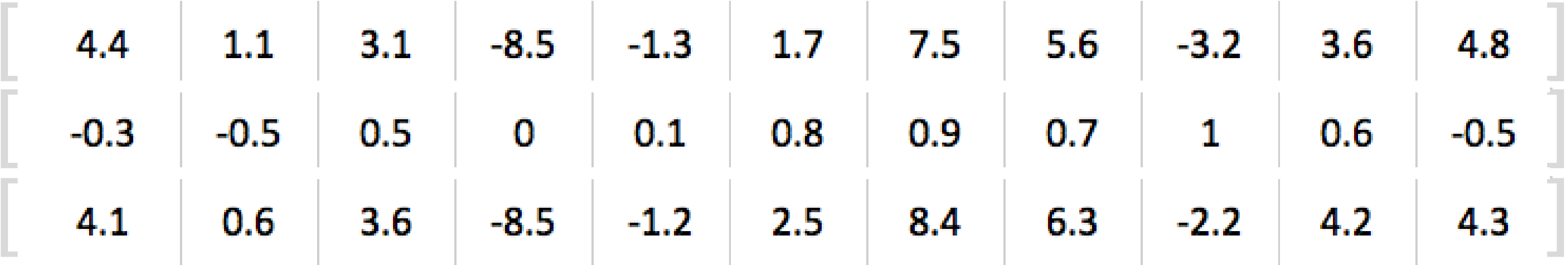
What is a Vector?

Vector of numbers



4.4	1.1	3.1	-8.5	-1.3	1.7	7.5	5.6	-3.2	3.6	4.8

Vector addition



	4.4	1.1	3.1	-8.5	-1.3	1.7	7.5	5.6	-3.2	3.6	4.8
+	-0.3	-0.5	0.5	0	0.1	0.8	0.9	0.7	1	0.6	-0.5
=	4.1	0.6	3.6	-8.5	-1.2	2.5	8.4	6.3	-2.2	4.2	4.3

…and Vector multiplication

	4.4	1.1	3.1	-8.5	-1.3	1.7	7.5	5.6	-3.2	3.6	4.8
+	-0.3	-0.5	0.5	0	0.1	0.8	0.9	0.7	1	0.6	-0.5
=	4.1	0.6	3.6	-8.5	-1.2	2.5	8.4	6.3	-2.2	4.2	4.3

	4.4	1.1	3.1	-8.5	-1.3	1.7	7.5	5.6	-3.2	3.6	4.8
×	-0.3	-0.5	0.5	0	0.1	0.8	0.9	0.7	1	0.6	-0.5
=	-1.32	-0.55	1.55	0	-0.13	1.36	6.75	3.92	-3.2	2.16	-2.4

An example

vector data operations: data operations done in parallel

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

vector data operations: data operations done in parallel

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

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

vector data operations:
data operations done in parallel

```
void v_add (float *c,
```

Loop:

1. LOADv4 a[i:i+3] -> Rva
2. LOADv4 b[i:i+3] -> Rvb
3. ADDv4 Rva, Rvb -> Rvc
4. STOREv4 Rvc -> c[i:i+3]
5. ADD i + 4 -> i

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

vector data operations: data

We call this “vectorization”

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

Loop:

1. LOADv4 a[i:i+3] -> Rva
2. LOADv4 b[i:i+3] -> Rvb
3. ADDv4 Rva, Rvb -> Rvc
4. STOREv4 Rvc -> c[i:i+3]
5. ADD i + 4 -> i

Loop:

1. LOAD a[i] -> Ra
2. LOAD b[i] -> Rb
3. ADD Ra, Rb -> Rc
4. STORE Rc -> c[i]
5. ADD i + 1 -> i

vector data operations:
data operations done in parallel

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

vector data operations:
data operations done in parallel

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

PROBLEM:
This LOOP is NOT LEGAL to (automatically) VECTORIZE in C / C++ (without more information).

Arrays *not* really in the language

Pointers are, evil pointers!

Choice 1: use a compiler switch for auto-vectorization

(and *hope* it vectorizes)

Choice 2:

give your compiler hints

(and *hope* it vectorizes)

C99 *restrict* keyword

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

IVDEP (ignore assumed vector dependencies)

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

Choice 3:
code explicitly for vectors

(mandatory vectorization)

OpenMP* 4.0: #pragma omp simd

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

OpenMP* 4.0: #pragma omp declare simd

```
#pragma omp declare simd
void v1_add (float *c,
        float *a,
        float *b)
{

       *c=*a+*b;
}
```

SIMD instruction intrinsics

```
void v_add (float *c,
            float *a,
            float *b)
{
   __m128* pSrc1 = (__m128*) a;
   __m128* pSrc2 = (__m128*) b;
   __m128* pDest = (__m128*) c;
   for (int i=0; i<= MAX/4; i++)
       *pDest++ = _mm_add_ps(*pSrc1++, *pSrc2++);
}
```

Hard coded to 4 wide !

array operations (Cilk™ Plus)

```
void v_add (float *c,
            float *a,
            float *b)
{
          c[0:MAX]=a[0:MAX]+b[0:MAX];
}
```

Challenge: long vector slices can cause cache issues; fix is to keep vector slices short.

Cilk™ Plus is supported in Intel compilers, and gcc (4.9).

vectorization solutions

1. auto-vectorization (use a compiler switch and hope it vectorizes)
   - sequential languages and practices gets in the way
2. give your compiler hints and hope it vectorizes
   - C99 restrict (implied in FORTRAN since 1956)
   - #pragma ivdep
3. code explicitly
   - OpenMP 4.0 #pragma omp simd
   - Cilk™ Plus array notations
   - SIMD instruction intrinsics
   - Kernels: OpenMP 4.0 #pragma omp declare simd; OpenCL; CUDA kernel functions

vectorization solutions

1. auto-vectorization (use a compiler switch and hope it vectorizes)
   - sequential languages and practices gets in the way
2. give your compiler hints and hope it vectorizes
   - C99 restrict (implied in FORTRAN since 1956)
   - #pragma ivdep
3. code explicitly
   - OpenMP 4.0 #pragma omp simd
   - Cilk™ Plus array notations
   - SIMD instruction intrinsics
   - Kernels: OpenMP 4.0 #pragma omp declare simd; OpenCL; CUDA kernel functions

Best at being
Reliable, predictable and portable

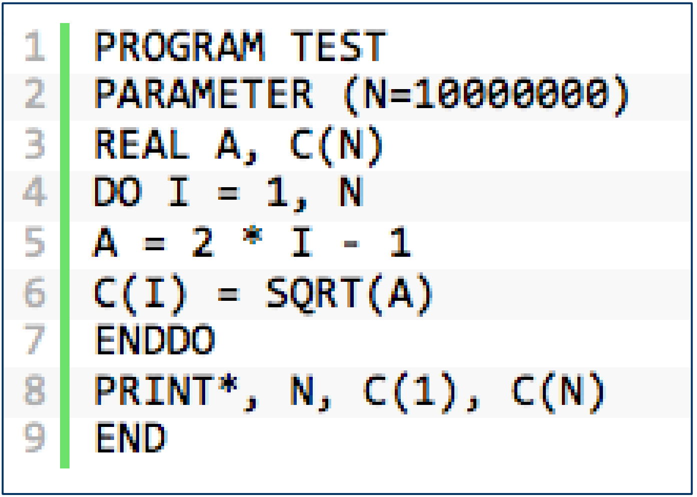
Explicit parallelism

parallelization

Try auto-parallel capability

-parallel (Linux* or OS X*)

-Qparallel (Windows*)



```
PROGRAM TEST
PARAMETER (N=10000000)
REAL A, C(N)
DO I = 1, N
A = 2 * I - 1
C(I) = SQRT(A)
ENDDO
PRINT*, N, C(1), C(N)
END
```

Or explicitly use…

Fortran directive (!DIR$ PARALLEL)

C pragma (#pragma parallel)

Intel® Threading Building Blocks (TBB)

parallelization

Try auto-parallel capability:

-parallel (Linux or OS X*)

-Qparallel (Windows)

Best at being
Reliable, predictable and portable

Or explicitly use…

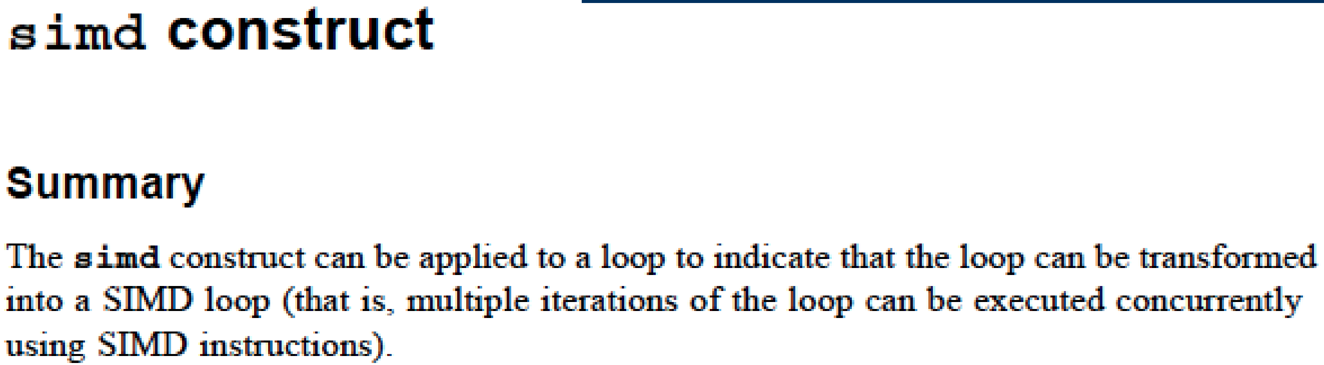
OpenMP

Intel® Threading Building Blocks (T

```
c$OMP PARALLEL DO
 DO I=1,N B(I) = (A(I) + A(I-1)) / 2.0
 END DO
c$OMP END PARALLEL DO
```

OpenMP 4.0

Based on a proposal from Intel based on customer success with the
Intel® Cilk™ Plus features in Intel compilers.



`simd` construct

Summary

The **`simd`** construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

OpenMP 4.0

Based on a proposal from Intel based on customer success with the Intel® Cilk™ Plus features in Intel compilers.

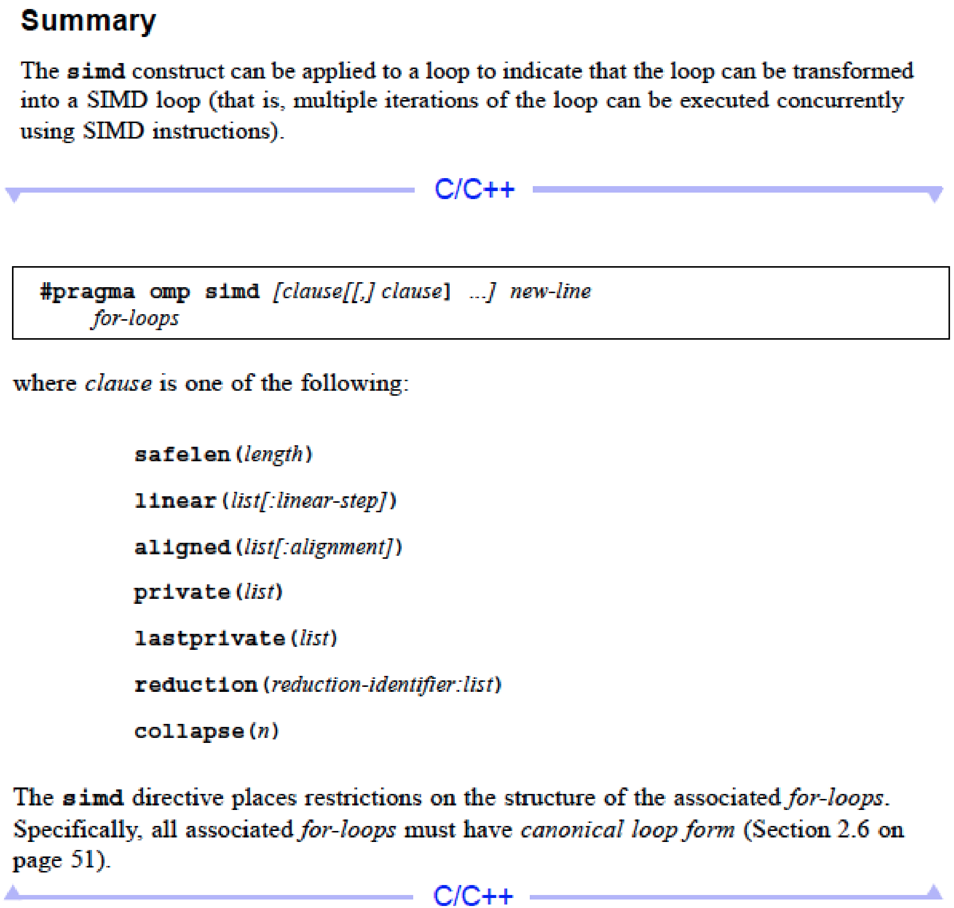
simd construct

```
#pragma omp simd reduction(+:val) reduction(+:val2)
  for(int pos = 0; pos < RAND_N; pos++) {
     float callValue=
              expectedCall(Sval,Xval,MuByT,VBySqrtT,l_Random[pos]);
     val  += callValue;
     val2 += callValue * callValue;
}
```

simd construct

(OpenMP 4.0)

YES – VECTORIZE THIS !!!



Summary

The **simd** construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

C/C++

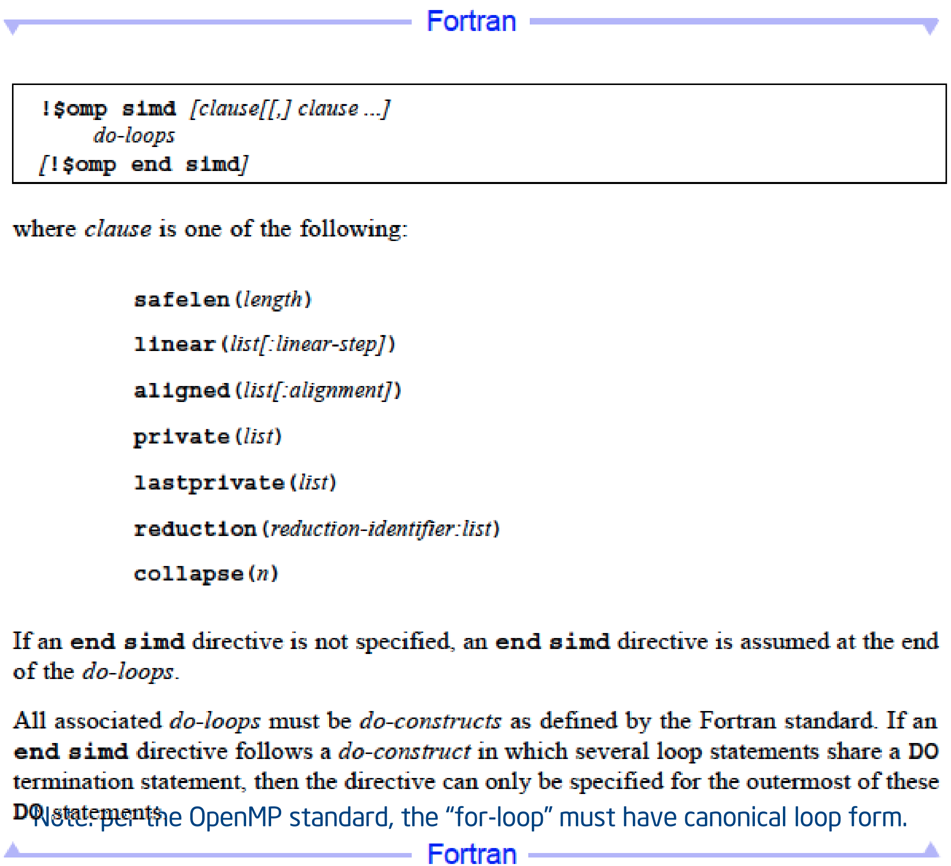
```
#pragma omp simd [clause[[,] clause] ...] new-line
    for-loops
```

where *clause* is one of the following:

- **safelen**(*length*)
- **linear**(*list[:linear-step]*)
- **aligned**(*list[:alignment]*)
- **private**(*list*)
- **lastprivate**(*list*)
- **reduction**(*reduction-identifier:list*)
- **collapse**(*n*)

The **simd** directive places restrictions on the structure of the associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (Section 2.6 on page 51).

C/C++



Fortran

```
!$omp simd [clause[[,] clause ...]
    do-loops
[!$omp end simd]
```

where *clause* is one of the following:

- **safelen**(*length*)
- **linear**(*list[:linear-step]*)
- **aligned**(*list[:alignment]*)
- **private**(*list*)
- **lastprivate**(*list*)
- **reduction**(*reduction-identifier:list*)
- **collapse**(*n*)

If an **end simd** directive is not specified, an **end simd** directive is assumed at the end of the *do-loops*.

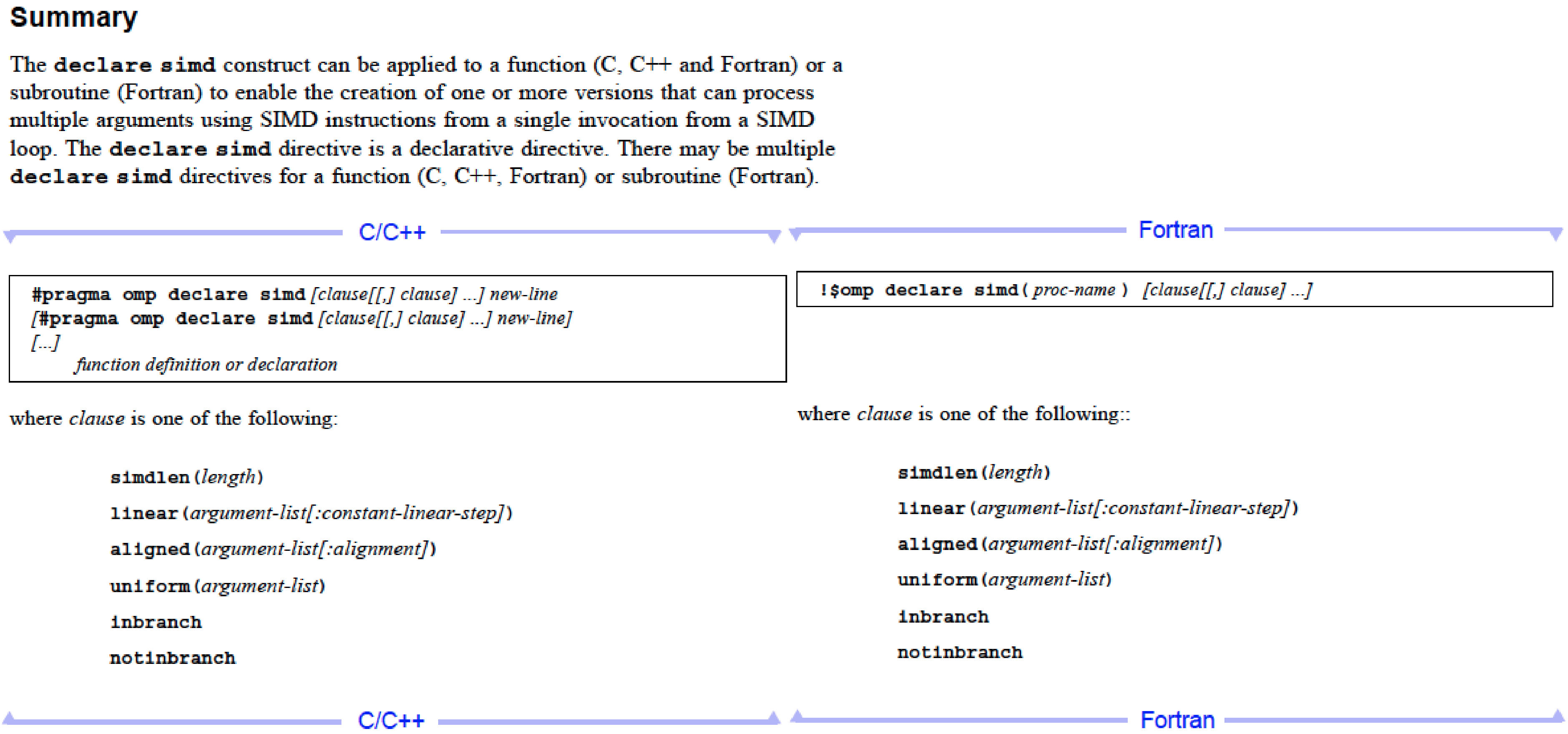
All associated *do-loops* must be *do-constructs* as defined by the Fortran standard. If an **end simd** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements.

Fortran

Note: per the OpenMP standard, the "for-loop" must have canonical loop form.

declare simd construct (OpenMP 4.0)

Make VECTOR versions of this function.



Summary

The **declare simd** construct can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop. The **declare simd** directive is a declarative directive. There may be multiple **declare simd** directives for a function (C, C++, Fortran) or subroutine (Fortran).

C/C++

```
#pragma omp declare simd [clause[[,] clause] ...] new-line
[#pragma omp declare simd [clause[[,] clause] ...] new-line]
[...]
      function definition or declaration
```

where *clause* is one of the following:

simdlen (*length*)

linear (*argument-list[:constant-linear-step]*)

aligned (*argument-list[:alignment]*)

uniform (*argument-list*)

inbranch

notinbranch

C/C++

Fortran

```
!$omp declare simd ( proc-name )  [clause[[,] clause] ...]
```

where *clause* is one of the following::

simdlen (*length*)

linear (*argument-list[:constant-linear-step]*)

aligned (*argument-list[:alignment]*)

uniform (*argument-list*)

inbranch

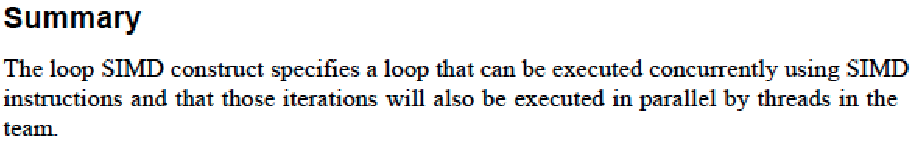
notinbranch

Fortran

Parallelize and Vectorize.

Loop SIMD construct

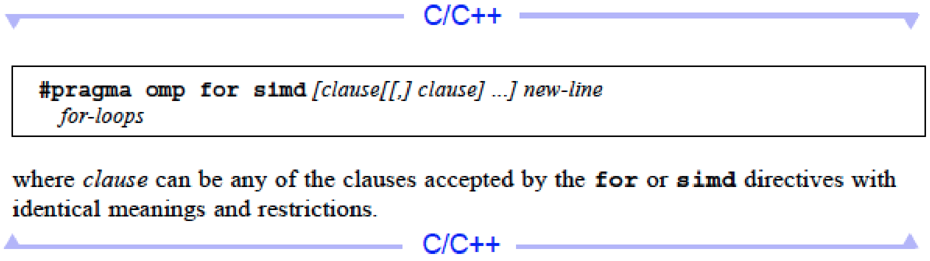
(OpenMP 4.0)



Summary

The loop SIMD construct specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel by threads in the team.

Syntax

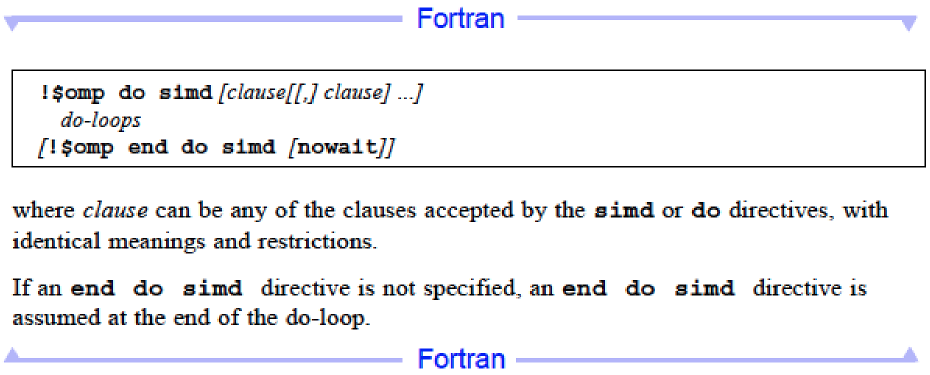


C/C++

```
#pragma omp for simd [clause[[,] clause] ...] new-line
   for-loops
```

where *clause* can be any of the clauses accepted by the **for** or **simd** directives with identical meanings and restrictions.

C/C++



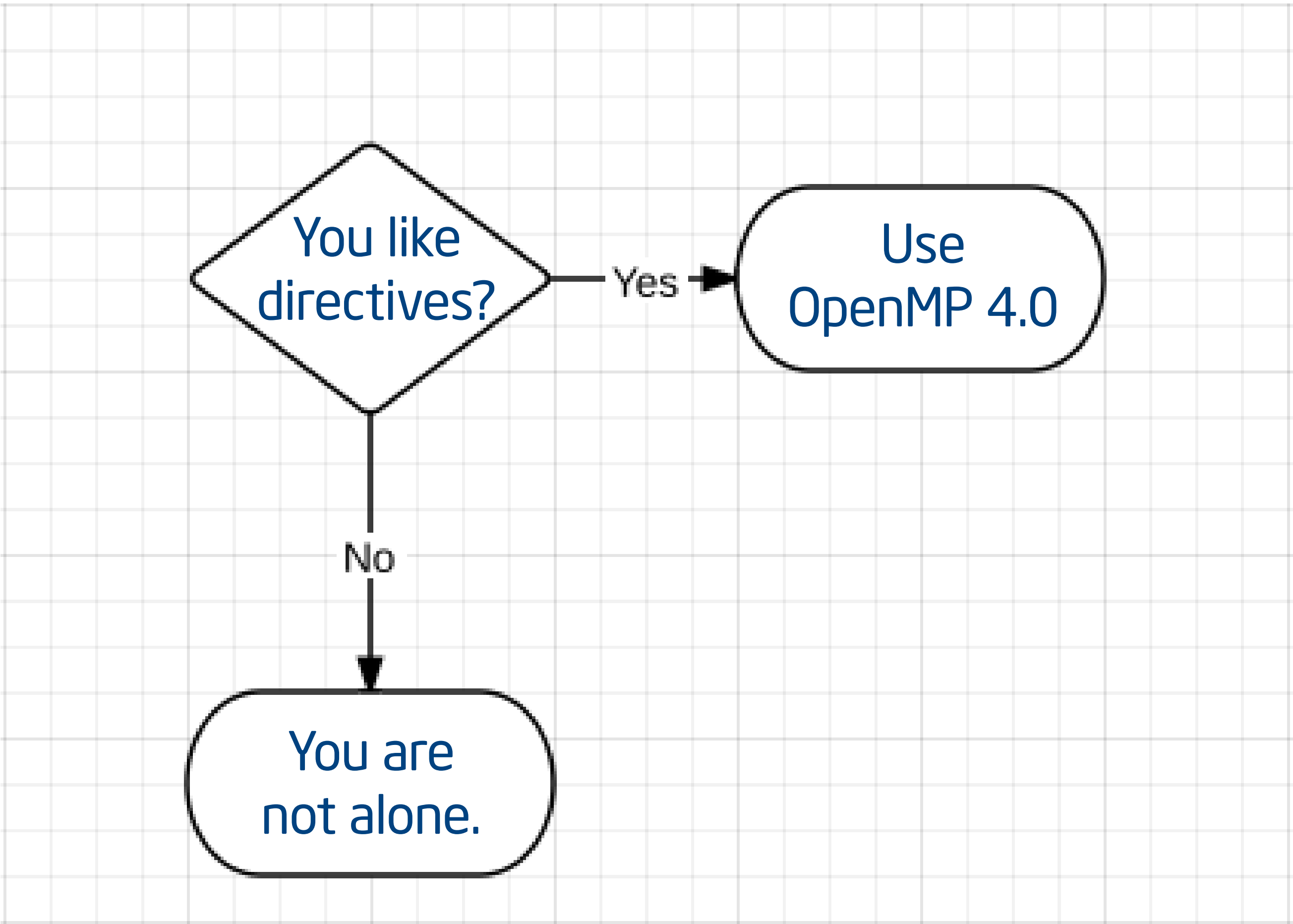
Fortran

```
!$omp do simd [clause[[,] clause] ...]
   do-loops
[!$omp end do simd [nowait]]
```

where *clause* can be any of the clauses accepted by the **simd** or **do** directives, with identical meanings and restrictions.

If an **end do simd** directive is not specified, an **end do simd** directive is assumed at the end of the do-loop.

Fortran



You like directives?

Yes

Use OpenMP 4.0

No

You are not alone.

for your consideration:

Intel 15.0 Compilers (in beta now) support **keywords** as an alternative

- Keyword versions of SIMD pragmas added:
  `_Simd, _Safelen, _Reduction`
- `__intel_simd_lane()` intrinsic for SIMD enabled functions

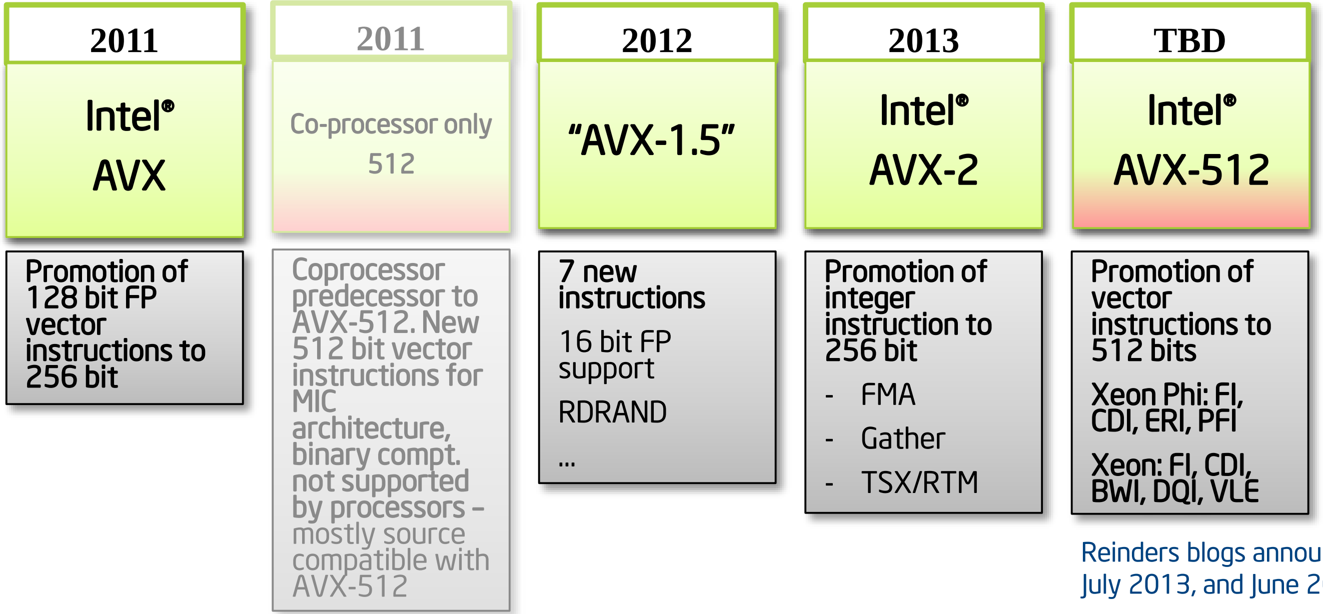
Keywords / library interfaces being discussed for SIMD constructs in C and C++ standards

History of Intel vector instructions

Intel Instruction Set Vector Extensions from 1997-2008

1997	1998	1999	2004	2006	2007	2008
Intel® MMX™ technology	Intel® SSE	Intel® SSE2	Intel® SSE3	Intel® SSSE3	Intel® SSE4.1	Intel® SSE4.2
57 new instructions	70 new instructions	144 new instructions	13 new instructions	32 new instructions	47 new instructions	7 new instructions
64 bits	128 bits	128 bits	128 bits	128 bits	128 bits	128 bits
Overload FP stack	4 single-precision vector FP	2 double-precision vector FP	FP vector calculation	enhanced packed integer calculation	packed integer calculation & conversion	string (XML) processing
Integer only	scalar FP instructions	8/16/32/64 vector integer	x87 integer conversion		better vectorization by compiler	POP-Count
media extensions	cacheability instructions	128-bit integer	128-bit integer unaligned load		load with streaming hint	CRC32
	control & conversion instructions	memory & power management	thread sync.			
	media extensions					

Intel Instruction Set Vector Extensions since 2011



2011

Intel® AVX

Promotion of 128 bit FP vector instructions to 256 bit

2011

Co-processor only 512

Coprocessor predecessor to AVX-512. New 512 bit vector instructions for MIC architecture, binary compt. not supported by processors – mostly source compatible with AVX-512

2012

“AVX-1.5”

7 new instructions

16 bit FP support

RDRAND

…

2013

Intel® AVX-2

Promotion of integer instruction to 256 bit

- FMA
- Gather
- TSX/RTM

TBD

Intel® AVX-512

Promotion of vector instructions to 512 bits

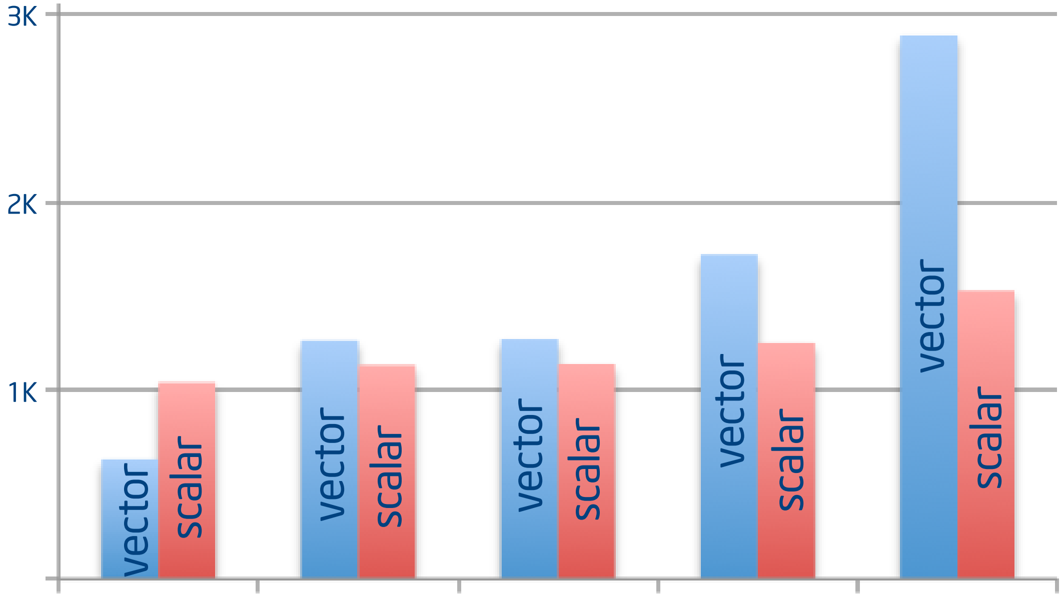
Xeon Phi: FI, CDI, ERI, PFI

Xeon: FI, CDI, BWI, DQI, VLE

Reinders blogs announced – July 2013, and June 2014.

		width	Int.	SP	DP
1997	MMX	64	✓		
1999	SSE	128	✓	✓(x4)	
2001	SSE2	128	✓	✓	✓(x2)
2004	SSE3	128	✓	✓	✓
2006	SSSE 3	128	✓	✓	✓
2006	SSE 4.1	128	✓	✓	✓
2008	SSE 4.2	128	✓	✓	✓
2011	AVX	256	✓	✓(x8)	✓(x4)
2013	AVX2	256	✓	✓	✓
future	AVX-512	512	✓	✓(x16)	✓(x8)

Growth is in vector instructions

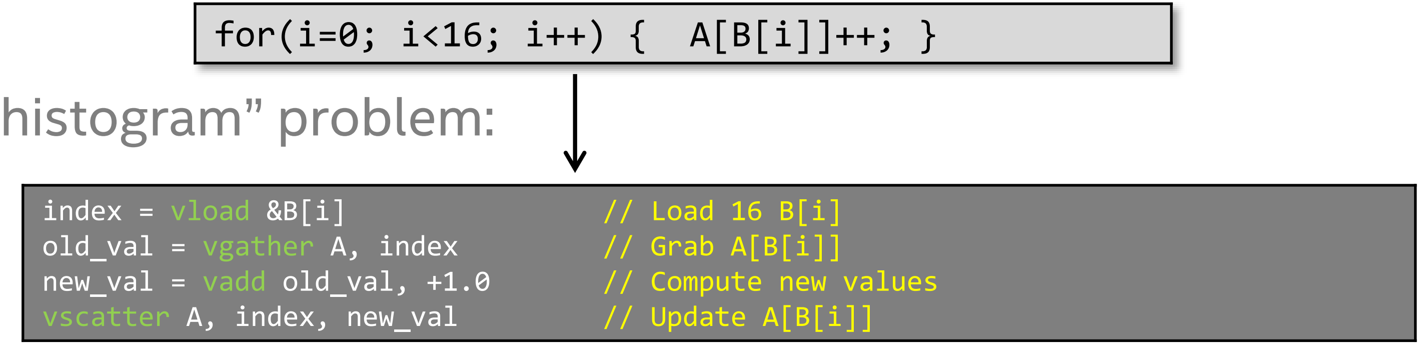


Disclaimer: Counting/attributing instructions is in inexact science. The exact numbers are easily debated, the trend is quite real regardless.

Motivation for AVX-512 Conflict Detection

Sparse computations are common in HPC, but hard to vectorize due to race conditions

Consider the “histogram” problem:



```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

```
index = vload &B[i]                 // Load 16 B[i]
old_val = vgather A, index          // Grab A[B[i]]
new_val = vadd old_val, +1.0        // Compute new values
vscatter A, index, new_val          // Update A[B[i]]
```

Motivation for AVX-512 Conflict Detection

Sparse computations are common in HPC, but hard to vectorize due to race conditions

```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

Consider the “histogram” problem:

```
index = vload &B[i]                 // Load 16 B[i]
old_val = vgather A, index          // Grab A[B[i]]
new_val = vadd old_val, +1.0        // Compute new values
vscatter A, index, new_val          // Update A[B[i]]
```

- Code above is wrong if any values within B[i] are duplicated
  - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts

Conflict Detection Instructions in AVX-512 *improve vectorization!*

VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes

- Allows to generate a mask with a subset of elements that are guaranteed to be conflict free
- The computation loop can be re-executed with the remaining elements until all the indexes have been operated upon

CDI instr.
VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem
VPBROADCASTM{W2D,B2Q} zmm1, k2
VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem
VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

```
index = vload &B[i]                             // Load 16 B[i]
pending_elem = 0xFFFF;                          // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index       // Grab A[B[i]]
   new_val = vadd old_val, +1.0                 // Compute new values
   vscatter A {curr_elem}, index, new_val       // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem      // remove done idx
} while (pending_elem)
```

for illustration: this not even the fastest version

-vec-report

“Dear compiler, did you vectorize my loop?” We heard your feedback……

-vec-report output was hard to understand;

Messages were too cryptic to understand;

Information about one loop showing up at many places of report;

Was easy to be confused about multiple versions of one loop created by the compiler

We couldn’t do everything you asked, but here are the improvements made for 15.0 compiler.

Expect more changes to come, during beta and in future versions.

Optimization Report Redesign

- Old functionality implemented under `-opt-report`, `-vec-report`, `-openmp-report`, `-par-report` replaced by unified `-opt-report` compiler options
  - `[vec,openmp,par]-report` options deprecated and map to equivalent opt-report-phase

- Can still select phase with `-opt-report-phase` option. For example, to only get vectorization reports, use `-opt-report-phase=vec`

- Output now defaults to a `<name>.optrpt` file where `<name>` corresponds to the output object name. This can be changed with `-opt-report-file=[<name>|stdout|stderr]`

- Windows*: `/Qopt-report, /Qopt-report-phase=<phase> etc.`
  - Optimization report integration with Microsoft* Visual Studio planned to appear in beta update 1

Summary

We need to embrace explicit vectorization in our programming.

Vectorization today uses

“Not your father’s vectorizer”

Make a Date with the "88"
LOWEST-PRICED "ROCKET"-HYDRA-MATIC CAR
FUTURAMIC
OLDSMOBILE
POWER PACKAGE FOR PLUS PERFORMANCE –
"ROCKET" ENGINE AND HYDRA-MATIC DRIVE !

You won't find this Oldsmobile tooling through anybody's family album.

For the past five years, this all new Cutlass Supreme has existed only in the minds of an elite cadre of automotive engineers. And on a million miles of test track.

It resists the wind with an affinity that would do a needle proud.

And sticks to the road like gum.

At 0.297 its drag coefficient hovers somewhere between exceptional and whoopee.

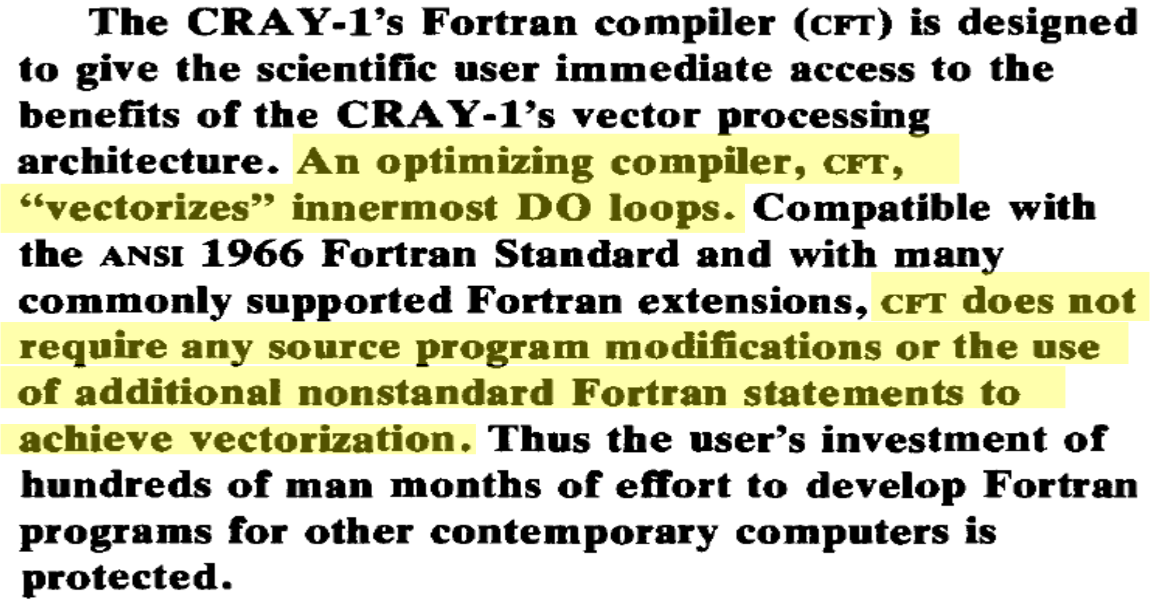
But beneath its sleek, double-sided, galvanized exterior, the Cutlass Supreme is pure muscle. Its body strength and rigidity are the basis of one of the world's most totally developed suspension systems.

As you'd expect on such a world-class automobile, technological achievements such as four-wheel independent suspension, front wheel drive, four wheel disc brakes, and a 2.8 liter multiport fuel injected V6 are, of course, standard.

For more information on this remarkable vehicle, send for a free catalog. Write: Oldsmobile Cutlass Supreme Catalog, P.O. Box 14238, Lansing, Michigan 48901.

Vectorization solved in 1978?

Communications of the ACM
The CRAY-1 computer system
By Richard M. Russell
Cray Research, Inc., Minneapolis, MN
Communications of the ACM,
January 1978 (Vol. 21 No. 1), Pages 63-72



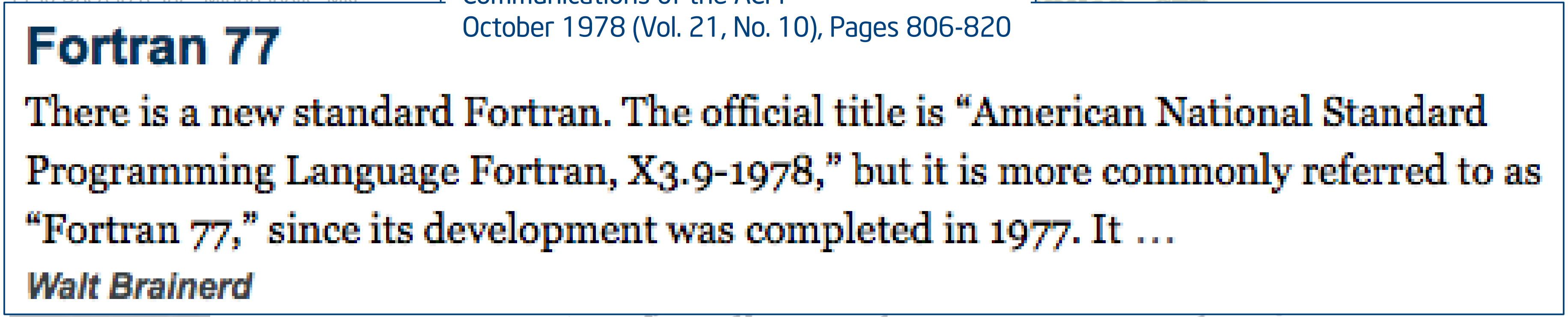
The CRAY-1's Fortran compiler (CFT) is designed to give the scientific user immediate access to the benefits of the CRAY-1's vector processing architecture. An optimizing compiler, CFT, "vectorizes" innermost DO loops. Compatible with the ANSI 1966 Fortran Standard and with many commonly supported Fortran extensions, CFT does not require any source program modifications or the use of additional nonstandard Fortran statements to achieve vectorization. Thus the user's investment of hundreds of man months of effort to develop Fortran programs for other contemporary computers is protected.

Vectorization solved in 1978?

Communications of the ACM
The CRAY-1 computer system
By Richard M. Russell
Cray Research, Inc., Minneapolis, MN

The CRAY-1's Fortran compiler (CFT) is designed to give the scientific user immediate access to the benefits of the CRAY-1's vector processing

Communications of the ACM
October 1978 (Vol. 21, No. 10), Pages 806-820



Fortran 77

There is a new standard Fortran. The official title is "American National Standard Programming Language Fortran, X3.9-1978," but it is more commonly referred to as "Fortran 77," since its development was completed in 1977. It …

Walt Brainerd

programs for other contemporary computers is protected.

Livermore loop #1

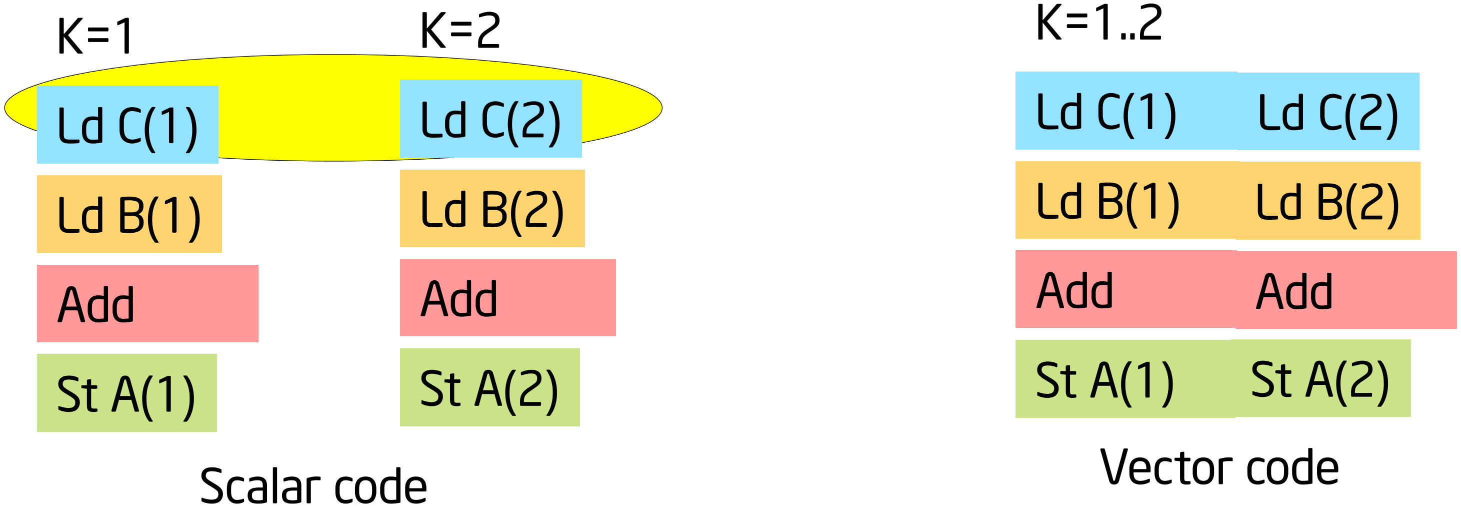
```
c
c***********************************************************************
c***  KERNEL 1      HYDRO FRAGMENT
c***********************************************************************
c
cdir$ ivdep
 1001    DO 1 k = 1,n
    1       X(k)= Q + Y(k) * (R * ZX(k+10) + T * ZX(k+11))
c
```

Vector code generation straightforward
Emphasis on analysis and disambiguation

It's messy today

Vectorization yesterday

```
        DO 1 k = 1,n
1       A(k) = B(k) + C(k)
```



K=1

Ld C(1)

Ld B(1)

Add

St A(1)

K=2

Ld C(2)

Ld B(2)

Add

St A(2)

Scalar code

K=1..2

Ld C(1) Ld C(2)

Ld B(1) Ld B(2)

Add Add

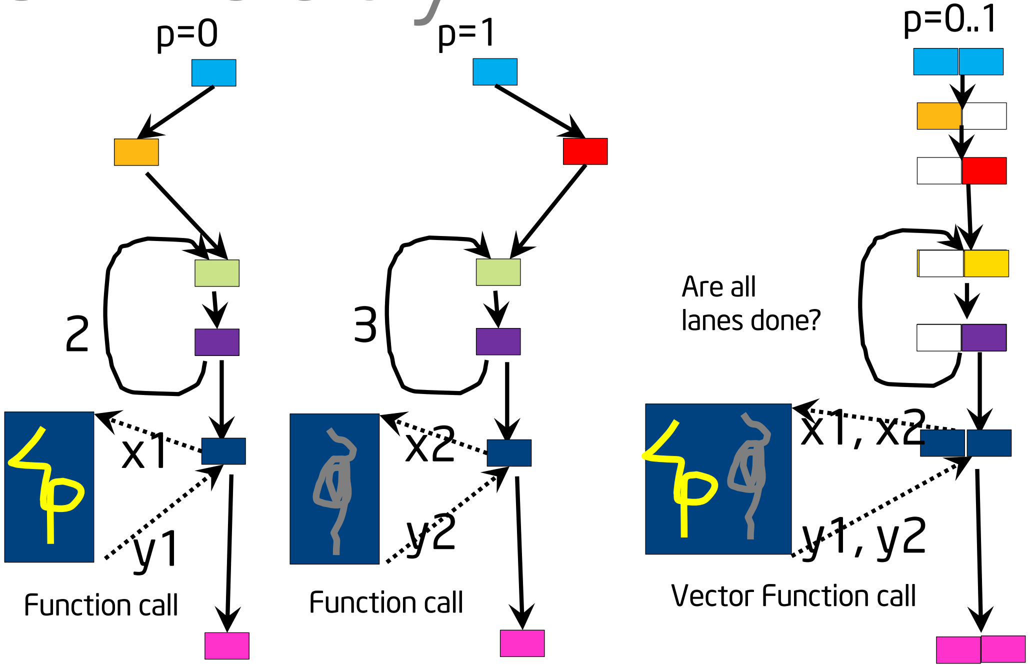
St A(1) St A(2)

Vector code

Vector code generation was straightforward
Emphasis on analysis and disambiguation

Vectorization today

```
#pragma omp simd reduction(+:....)
for(p=0; p<N; p++) {
   // Blue work
   if(...) {
      // Green work
   } else {
      // Red work
   }
   while(...) {
      // Gold work
      // Purple work
   }
   y = foo (x);
   Pink work
}
```



Two fundamental problems
- Data divergence
- Control divergence

Vector code generation has become a more difficult problem
Increasing need for user guided explicit vectorization
Explicit vectorization maps threaded execution to simd hardware

```
#pragma omp simd
for (x = 0; x < w; x++)  {
     for (v = 0; v < nsubsamples; v++) {
          for (u = 0; u < nsubsamples; u++) {
          float px = (x + (u / (float)nsubsamples) - (w / 2.0f)) / (w / 2.0f);
          Ray ray;  Isect isect;
              ....
              ray.dir.x = px;
                    ....
                    vnormalize(&(ray.dir));
                    ......
                    ray_sphere_intersect(&isect, &ray, &spheres[0]);
                    ......
                    ray_plane_intersect (&isect, &ray, &plane);

                    if (isect.hit) {
                        vec col;
                        ambient_occlusion_simd(&col, &isect);
                        fimg[3 * (y * w + x) + 0] += col.x;
              ........
               }
           }
      }
}
```

Loops

Function calls

Conditionals

Conditional Function calls

Motivational Example

```
//foo.c
float in_vals[];
for(int x = 0; x < Width; ++x) {
    count[x] = lednam(in_vals[x]);
}
```

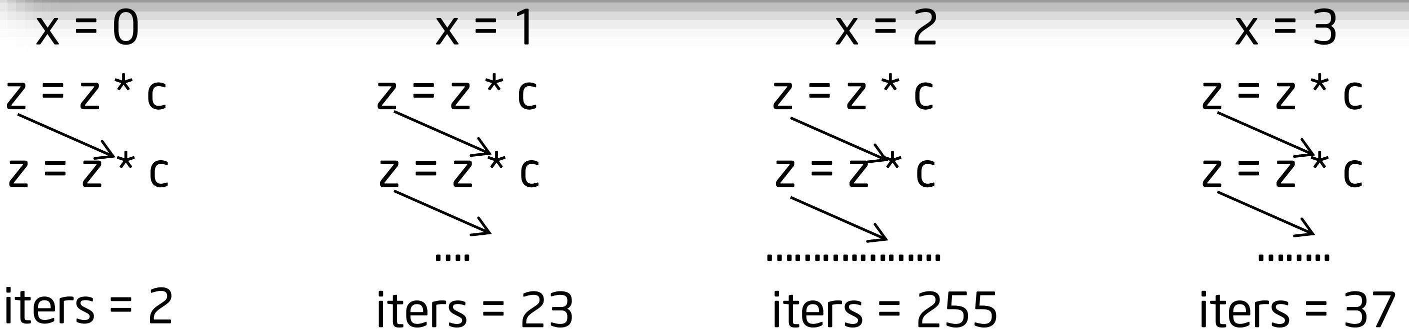
```
//bar.c
int lednam(float c)
{   // Compute n >= 0 such that c^n > LIMIT
    float z = 1.0f;
    int iters = 0;
    while (z < LIMIT) {
        z = z * c; iters++;
    }
    return iters;
}
```

What are the simplest changes required for the program to utilize today’s multicore and simd hardware?

```
float in_vals[];

for(int x = 0; x < Width; ++x) {
    count[x] = lednam(in_vals[x]);
}
```

```
#pragma omp declare simd
int lednam(float c)
{   // Compute n >= 0 such that c^n > LIMIT
    float z = 1.0f; int iters = 0;
    while (z < LIMIT) {
        z = z * c; iters++;
    }
    return iters;
}
```



x = 0

z = z * c

z = z * c

iters = 2

x = 1

z = z * c

z = z * c

....

iters = 23

x = 2

z = z * c

z = z * c

....

iters = 255

x = 3

z = z * c

z = z * c

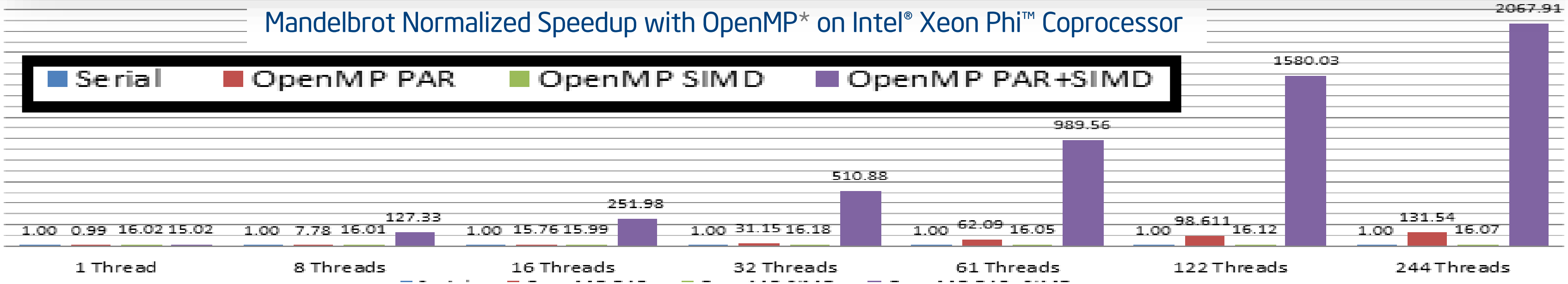
....

iters = 37

Mandelbrot

```
#pragma omp parallel for
for (int y = 0; y < ImageHeight; ++y) {
   #pragma omp simd
   for (int x = 0; x < ImageWidth; ++x) {
      count[y][x] = mandel(in_vals[y][x]);
   }
}
```

```
#pragma omp declare simd
int mandel(fcomplex c)
{  // Computes number of iterations for c to escape
   fcomplex z = c;
   for (int iters=0; (cabsf(z) < 2.0f) && (iters < LIMIT); iters++) {
        z = z * z + c;
   }
   return iters;
}
```


Mandelbrot Normalized Speedup with OpenMP* on Intel® Xeon Phi™ Coprocessor

Summary

We need to embrace explicit vectorization in our programming.

But, generally use parallelism first (tasks, threads, MPI, etc.)

Questions?

james.r.reinders@intel.com

James Reinders. Parallel Programming Evangelist. Intel.

James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), and Multithreading for Visual Effects (A K Peters/CRC Press, 2014). James is working on a project to publish a book of programming examples featuring Intel Xeon Phi programming scheduled to be published in late 2014.

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804