Performance, SIMD, Vectorization and Performance Tuning

James Reinders
August 1, 2016, Pheasant Run, St Charles, IL
13:00-13:45
Knights Landing Clustering and Memory Modes, use and implications on the future of architecture and memory configurations.

Vectorization, current state of the art thinking, use and implications on the future of data parallelism through threading + SIMD instructions.
Parallel first

Vectorize second
Multithreading is more powerful than vectorization — by simple math:

16 way from vectorization

244 way from thread parallelism
There is an urban legend that Albert Einstein once said that compounding interest is the most powerful force in the universe.
16 x 244 = 3904

MULTIPLICATION
EINSTEIN WAS RIGHT

Multithreading
Vectorization
Assertion:

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

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

© 2016, James Reinders. 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.
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 |

© 2016, James Reinders. 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.
## Vector addition

\[
\begin{bmatrix}
4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\
-0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\
4.1 & 0.6 & 3.6 & -8.5 & -1.2 & 2.5 & 8.4 & 6.3 & -2.2 & 4.2 & 4.3
\end{bmatrix}
\]
...and Vector multiplication

\[
\begin{bmatrix}
4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\
-0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\
4.1 & 0.6 & 3.6 & -8.5 & -1.2 & 2.5 & 8.4 & 6.3 & -2.2 & 4.2 & 4.3
\end{bmatrix}
\]

\[
\begin{bmatrix}
4.4 & 1.1 & 3.1 & -8.5 & -1.3 & 1.7 & 7.5 & 5.6 & -3.2 & 3.6 & 4.8 \\
-0.3 & -0.5 & 0.5 & 0 & 0.1 & 0.8 & 0.9 & 0.7 & 1 & 0.6 & -0.5 \\
-1.32 & -0.55 & 1.55 & 0 & -0.13 & 1.36 & 6.75 & 3.92 & -3.2 & 2.16 & -2.4
\end{bmatrix}
\]
An example
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];
}
```

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 -&gt; c[i:i+3]
5. ADD i + 4 -&gt; i

Loop:
1. LOAD a[i] -&gt; Ra
2. LOAD b[i] -&gt; Rb
3. ADD Ra, Rb -&gt; Rc
4. STORE Rc -&gt; c[i]
5. ADD i + 1 -&gt; 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

We call this “vectorization”

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];
}
```
IVDEP (ignore assumed vector dependencies)

```c
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;
}
```
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++);
}
array operations (Cilk™ Plus)

```c
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
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)

Or explicitly use…
OpenMP
Intel® Threading Building Blocks (TBB)

Best at being
Reliable, predictable and portable

```c
c$OMP PARALLEL DO
  DO I=1,N B(I) = (A(I) + A(I-1)) / 2.0
END DO
```
```c
C$OMP END PARALLEL DO
```
OpenMP 4.0

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).

Based on a proposal from Intel based on customer success with the Intel® Cilk™ Plus features in Intel.
OpenMP 4.0

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

```
#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)*

**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++**

```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)`

**Fortran**

```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)`

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).

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

```c
#pragma omp declare simd [clause[ , ] clause] ... new-line
#pragma omp declare simd [clause[ , ] clause] ... new-line

... function definition or declaration
```

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`

where `clause` is one of the following:

- `simdlen(length)`
- `linear(argument-list[:constant-linear-step])`
- `aligned(argument-list[:alignment])`
- `uniform(argument-list)`
- `inbranch`
- `notinbranch`
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++

```c
#pragma omp for simd [clause] [clause] ... 
for-loops
```

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

Fortran

```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.
You like directives?

Yes

Use OpenMP 4.0

No

You are not alone.
**for your consideration:**

Intel Compilers 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

<table>
<thead>
<tr>
<th>Year</th>
<th>Technology</th>
<th>New Instructions</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>MMX™ technology</td>
<td>57 new instructions</td>
<td>64 bits, Overload FP stack, Integer only media extensions</td>
</tr>
<tr>
<td>1998</td>
<td>Intel® SSE</td>
<td>70 new instructions</td>
<td>128 bits, 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 new instructions</td>
<td>128 bits, 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 new instructions</td>
<td>128 bits, 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 new instructions</td>
<td>128 bits, enhanced packed integer calculation</td>
</tr>
<tr>
<td>2007</td>
<td>Intel® SSE4.1</td>
<td>47 new instructions</td>
<td>128 bits, 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 new instructions</td>
<td>128 bits, string (XML) processing, POP-Count, CRC32</td>
</tr>
</tbody>
</table>
## Intel Instruction Set Vector Extensions since 2011

<table>
<thead>
<tr>
<th>Year</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Intel® AVX</td>
<td>Co-processor only 512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promotion of 128 bit FP vector instructions to 256 bit</td>
</tr>
</tbody>
</table>
| 2011 | “AVX-1.5” | 7 new instructions  
16 bit FP support  
RDRAND  
... |
| 2012 | Intel® AVX-2 | Promotion of integer instruction to 256 bit  
- FMA  
- Gather  
- TSX/RTM |
| 2013 | Intel® AVX-512 | Promotion of vector instructions to 512 bits  
Xeon Phi: FI, CDI, ERI, PFI  
Xeon: FI, CDI, BWI, DQI, VLE |
<p>| TBD  |         | Reinders blogs announced – July 2013, and June 2014. |</p>
<table>
<thead>
<tr>
<th>Year</th>
<th>Prefix</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><strong>Future</strong></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:

```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
Motivation for AVX-512 Conflict Detection

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

Consider the “histogram” problem:

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

- 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

```c
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*
“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 (in 2014).

Expect more changes to come, during beta and in future versions.
Optimization Reports (since 2014)

  - `[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
"Vectorization Advisor" – Advisor XE

1. "All the data you need in one place"
   Leverages Intel Compiler opt-report+ and dynamic profile.
   Support for other compilers, C, C++, Fortran, for MPI env.

2. Detects “hot” un-vectorized or “under vectorized” loops.
   Identifies what is blocking efficient vectorization, where to add it

3. Identify performance penalties and recommend fixes
   Explicit advices with “true intelligence”, covering OpenMP4.x

4. Memory layout analysis

5. Increase the confidence that vectorization is safe
Vectorization Advisor.
Assist code modernization for x86 SIMD

1. Compiler diagnostics + Performance
   Data + SIMD efficiency information

2. Guidance: detect problem and
   recommend how to fix it

3. “Accurate” Trip Counts: understand
   parallelism granularity and overheads

4. Loop-Carried Dependency Analysis

5. Memory Access Patterns Analysis
Summary

We need to embrace explicit vectorization in our programming.
Summary

We need to embrace explicit vectorization in our programming. But, generally use parallelism first (tasks, threads, MPI, etc.)
KEEP CALM AND LOOK FOR WHAT YOU DO NOT SEE
It is hard to “see” if you do not look.
It is hard to “see” if you do not look.

We could guess,

after all – we are smart enough
to believe we know what is happening.
Look for:

?
Look for:

• Confirmation
Look for:
• Confirmation
• Surprises
Look for:
• Confirmation
• Surprises

Your EXPERTISE will grow as you investigate.
Optimization: A Top-down Approach

Optimization: A Top-down Approach

**Hardware tuning:**
- BIOS (TB, HT)
- Memory
- Network I/O

**Operating System tuning:**
- Page size
- Swap file
- RAM Disk
- Power settings

**Better application design:**
- Parallelization
- Fast algorithms / data bases
- Programming language and RT libs
- Performance libraries
- Driver tuning

**Tuning for Microarchitecture:**
- Compiler settings/Vectorization
- Memory/Cache usage
- CPU pitfalls

Application Tuning

Who: Software Developers, Performance Engineers, Domain Experts

How:
• Workload selection
  • Repeatable results
  • Steady stat
• Define Metrics and Collect Baseline
  • Wall-clock time, FLOPS, FPS
  • <insert your metric here>
• Identify Hotspots
  • Focus effort where it counts
  • Use Tools
• Determine inefficiencies
  • Is there parallelism?
  • Are you memory bound?
  • Will better algorithms or programming languages help?

This step often requires some knowledge of the application and its algorithms
Application Tuning
Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

```
$ oprofile --exclude-dependent --demangle=smart --symbols `which lyx`
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask of 0x00 (No unit mask)

<table>
<thead>
<tr>
<th>vma</th>
<th>samples</th>
<th>%</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>081ec974</td>
<td>5016</td>
<td>8.5096</td>
<td>_Rb_tree&lt;unsigned short, pair&lt;unsigned short const, int&gt;, unsigned short const</td>
</tr>
<tr>
<td>0810c4ec</td>
<td>3323</td>
<td>5.6395</td>
<td>Paragraph::getFontSettings(BufferParams const&amp;, int) const</td>
</tr>
<tr>
<td>081319d8</td>
<td>3220</td>
<td>5.4627</td>
<td>LyXText::getFont(Buffer const*, Paragraph*, int) const</td>
</tr>
<tr>
<td>080e45d8</td>
<td>3011</td>
<td>5.1082</td>
<td>LyXFont::realize(LyXFont const*)</td>
</tr>
<tr>
<td>080e3d78</td>
<td>2623</td>
<td>4.4499</td>
<td>LyXFont::LyXFont()</td>
</tr>
<tr>
<td>081255a4</td>
<td>1823</td>
<td>3.0927</td>
<td>LyXText::singleWidth(BufferView*, Paragraph*, int, char) const</td>
</tr>
<tr>
<td>080e3cf0</td>
<td>1804</td>
<td>3.0505</td>
<td>operator==((LyXFont::FontBits const&amp;, LyXFont::FontBits const&amp;)</td>
</tr>
<tr>
<td>081128e0</td>
<td>1729</td>
<td>2.9332</td>
<td>Paragraph::Paragraph::getChar(int) const</td>
</tr>
<tr>
<td>081ead02</td>
<td>1380</td>
<td>2.3412</td>
<td>font_metrics::width(char const*, unsigned, LyXFont const&amp;)</td>
</tr>
<tr>
<td>08110d60</td>
<td>1310</td>
<td>2.2224</td>
<td>Paragraph::Paragraph::getChar(int) const</td>
</tr>
<tr>
<td>081e8bc94</td>
<td>1227</td>
<td>2.0816</td>
<td>qfont_loader::getFontInfo(LyXFont const&amp;)</td>
</tr>
</tbody>
</table>
```

oprofile: http://oprofile.sourceforge.net/
Application Tuning
Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

sysprof: http://sysprof.com
Application Tuning

Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

Application Tuning

Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

Application Tuning

Find Hotspots

- This could be at the module, function, or source code level
- Determine your own granularity

This may reinforce your understanding of the application but often reveals surprises
Application Tuning
Resource Utilization

- Is the application parallel?
- Multi-thread vs. Multi-process
- Memory Bound?

```
last pid: 86494; load averages: 0.83, 0.65, 0.69 up 67:22:48:43 14:44:15
227 processes: 1 running, 224 sleeping, 2 zombie
CPU: 20.2% user, 0.0% nice, 6.5% system, 0.2% interrupt, 73.1% idle
Mem: 1657M Active, 1868M Inact, 273M Wired, 190M Cache, 112M Buf, 11M Free
Swap: 4500M Total, 249M Used, 4251M Free, 5% Inuse

<table>
<thead>
<tr>
<th>PID</th>
<th>USERNAME</th>
<th>THR</th>
<th>PRI</th>
<th>NICE</th>
<th>SIZE</th>
<th>RES</th>
<th>STATE</th>
<th>C</th>
<th>TIME</th>
<th>WCPU</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>86490</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>150M</td>
<td>30204K</td>
<td>accept 1</td>
<td>0:02</td>
<td>11.18%</td>
<td>php-cgi</td>
<td></td>
</tr>
<tr>
<td>86490</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>150M</td>
<td>29912K</td>
<td>accept 0</td>
<td>0:02</td>
<td>8.93%</td>
<td>php-cgi</td>
<td></td>
</tr>
<tr>
<td>86490</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>94M</td>
<td>99M</td>
<td>swait 1</td>
<td>0:01</td>
<td>7.96%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>85005</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>150M</td>
<td>35204K</td>
<td>accept 2</td>
<td>0:07</td>
<td>7.57%</td>
<td>php-cgi</td>
<td></td>
</tr>
<tr>
<td>85274</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>149M</td>
<td>40868K</td>
<td>sbwait 3</td>
<td>0:27</td>
<td>5.18%</td>
<td>php-cgi</td>
<td></td>
</tr>
<tr>
<td>85267</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>151M</td>
<td>40944K</td>
<td>sbwait 2</td>
<td>0:33</td>
<td>4.59%</td>
<td>php-cgi</td>
<td></td>
</tr>
<tr>
<td>85004</td>
<td>www</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>150M</td>
<td>41504K</td>
<td>accept 2</td>
<td>0:14</td>
<td>4.59%</td>
<td>php-cgi</td>
<td></td>
</tr>
<tr>
<td>85267</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>95M</td>
<td>128M</td>
<td>sbwait 1</td>
<td>0:04</td>
<td>4.29%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>85066</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>94M</td>
<td>161M</td>
<td>sbwait 0</td>
<td>0:00</td>
<td>3.37%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>86459</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>94M</td>
<td>75960K</td>
<td>sbwait 2</td>
<td>0:01</td>
<td>3.37%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>85274</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>95M</td>
<td>192M</td>
<td>sbwait 2</td>
<td>0:14</td>
<td>2.39%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>85269</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>95M</td>
<td>199M</td>
<td>sbwait 1</td>
<td>0:19</td>
<td>2.20%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>85267</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>152M</td>
<td>44366K</td>
<td>sbwait 2</td>
<td>0:32</td>
<td>1.17%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>85273</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>95M</td>
<td>215M</td>
<td>sbwait 0</td>
<td>0:19</td>
<td>1.17%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>97982</td>
<td>psql</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>26020K</td>
<td>6832K</td>
<td>select 0</td>
<td>16:55</td>
<td>0.00%</td>
<td>postgres</td>
<td></td>
</tr>
<tr>
<td>892</td>
<td>root</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3168K</td>
<td>8K</td>
<td>-</td>
<td>2:13</td>
<td>33.33%</td>
<td>nfsd</td>
<td></td>
</tr>
<tr>
<td>1796</td>
<td>root</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>19780K</td>
<td>13660K</td>
<td>select 3</td>
<td>12:43</td>
<td>0.00%</td>
<td>Xvfb</td>
<td></td>
</tr>
</tbody>
</table>
Application Tuning
Resource Utilization

- Is the application parallel?
Application Tuning

Resource Utilization

• Memory Bound?

• Know your max theoretical memory bandwidth
Application Tuning
Resource Utilization

MPI applications have added communication complexity

Application Tuning
What’s Next?

• If your Hotspots are common algorithms:
  • Look for optimized libraries
• If your Hotspots are uncommon:
  • Compiler optimizations
  • Expert analysis and refactoring of an algorithm
    • The opposite of “low-hanging fruit”
  • Deeper analysis of hardware performance
    • More on this later
• If the system is underutilized:
  • Add parallelism - multi-thread or multi-process
    • OpenMP, TBB, Cilk, MPI, etc…

- Tools can help you determine where to look and may identify some issues.
- Some tools may provide suggestions for fixes.
- In the end – the developer and/or expert has to make the changes and decisions – there is no silver bullet.
Optimization: A Top-down Approach

- **System**
  - H/W tuning:
    - BIOS (TB, HT)
    - Memory
    - Network I/O
  - OS tuning:
    - Page size
    - Swap file
    - RAM Disk
    - Power settings

- **Application**
  - Better application design:
    - Parallelization
    - Fast algorithms / data bases
    - Programming language and RT libs
    - Performance libraries
    - Driver tuning

- **Processor**
  - Tuning for Microarchitecture:
    - Compiler settings/Vectorization
    - Memory/Cache usage
    - CPU pitfalls
Microarchitecture Tuning

Who: **Architecture Experts**  
Software Developers, Performance Engineers, Domain Experts

How:
- Use architecture specific hardware events
- Use predefined metrics and best known methods
  - Often hardware specific
  - (Hopefully) provided by the vendor
- Tools make this possible for the non-expert
  - Linux perf
  - Intel® VTune™ Amplifier XE
- Follow the Top-Down Characterization
  - Locate the hardware bottlenecks
Introduction to Performance Monitoring Unit (PMU)

Registers on Intel CPUs to count architectural events
- E.g. Instructions, Cache Misses, Branch Mispredict

Events can be counted or sampled
- Sampled events include Instruction Pointer

Raw event counts are difficult to interpret
- Use a tool like VTune or Perf with predefined metrics
Raw PMU Event Counts vs Metrics
Adding Regression Tests for Performance

Regression testing isn’t just for bugs

1. Create a baseline performance characterization
2. After each change or at a regular interval
   1. Compare new results to baseline
   2. Compare new results to previous results
   3. Evaluate the change
3. goto (1)

Performance tuning is easier if it’s always on your mind and integrated into your development
Scientific Approach to Analysis

• None of the tools provide exact results
  • Data collection overhead or dropping details
  • Define what results need to be precise

• Low overhead tools provide statistical results
  • Statistical theory is applicable
  • Think of proper sampling frequency (for data bandwidth)
  • Think of proper length of data collection (for process)
  • Think of proper number of experiments and results deviation

• Take into account other processes in a system
  • Anti-virus
  • Daemons and services
  • System processes

• Start early – tune often!

BETA for “2017” Product – NOW
Vectorization advisor

Many factors impact achieving good vectorization for our applications. The Vectorization Advisor directly analyzes an application and provides feedback on the extent of current vectorization and on possible steps to achieve more effective vectorization. Vectorization Advisor works with any compiler although some features in the Intel® compilers will increase the effectiveness of advice from the Vectorization Advisor tool. It is like having an expert sitting next to us who never tires of digging into an application to analyze what is really happening.

The Vectorization Advisor is one of the two major workflows (feature sets) available in the Intel® Advisor “2016” and later versions. The Intel Advisor also includes a thread prototyping feature set which can be useful for analysis of scaling for threads. In this chapter, we focus on using the Vectorization Advisor to help us maximize our vectorization performance.

What is new with Knights Landing in this chapter?

AVX-512 and the Vectorization Advisor within the Intel® Advisor tool.

I will talk about some NEW “2017” features – which help Intel Xeon processors tuning and Intel Xeon Phi processor tuning BOTH – Of Course!
Memory Access Pattern Report

MEMORY ACCESS PATTERN REPORT
An initial survey analysis of hot loops often identifies inefficient memory access patterns as a main bottleneck. Memory access patterns issues are the toughest and most frequent performance problem in code not yet modernized for vector SIMD parallelism.

Applying straightforward SIMD and threading optimizations often does not provide desirable speedups because some parts of applications (including vectorized hot loops) become memory bound. Memory-access-patterns-bound code is just one subtype of a larger memory-bound class of problems, along with memory-bandwidth-bound and partially overlapping with memory-latency-bound subtypes.

Can recommend:
- AoS to SoA
- AoSoA
- Use of SDLT
- Use of MCDRAM

Gather/Scatter Report
Data Layout: AoS vs. SoA

Array of structures (AoS) tends to cause cache alignment problems, and is hard to vectorize.

Structure of arrays (SoA) can be easily aligned to cache boundaries and is vectorizable.
Data Layout: Alignment

Array of Structures (AoS), padding at end.

Array of Structures (AoS), padding after each structure.

Structure of Arrays (SoA), padding at end.

Structure of Arrays (SoA), padding after each component.
Mask Utilization and FLOPS Profiler

FLOPs Report
• Vector Efficiency
• Memory Access Pattern,
• Roofline Analysis Graph

While masked operators have been traditionally used in scientific computing applications, they also open up opportunities for vectorization, which can improve performance.

This is an early version of the Loop Analytics tab; it will be enhanced with more information over time.

Vectorization efficiency and FLOP/s in Survey Report and Loop Analytics.
...supplements AI-based analysis with a dynamic FLOP/s profile and peak FLOPs and memory sub-system throughput levels providing enlightening “bounds and bottlenecks” analysis for complex workloads.
References

• Top-Down Performance Tuning Methodology
  • www.software.intel.com/en-us/articles/de-mystifying-software-performance-optimization

• Top-Down Characterization of Microarchitectural Bottlenecks

• Intel® VTune™ Amplifier XE
  • www.intel.ly/vtune-amplifier-xe

• Tuning Guides
  • www.intel.com/vtune-tuning-guides
Look for:
• Confirmation
• Surprises

Do not skip either
KEEP CALM AND MIND YOUR ALGORITHMS
Book Cover Background: Photo of the COSMOS@DiRAC SGI UV2000 based Supercomputer manufactured by SGI, Inc and operated by the Stephen Hawking Centre for Theoretical Cosmology, University of Cambridge. Photo courtesy of Philip Mynott. Book Cover Foreground: 3D visualization of statistical fluctuations in the Cosmic Microwave Background, the remnant of the first measurable light after the Big Bang. CMB data is from the Planck satellite and is the topic of Chapter 10 providing insights into new physics and how the universe evolved. Visualization rendered with Intel's OSPRay ray tracing open source software by Gregory P. Johnson and Timothy Rowley, Intel Corporation.

#Moderncode: COSMOS
#Moderncode: COSMOS

## High Performance Parallelism Pearls

*Volume Two*

More proven techniques and real-world examples of highly scalable parallel programming

James Reinders and Jiri Laffers

---

**What?**

![Graph showing speed-up comparison between processor and coprocessor versions of code.](chart.png)

*Graph: Speed-up (Relative to Version 1 on Processor)*

- **Processor:** Various values indicate speed-up factors across different code versions.
- **Coprocessor:** Corresponding values showing the speed-up relative to the processor versions.

---

Book Cover Background: Photo of the COSMOS@Dirac SGI UV2000 based Supercomputer manufactured by SGI Inc and operated by the Stephen Hawking Centre for Theoretical Cosmology, University of Cambridge. Photo courtesy of Philip Mynott.

Book Cover Foreground: 3D visualization of statistical fluctuations in the Cosmic Microwave Background, the remnant of the first measurable light after the Big Bang. CMB data is from the Planck satellite and is the topic of Chapter 10 providing insights into new physics and how the universe evolved. Visualization rendered with Intel's OSPRay ray tracing open source software by Gregory P. Johnson and Timothy Rowley, Intel Corporation.
We find that using a simple trapezium rule integrator combined with hand-selected sampling points (to improve accuracy in areas of interest) provides sufficient numerical accuracy to obtain a physically meaningful result, and the reduced space and time requirements of this simplified method give a speed-up of $O(10x)$. 

<table>
<thead>
<tr>
<th>Version</th>
<th>Processor (s)</th>
<th>Coprocessor (s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2887.0</td>
<td>-</td>
<td>Original code.</td>
</tr>
<tr>
<td>2</td>
<td>2610.0</td>
<td>-</td>
<td>Loop simplification.</td>
</tr>
<tr>
<td>3</td>
<td>882.0</td>
<td>-</td>
<td>Intel® MKL integration routines and function inlining.</td>
</tr>
<tr>
<td>4</td>
<td>865.9</td>
<td>1991.6</td>
<td>Flattened loops and introduced OpenMP threads.</td>
</tr>
<tr>
<td>5</td>
<td>450.6</td>
<td>667.9</td>
<td>Loop reordering and manual nested threading.</td>
</tr>
<tr>
<td>6</td>
<td>385.6</td>
<td>655.0</td>
<td>Blocked version of the loop (for cache).</td>
</tr>
<tr>
<td>7</td>
<td>46.9</td>
<td>49.5</td>
<td>Numerical integration routine (Trapezium Rule).</td>
</tr>
<tr>
<td>8</td>
<td>37.4</td>
<td>37.7</td>
<td>Reduction with DGEMM.</td>
</tr>
<tr>
<td>9</td>
<td>35.1</td>
<td>34.5</td>
<td>Data alignment (for vectorization).</td>
</tr>
<tr>
<td>10</td>
<td>34.3</td>
<td>26.6</td>
<td>Tuning of software prefetching distances.</td>
</tr>
</tbody>
</table>
KEEP CALM
HPC WILL SAVE THE WORLD
James Reinders. Parallel Programming Enthusiast

James has been involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. James worked 10,001 days as an Intel employee 1989-2016, and contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red), first 3 TeraFLOP/s supercomputer (ASCI Red upgrade), the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor) and the world's first 3 TeraFLOP/s microprocessor (Intel® Xeon Phi™ Processor). 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), Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls Volume 1 (Morgan Kaufmann, Nov. 2014), High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, Aug. 2015), and Intel® Xeon Phi™ Processor High Performance Programming - Knights Landing Edition (Morgan Kaufmann, 2016).