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
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
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

SIMD Speedup using C/C++ Vector Extensions built with SSE4.2

<table>
<thead>
<tr>
<th>Application</th>
<th>Serial</th>
<th>Normalized SIMD Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>AoBench</td>
<td>1.00</td>
<td>3.09</td>
</tr>
<tr>
<td>Collision Detection</td>
<td>1.00</td>
<td>3.07</td>
</tr>
<tr>
<td>Grasshopper</td>
<td>1.00</td>
<td>3.34</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1.00</td>
<td>4.42</td>
</tr>
<tr>
<td>N-Body</td>
<td>1.00</td>
<td>4.55</td>
</tr>
<tr>
<td>RTM-Stencil</td>
<td>1.00</td>
<td>4.69</td>
</tr>
<tr>
<td>Volume Rendering</td>
<td>1.00</td>
<td>2.33</td>
</tr>
<tr>
<td>Geonnean</td>
<td>1.00</td>
<td>3.54</td>
</tr>
</tbody>
</table>

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

© 2014, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Inside, Cilk, VTune, Xeon, and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. Other names and brands may be claimed as the property of others.
Parallel first

Vectorize second
What is a Vector?
Vector of numbers

\[
\begin{bmatrix}
4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8
\end{bmatrix}
\]
### Vector addition

\[
\begin{bmatrix}
4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\
-0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\
\end{bmatrix}
\]

\[+
\]

\[
\begin{bmatrix}
4.1 & 0.6 & 3.6 & -8.5 & -1.2 & 2.5 & 8.4 & 6.3 & -2.2 & 4.2 & 4.3 \\
\end{bmatrix}
\]

\[=
\]
...and Vector multiplication

\[
\begin{bmatrix}
4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\
-0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
4.1 & 0.6 & 3.6 & -8.5 & -1.2 & 2.5 & 8.4 & 6.3 & -2.2 & 4.2 & 4.3 \\
\end{bmatrix}
\]
An example
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

```c
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

```c
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

We call this “vectorization”

```c
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

```c
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

```c
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

```c
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];
}
```
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

```c
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`

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

    *c=*a+*b;
}
```
SIMD instruction intrinsics

```c
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!
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 arraynotations
   ▪ 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*)

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 (TBB)

```c
#pragma omp parallel
  do i=1,n
    b(i) = (a(i) + a(i-1)) / 2.0
  end do
```
```c
#pragma 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.

```c
#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;
}
```
**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).

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).

**Fortran**

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

where `clause` is one of the following:

- `saeflen(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 determination statement, then the directive can only be specified for the outermost of these `do` statements.

Note per the OpenMP standard, the “for-loop” must have canonical loop form.

**C/C++**

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

where `clause` is one of the following:

- `saeflen(length)`
- `linear(list:linear-step)`
- `aligned(list:alignment)`
- `private(list)`
- `lastprivate(list)`
- `reduction(reduction-identifier:list)`
- `collapse(n)`

© 2014, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Inside, Cilk, VTune, Xeon, and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.
**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++

```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`

### Fortran

```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`
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] ...] nowait
```

Fortran

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

```fortran
!$omp do simd [nowait]
```

where `clause` can be any of the clauses accepted by the `for` or `simd` 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.
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
<table>
<thead>
<tr>
<th>Year</th>
<th>Instruction Set</th>
<th>New Instructions</th>
<th>Bits</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>Intel® MMX™ technology</td>
<td>57</td>
<td>64</td>
<td>Overload FP stack, Integer only media extensions</td>
</tr>
<tr>
<td>1998</td>
<td>Intel® SSE</td>
<td>70</td>
<td>128</td>
<td>4 single-precision vector FP, scalar FP instructions, cacheability instructions, control &amp; conversion instructions, media extensions</td>
</tr>
<tr>
<td>1999</td>
<td>Intel® SSE2</td>
<td>144</td>
<td>128</td>
<td>2 double-precision vector FP, 8/16/32/64 vector integer, 128-bit integer memory &amp; power management</td>
</tr>
<tr>
<td>2004</td>
<td>Intel® SSE3</td>
<td>13</td>
<td>128</td>
<td>FP vector calculation, x87 integer conversion, 128-bit integer unaligned load, thread sync.</td>
</tr>
<tr>
<td>2006</td>
<td>Intel® SSSE3</td>
<td>32</td>
<td>128</td>
<td>Enhanced packed integer calculation</td>
</tr>
<tr>
<td>2007</td>
<td>Intel® SSE4.1</td>
<td>47</td>
<td>128</td>
<td>Packed integer calculation &amp; conversion, better vectorization by compiler, load with streaming hint</td>
</tr>
<tr>
<td>2008</td>
<td>Intel® SSE4.2</td>
<td>7</td>
<td>128</td>
<td>String (XML) processing, POP-Count, CRC32</td>
</tr>
</tbody>
</table>
## Intel Instruction Set Vector Extensions since 2011

<table>
<thead>
<tr>
<th>Year</th>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Intel® AVX</td>
<td>Promotion of 128 bit FP vector instructions to 256 bit</td>
</tr>
<tr>
<td>2011</td>
<td>Co-processor only 512</td>
<td>Co-processor predecessor to AVX-512. New 512 bit vector instructions for MIC architecture, binary compat. not supported by processors - mostly source compatible with AVX-512</td>
</tr>
<tr>
<td>2012</td>
<td>“AVX-1.5”</td>
<td>7 new instructions: 16 bit FP support, RDRAND, ...</td>
</tr>
<tr>
<td>2013</td>
<td>Intel® AVX-2</td>
<td>Promotion of integer instruction to 256 bit: FMA, Gather, TSX/RTM</td>
</tr>
<tr>
<td>TBD</td>
<td>Intel® AVX-512</td>
<td>Promotion of vector instructions to 512 bits: Xeon Phi: Fl, CDI, ERI, PFI; Xeon: Fl, CDI, BWI, DQI, VLE</td>
</tr>
</tbody>
</table>

Reinders blogs announced - July 2013, and June 2014.
<table>
<thead>
<tr>
<th>Year</th>
<th>Instruction Set</th>
<th>Width</th>
<th>Int.</th>
<th>SP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>MMX</td>
<td>64</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>SSE</td>
<td>128</td>
<td>✓</td>
<td>✓</td>
<td>(x4)</td>
</tr>
<tr>
<td>2001</td>
<td>SSE2</td>
<td>128</td>
<td>✓</td>
<td>✓</td>
<td>(x2)</td>
</tr>
<tr>
<td>2004</td>
<td>SSE3</td>
<td>128</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2006</td>
<td>SSSE 3</td>
<td>128</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2006</td>
<td>SSE 4.1</td>
<td>128</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2008</td>
<td>SSE 4.2</td>
<td>128</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2011</td>
<td>AVX</td>
<td>256</td>
<td>✓</td>
<td>✓</td>
<td>(x8) (x4)</td>
</tr>
<tr>
<td>2013</td>
<td>AVX2</td>
<td>256</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>future</td>
<td>AVX-512</td>
<td>512</td>
<td>✓</td>
<td>✓</td>
<td>(x16) (x8)</td>
</tr>
</tbody>
</table>
Growth is in vector instructions

Disclaimer: Counting/attributing instructions is 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]]
```

• Code above is wrong if any values within B[i] are duplicated
  − Only one update from the repeated index would be registered!
Motivation for AVX-512 Conflict Detection

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

Consider the “histogram” problem:

```c
for(i=0; i<16; i++) { 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

```
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

  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”
Vectorization solved in 1978?

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’s Fortran compiler (CF7) is designed to give the scientific user immediate access to the benefits of the CRAY-1’s vector processing capability.

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 ...
Livermore loop #1

Vector code generation straightforward
Emphasis on analysis and disambiguation
It’s messy today
Vectorization yesterday

\[
\text{DO } 1 \ k = 1,n \\
\begin{align*}
1 & \quad A(k) = B(k) + C(k) \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>K=1</th>
<th>K=2</th>
<th>K=1..2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld C(1)</td>
<td>Ld C(2)</td>
<td>Ld C(1) Ld C(2)</td>
</tr>
<tr>
<td>Ld B(1)</td>
<td>Ld B(2)</td>
<td>Ld B(1) Ld B(2)</td>
</tr>
<tr>
<td>Add</td>
<td>Add</td>
<td>Add Add</td>
</tr>
<tr>
<td>St A(1)</td>
<td>St A(2)</td>
<td>St A(1) St A(2)</td>
</tr>
</tbody>
</table>

Scalar code

Vector code

Vector code generation was straightforward

Emphasis on analysis and disambiguation
Vectorization today

Vector code generation has become a more difficult problem.
Increasing need for user guided explicit vectorization.
Explicit vectorization maps threaded execution to SIMD hardware.
```c
#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 / 2.0)) / (w / 2.0);
            Ray ray; Issect 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;
            }
        }
    }
}
```
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          x = 1          x = 2          x = 3
z = z * c      z = z * c      z = z * c      z = z * c
z = z * c      z = z * c      z = z * c      z = z * c
       ....          ..................          ....

iters = 2      iters = 23     iters = 255     iters = 37
Mandelbrot

```c
#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]);
    }
}
```

```c
#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 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.
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.

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

### Optimization Notice

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

Notice revision #20110804