Agenda

• Introduction to Vectorization

• Ways to write vector code
  • Automatic loop vectorization
  • Array notation
  • Elemental functions

• Other optimizations

• Summary
What’s first?

Have you considered reusing well-tuned code rather than tuning your code?

Sure.

... but I have this application... and the compiler just need to get it right.

Have you considered tuning the code by identifying parts that are covered by a library?
Performance Libraries: Motivation

How and where to optimize?

1. Appropriate algorithm*
2. Performance Library
3. Multicore
4. SIMD

Delivered Values

• Easy access to high perf.
• Rich functionality
• Support

* The best parallel algorithm might be unrelated to the best serial algorithm; hence exploiting parallelism is not necessarily an incremental “optimization”.

```c
for (int i = 0; i < M; ++i) {
    for (int j = 0; j < N; ++j) {
        c[i*K+j] = 0;
        for (int k = 0; k < K; ++k) {
            c[i*K+j] += a[i*N+k] * b[k*K+j];
        }
    }
}
```
### Intel® Math Kernel Library (Intel® MKL)

#### Linear Algebra
- BLAS, Sparse BLAS
- LAPACK solvers
- Sparse Solvers (DSS, PARADISO)
- Iterative solver (RCI)
- ScaLAPACK, PBLAS

#### Fast Fourier Transforms
- Multidimensional
- FFTW interfaces
- Cluster FFT
- Trig. Transforms
- Poisson solver
- Convolution via VSL

#### Vector Math
- Trigonometric
- Hyperbolic
- Exponential, Logarithmic
- Power / Root

#### Random Number Gen.
- Congruential
- Wichmann-Hill
- Mersenne Twister
- Sobol
- Neiderreiter
- Non-deterministic

#### Summary Statistics
- Kurtosis
- Variation coefficient
- Quantiles
- Order statistics
- Min/max
- Variance-covariance

#### Data Fitting
- Spline-based
- Interpolation
- Cell search
History of SIMD ISA extensions

Intel® Pentium® processor (1993)

MMX™ (1997)

Intel® Streaming SIMD Extensions (Intel® SSE in 1999 to Intel® SSE4.2 in 2008)

Intel® Advanced Vector Extensions (Intel® AVX in 2011 and Intel® AVX2 in 2013)

Intel Many Integrated Core Architecture (Intel® MIC Architecture in 2013)

* Illustrated with the number of 32-bit data elements that are processed by one “packed” instruction.
Vectors (SIMD)

float *restrict A;
float *B, *C;

for (i=0; i<n; ++i) {
    A[i] = B[i] + C[i];
}

Scalar code computes the above with one-element at a time.

addps xmm1, xmm2

- **SSE**: 4 elements at a time
  addps xmm1, xmm2

- **AVX**: 8 elements at a time
  vaddps ymm1, ymm2, ymm3

- **MIC**: 16 elements at a time
  vaddps zmm1, zmm2, zmm3
Vector Instructions

• Compile with –S to see assembly code (if you like)
• Documentation: Optimization Guide (quick: zmmintrin.h)

• A vectorized loop contains instructions like
  vfmadd213ps %zmm23, %zmm8, %zmm2 # fma instruction
  vaddps %zmm25, %zmm2, %zmm0 # single precision add

• In a scalar loop, these instructions will be masked, e.g.
  vfmadd213ps %zmm17, %zmm20, %zmm1{%k1}
  vaddps %zmm23, %zmm1, %zmm0{%k1}

• Example of vectorized math function for Intel® MIC architecture:
  call __svml_sinf16 # calculates sin(x) for 16 floats
  call __svml_sinf16_mask
Auto-vectorization, array notation, and vect. hints

- Multiple code paths possible (-ax, /Qax)
- Forward-scaling (different SIMD widths)

**Ease of use**

**Fully automatic vectorization**

**Built-in vectors (explicit)**

Auto vect. hints (#pragma ivdep, ...)

SIMD intrinsic class (F32vec4 add)

Vector intrinsic (mm_add_ps())

Assembler code (addps)

**Ultimate control**
Multiple Code Paths (Retargeting)

double A[1000], B[1000], C[1000];

void add() {
    for (int i = 0; i < 1000; ++i) {
        if (A[i] > 0) {
            A[i] += B[i];
        } else {
            A[i] += C[i];
        }
    }
}

.B1.2::
    movaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpltppd xmm0, xmm2
movaps xmm1, B[rdx*8]
andps xmm1, xmm0
andnps xmm0, C[rdx*8]
orps xmm1, xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add rdx, 2
cmp rdx, 1000
jl .B1.2 SSE2

.B1.2::
    movaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpltppd xmm0, xmm2
movaps xmm1, C[rdx*8]
blendvpd xmm1, B[rdx*8], xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add rdx, 2
cmp rdx, 1000
jl .B1.2 SSE4.1

AVX

SSE2

Multiple Code Paths (Retargeting)
Basic Vectorization Switches I

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

- **Linux**, Mac OS X: `-m<feature>`, Windows*: `/arch:<feature>`
  - Neither check nor specific optimizations for Intel processors:
    Application optimized for both Intel and non-Intel processors for selected SIMD feature
  - Missing check can cause application to fail in case extension not available

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

Array Notation

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

Elemental Function

```c
__declspec(vector)
float ef(float a, float b) {
    return a + b;
}
A[:] = ef(B[:], C[:]);
```

SIMD Directive

```c
#pragma simd
for (int i = 0; i < N; ++i) {
    A[i] = B[i] + C[i];
}
```

Auto-Vectorization

```c
for (int i = 0; i < N; ++i) {
    A[i] = B[i] + C[i];
}
```
Automatic Vectorization

The vectorizer for Intel® MIC architecture works just like for SSE or AVX on the host, for C, C++ and Fortran

- Enabled at default optimization level (-O2)
- Data alignment should be to 64 bytes, instead of 16 (see later)
- More loops can be vectorized, because of masked vector instructions, gather/scatter instructions, fused multiply-add (FMA)
- Avoid 64 bit integers where not essential

Vectorized loops may be recognized by:

- Vectorization and optimization reports (simplest), e.g.
  - vec-report2 or -opt-report-phase hpo
- Unmasked vector instructions (there are no separate scalar instructions; masked vector instructions are used instead)
- Gather & scatter instructions
- Math library calls to libsvml
Requirements for Auto-Vectorization

- Innermost loop of nest (a few simple exceptions)
- Straight-line code (masked assignments OK)
- Avoid:
  - Function/subroutine calls (unless inlined or vector)
  - Non-mathematical operators
  - Data-dependent loop exit conditions
    - Iteration count should be known at entry to loop
  - Loop carried data dependencies (Reduction loops OK)
  - Non-contiguous data (indirect addressing; non-unit stride)
    - Inefficient
  - Inconsistently aligned data
- Directives/pragmas can help:
  - #pragma ivdep ...... ignore potential dependencies
  - #pragma vector always ignore efficiency heuristics
  - aligned assume data aligned
  - Compiler can generate runtime alignment and dependency tests for simple loops (but less efficient)

See http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
# Vectorizable Math Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>ceil</td>
<td>fabs</td>
<td>round</td>
</tr>
<tr>
<td>acosh</td>
<td>cos</td>
<td>floor</td>
<td>sin</td>
</tr>
<tr>
<td>asin</td>
<td>cosh</td>
<td>fmax</td>
<td>sinh</td>
</tr>
<tr>
<td>asinh</td>
<td>erf</td>
<td>fmin</td>
<td>sqrt</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
<td>log</td>
<td>tan</td>
</tr>
<tr>
<td>atan2</td>
<td>erfinv</td>
<td>log10</td>
<td>tanh</td>
</tr>
<tr>
<td>atanh</td>
<td>exp</td>
<td>log2</td>
<td>trunc</td>
</tr>
<tr>
<td>cbrt</td>
<td>exp2</td>
<td>pow</td>
<td></td>
</tr>
</tbody>
</table>

Also float versions, such as `sinf()`

Uses short vector math library, `libsvml`
Math Functions and Libraries

Intel Compiler: scalar and vectorized math functions

- **IMF**: Intel Math Functions (scalar)
  - `-fimf-precision=<high|medium|low>`
    - Default/usually: scalar/high and vect./medium
      - high: ~0.55 ulps, medium: ~2 ulps (but < 4 ulps)
    - `-fimf-arch-consistency=<true|false>`
      - Default is “false” even with `-fp-model=precise`
      - Not available across 32-bit / 64-bit

- **SVML**: Short Vector Math Functions
  
  General form: `_mm* _svml_[function]_p[s|d]`
  
  e.g., `_mm_svml_round_ps`, or `_mm256_erfc_pd`

Intel MKL: vectorized and parallelized math functions

- **VML**: optimized for throughput – three accuracy/performance levels
Vectorization Report

Get details on vectorization’s success and failure

• L&M: \(-vec\)-report\(<n>\), \(n=0,1,2,3,4,5,6,7\)*
• W: /Qvec-report\(<n>\), \(n=0,1,2,3,4,5,6,7\)*

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

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

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

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

* Diagnostic level: (0) no diagnostic, (1) vectorized loops, (2) vectorized loops and non-vect. loops
Vectorization Report Messages

“Loop was not vectorized” because:

• “Low trip count”
• “Existence of vector dependence”
  - Possible dependence of one loop iteration on another, e.g.
    for (j=1; j<MAX; j++) a[j] = a[j] + c * a[j-n];
• "vectorization possible but seems inefficient"
• “Not Inner Loop”

• It may be possible to overcome these using switches, pragmas, source code changes or explicit vector programming
Other Reports

-opt-report-phase hlo
  summarizes loop optimizations including loop interchange, fusion, distribution, unrolling, multi-versioning, cache blocking, prefetching, etc.

-opt-report-phase ipo_inl
  summarizes function inlining

-opt-report-phase offload
  gives a compile time summary of which data are copied to and from the coprocessor

**OFFLOAD_REPORT=2** (environment variable on host)
  gives run-time summary of data copied to and from the coprocessor and computation time on the coprocessor

Main optimization opportunity for the data offload:
Don’t transfer data that you don’t need! (e.g., copy in but not out)
User-Mandated Vectorization

User-mandated vectorization: SIMD directive / pragma

• Enables vectorization of vectorizable inner and outer loops
• Compiler heuristics are overwritten (incorrect code possible)
• Supplements automatic vectorization and other directives (IVDEP, VECTOR ALWAYS)
SIMD Directive Notation

C/C++:  #pragma simd [clause [,clause] ...]

Fortran:  !DIR$ SIMD [clause [,clause] ...]

Without an additional clause, the directive enforces vectorization of a vectorizable loop.

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

* Without the SIMD directive, vectorization will fail (too many pointer references to do a run-time overlap-check).
Clauses for SIMD directives

The programmer (i.e. you!) is responsible for correctness
- Just like for race conditions in OpenMP* loops

Available clauses (both OpenMP and Intel versions)
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE --- like OpenMP
- REDUCTION
- COLLAPSE (OpenMP 4.0 RC1 only; for nested loops)
- LINEAR (additional induction variables)
- SAFELEN (OpenMP 4.0 RC1 only)
- VECTORLENGTH (Intel only)
- ALIGNED (OpenMP 4.0 RC1 only)
- ASSERT (Intel only; “vectorize or die!”)
Aligning Data

● Allocate memory on heap aligned to n byte boundary:
  ```c
  void* _mm_malloc(int size, int n)
  int posix_memaligned(void **p, size_t n, size_t size)
  ```

● Alignment for variable declarations:
  ```c
  __attribute__((aligned(n))) var_name  or  __declspec(align(n)) var_name
  ```

And tell the compiler...

```c
#pragma vector aligned
```

• Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
  • May cause fault if data are not aligned

```c
__assume_aligned(array, n)
```

• Compiler may assume array is aligned to n byte boundary

**n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE**
Aligning Data

- Intel Xeon Phi is sensitive to unaligned load/store
  - It’s about the start address for homogenous data
  - It’s about each data member for structured data
- Alignment: \textbf{vector width} (64 Byte / 512 bit)
- Intel Xeon Phi fastest offload transfers
  - Alignment: \textbf{page-granularity} (4k... 2MB)
  - Multiple of vector width / page size
- Memory alignment for offloaded code section is \textbf{inherited} from alignment on the host unless specified otherwise (offload pragma’s align mod.)
Pointer Aliasing

Solutions for C/C++ (less of a problem in Fortran)

- ANSI rules / conformance
- Compiler switches
- Restrict keyword / intrinsics

ANSI rules
Type deduction and qualifiers specify what cannot alias each other.

Compiler switches
- fargument-noalias
- fstrict-aliasing (-ansi-alias)
- alias-const
- fno-alias

Example
Option -no-alias assumes that there is no aliasing.
Keyword restrict

Linux
- restrict
- std=c99

Windows
/Qrestrict
/Qstd=c99

• Breaks aliasing on a per-function basis
• Assertion to compiler
  – Only this pointer points to the underlying data
  – Also applies to the incremented pointer etc.
• Available for C (not part of the C++ standard)
  – Intel Compiler supports it for C++
Keyword restrict (cont.)

Make the restrict qualifier more portable*

```c
// #define USE_RESTRICT_OPTION
#if defined(__INTEL_COMPILER) && defined(USE_RESTRICT_OPTION)
#define RESTRICT restrict
#elif defined(__GNUC__) && !defined(_WIN32) && !defined(__CYGWIN32__)
#define RESTRICT __restrict__
#elif defined(_MSC_VER)
#define RESTRICT __restrict
#else
#define RESTRICT
#endif
```

* Or more handy: “better use RESTRICT”.
Example: OpenMP* vs. Vectors
Increase parallelism by combining nested loops

- More thread parallelism, less SIMD parallelism
- For example, A is too small for many cores
  - Break-up computation into S-blocks
  - Increase thread parallelism by B/S

```c
#pragma omp parallel for collapse(2)
for (int i = 0; i < A; ++i) {
    for (int s = 0; s < B; s += S) {
        int N = min(B - s, S);
        result[i*B+s:N] = a[i*B+s:N] * b[i*B+s:N];
    }
}
```
Example: OpenMP* and Vectors

OpenMP* 4.0 introduces several vector constructs
Helps to improve thread-vector interoperability
For example may help to avoid false sharing

```c
#pragma omp parallel
#pragma omp for simd
for (int i = 0; i < end; ++i) {
    for (int j = 0; j < M; ++j) {
    }
}
```

* See [http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf](http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf)
Pragmas and Directives

List available pragmas: `icc -help-pragma <dummy-file>`

Examples
- **IVDEP**: ignore vector dependency
- **LOOP COUNT**: advise typical iteration count(s)
- **UNROLL**: suggest loop unroll factor
- **DISTRIBUTE POINT**: advise where to split loop
- **VECTOR**: vectorization hints
  - **Aligned**: assume data is aligned
  - **Always**: override cost model
  - **Nontemporal**: advise use of streaming stores
- **NOVECTOR**: do not vectorize
- **NOFUSION**: do not fuse loops
- **INLINE/FORCEINLINE**: invite/require function inlining
- **SIMD ASSERT**: “vectorize or die”
Intel® Cilk™ Plus

Language extension (C/C++) for task-parallelism
• Usual advantages of built-in functionality
• Scheduler inspired others (e.g. Intel TBB)
• Blends well with existing code
• Only three main keywords

Data parallelism based on vectors*
• Complements auto-vectorization
• Notation for array sections (slices)
• Elemental functions (kernels)
• Reductions, gather, scatter, etc.

User-mandated vectorization (pragma simd)

* For example, Guy E. Blelloch: Vector Models for Data-Parallel Computing, 1990
Vector and Elemental Processing

**Vector Processing**

Natural in case of scatter, or with sync. primitives

\[
\begin{align*}
z[0:10:10] &= a[20:10:2] + y[x[0:10]]; \\
\end{align*}
\]

**Elemental Processing**

Kernel Function

\[
\begin{align*}
y[0:10:10] &= \sin(x[20:10:2]); \\
\end{align*}
\]

* The Intel Cilk Plus Array section syntax is [offset:size:stride] whereas F90 uses [lbound:ubound:stride].
Elemental Functions

- Essentially pre-vectorized functions
  - Can be called within a loop without inlining code
  - Control flow is supported (masked exec.)
  - Similar effect eventually via IP[O] (but more fragile)
  - Helps to avoid code bloat

- Great potential for building libraries
  - Binary kernel functions would vectorize
  - Means: vectorizable in a user’s loop!

- Launching an elemental function
  - Works with array sections (“range”)
Vector Elemental Function

Compiler generates vector version of a scalar function that can be called from a vectorized loop:

```c
__attribute__((vector(uniform(y, xp, yp))))
float func(float x, float y, float xp, float yp) {
    float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
    denom = 1./sqrtf(denom);
    return denom;
}
```

```c
#pragma simd private(x) reduction(+:sumx)
for (i=1; i<nx; i++) {
    x = x0 + (float)i * h;
    sumx = sumx + func(x, y, xp, yp);
}
```

y, xp and yp are constant, x can be a vector

These clauses are required for correctness, just like for OpenMP.*

SIMD LOOP WAS VECTORIZED.
Clauses for Vector Functions

__attributes__((vector)) (Intel)
#pragma omp declare simd (OpenMP* 4.0 RC1)

Available clauses (both OpenMP and Intel versions)

- **LINEAR** (additional induction variables)
- **UNIFORM** (arguments that are loop constants)
- **PROCESSOR** (Intel)
- **VECTORLENGTH** (Intel)
- **MASK / NOMASK** (Intel)
- **INBRANCH / NOTINBRANCH** (OpenMP 4.0 RC1)
- **SIMDLEN** (OpenMP 4.0 RC1)
- **ALIGNED** (OpenMP 4.0 RC1)
Example: Elemental Function

```cpp
__declspec(vector)
void kernel(int& result, int a, int b)
{
    result = a + b;
}

void sum(int* result, const int* a, const int* b,
         std::size_t size)
{
    for (std::size_t i = 0; i < size; ++i) {
        kernel(result[i], a[i], b[i]);
    }
}
```
Example: Threads and Vectors

```cpp
__declspec(vector)
void kernel(int& result, int a, int b)
{
    result = a + b;
}

void sum(int* result, const int* a, const int* b, std::size_t size)
{
    cilk_for (std::size_t i = 0; i < size; ++i) {
        kernel(result[i], a[i], b[i]);
    }
}
```
Array Section

Correspond to vector processing (SIMD)
• Explicit construct to *express* vectorization
• Compiler assumes no aliasing of pointers

Synonyms
• array notation, array section, array slice, vector

Syntax
• [start:size], or
• [start:size:stride]
• [:] → all elements*

* only works for array shapes known at compile-time
Array Section Operators

Most C/C++ operators work with array sections

- **Element-wise operators**
  (rank and size must match)
  
  a[0:10] * b[4:10]

- **Scalar expansion**
  
  a[10:10] * c

Assignment and evaluation

- Evaluation of RHS before assignment
  
  a[1:8] = a[0:8] + 1

- Parallel assignment to LHS
  
  ^ temp!

Gather and scatter

  a[idx[0:1024]] = 0
  b[idx[0:1024]] = a[0:1024]
  c[0:512] = a[idx[0:512:2]]
Array Section Reductions

Reductions

Built-in

\[
\text{__sec_reduce_add}(a[:]), \text{__sec_reduce_mul}(a[:]) \\
\text{__sec_reduce_min}(a[:]), \text{__sec_reduce_max}(a[:]) \\
\text{__sec_reduce_min_ind}(a[:]) \\
\text{__sec_reduce_max_ind}(a[:]) \\
\text{__sec_reduce_all_zero}(a[:]) \\
\text{__sec_reduce_all_nonzero}(a[:]) \\
\text{__sec_reduce_any_nonzero}(a[:])
\]

User-defined

result \text{__sec_reduce} (initial, a[:], fn-id)
void \text{__sec_reduce_mutating} (reduction, a[:], fn-id)
Other Operators

Index generation

\[ a[:] = \texttt{sec\_implicit\_index}(\text{rank}) \]

Shift operators

\[ b[:] = \texttt{sec\_shift} (a[:], \text{signed shift\_val}, \text{fill\_val}) \]
\[ b[:] = \texttt{sec\_rotate}(a[:], \text{signed shift\_val}) \]

Cast-operation (array dimensionality) e.g.,

float[100] \rightarrow float[10][10]
Example: Array Section

Array section:

\[ y[0:10:10] = \sin(x[20:10:2]) ; \]

Corresponding loop:

```
for (int i = 0, j = 0, k = 20;
     i < 10; ++i, j += 10, k += 2)
{
    y[j] = \sin(x[k]);
}
```
Example: Launch Elemental Function

```c
__declspec(vector)
void kernel(int& result, int a, int b)
{
    result = a + b;
}

void sum(int* result, const int* a, const int* b, std::size_t size)
{
    kernel(result[0:size], a[0:size], b[0:size]);
}
```
Example: Threads and Vectors

```cpp
__declspec(vector)
void kernel(int& result, int a, int b)
{
    result = a + b;
}

void sum(int* result, const int* a, const int* b,
         std::size_t size)
{
    cilk_for (std::size_t i = 0; i < size; ++i) {
        kernel(result[i], a[i], b[i]);
    }
}```
Example: Threads and Vectors (2)

```c
__declspec(vector)
void kernel(int& result, int a, int b)
{
    result = a + b;
}

void sum(int* result, const int* a, const int* b, std::size_t size)
{
    cilk_for (std::size_t i = 0; i < size; i += 8) {
        const std::size_t n = std::min(size - i, 8);
        kernel(result[i:n], a[i:n], b[i:n]);
    }
}

* For example, the remainder could be also handled separately (outside of the loop).
Example: Matrix-Vector Multiplication

```c
void mxv(double* result, const double* matrix, const double* vector, std::size_t nrows, std::size_t ncols)
{
    cilk_for (std::size_t i = 0; i < nrows; ++i) {
        const std::size_t start = i * ncols;
        result[i] = __sec_reduce_add(
            matrix[start:ncols] * vector[0:ncols]);
    }
}
```
Fixed-Size Array Sections

### Long Vector Coding
- Syntax: A[0:\textit{size}] where \textit{size} is only known at runtime
- VLA or otherwise allocated memory
- Referencing intermediate results req. scratch mem.
- Solution: stream “infinite” length data through a fixed-size local array

### Short Vector Coding
- Syntax: A[0:\textit{N}] (or A[:]) where \textit{N} is known at compile-time
- Local array (scope) can be entirely optimized away
- Referencing immediate results is light-weight
- No real memory consumption
Non-temporal Streaming Stores

- Store instruction-hint to leave data as hinted
- Load instructions may be hinted as well

```c
#pragma vector nontemporal(result)
for (int i = 0; i < N; ++i) {
    result[i] = a[i] + b[i];
}
```
Memory Prefetches - automatic

• Compiler prefetching is on by default for the Intel® Xeon Phi™ coprocessor at –O2 and above
  • Prefetches issued for regular memory accesses inside loops
  • But not for indirect accesses a[index[i]]
  • More important for Intel Xeon Phi coprocessor (in-order) than for Intel® Xeon® processors (out-of-order)
  • Very important for apps with many L2 cache misses
• Use the compiler reporting options to see detailed diagnostics of prefetching per loop
  -opt-report-phase hlo –opt-report 3 e.g.

Total #of lines prefetched in main for loop at line 49=4
Using noloc distance 8 for prefetching unconditional memory reference in stmt at line 49
Using second-level distance 2 for prefetching spatial memory reference in stmt at line 50

  -opt-prefetch=n (4 = most aggressive) to control
  -opt-prefetch=0 or –no-opt-prefetch to disable
Memory Prefetches - manual

- Use intrinsics
  
  ```c
  _mm_prefetch((char *) &a[i], hint);
  ```
  
  See xmmmintrin.h for possible hints (for L1, L2, non-temporal, ...)

  ```c
  MM_PREFETCH(A, hint)  for Fortran
  ```

- But you have to figure out and code how far ahead to prefetch

- Also gather/scatter prefetch intrinsics, see zmmmintrin.h and compiler user guide, e.g. _mm512_prefetch_i32gather_ps

- Use a pragma / directive (easier):
  
  ```c
  #pragma prefetch  a    [:hint[:distance]]
  #pragma noprefetch
  !DIR$  PREFETCH A, B, ...
  ```

- You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.

- Hardware L2 prefetcher is also enabled by default
  
  - If software prefetches are doing a good job, then hardware prefetching does not kick in
Memory Prefetches: Pragma Syntax

#pragma prefetch variable[:hint[:distance]]
CDEC$ prefetch variable[:hint[:distance]]

- Variable: array / pointer
- Hint: 0 – non-temporal (streaming store)
  1 – temporal (via cache hierarchy)
  2 – temporal (1\textsuperscript{st} level cache)
  3 – temporal (2\textsuperscript{nd} level cache)
- Distance: # elements/iter to be prefetched ahead
  - Pragma is applied in front of a loop
- Similar: pragma vector nontemporal(variable)
Example: Memory Prefetches

Make prefetches specific!

```c
#if defined(__MIC__)
#pragma prefetch a:1:16
#endif

for (int i = 0; i < N; ++i) {
  result += a[i];
}
```
Example: Prefetch Distance

Make the distance a safe constant...

{

    #define MYFUNC_PF_N 16

    # pragma prefetch a:1:MYFUNC_PF_N

    for (int i = 0; i < N; ++i) {
        result += a[i];
    }

    #undef MYFUNC_PF_N

}
Other Optimizations

Pragma \textit{unroll(factor)}

- Increase factor until no additional benefit can be measured: 1, 2, 4, 8, ...
- Excessive unrolling may increase register pressure

Commonly discovered (slow) code pattern

```c
for (int i = 0; i < M; ++i) {
    for (int j = 0; j < N; ++j) {
        dst[i] += src[j];
    }
}
```
Example: Reduction

```c
for (int i = 0; i < M; ++i) {
    float sum = src[0];
    for (int j = 1; j < N; ++j) {
        sum += src[j];
    }
    dst[i] = sum;
}
```
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

Trapping of floating-point exceptions in vector instructions is not supported

The bits of the SIMD floating-point control word that mask/unmask floating-point protections are protected
- If you try to unmask exceptions, your app will seg fault
- Unmasking by compiler switches such as –fp-trap or –fpe0 is disabled for native builds or for target part of an offload build
- The exception flags still get set, and you can test on these
- Otherwise, the computation just continues with QNaNs, infinities, etc
- -fp-model except or –fp-model strict preserves exception semantics
  - Generates x87 instead of vector instructions, big performance impact
  - May be useful for debugging

Denormals are supported
- Needs –no-ftz or –fp-model precise (like on host)
Floating-Point Behavior on Intel® Xeon Phi™ Coprocessors

-fp-model fast=2 enables some more aggressive optimizations
• Faster inlined versions of some math functions
  – May not give standard behavior for extreme or exceptional arguments

Floating-point results on Intel® Xeon Phi™ may not be bit-for-bit identical to results obtained on Intel® Xeon processors
• Most common cause is fused multiply-add (FMA) instructions
  – Not disabled by -fp-model precise
  – Can disable for testing with -no-fma
    – With some impact on performance
• Implementation of math functions might also differ
• To get close, try -fp-model precise -no-fma -fimf-precision=high
  – But most parallel reductions will still cause differences
Intel® MKL Vector Math Library

High Accuracy (HA)
• Correct rounding (>99%)
• Behaves according to C99
• Slowest, default mode

Low Accuracy (LA)
• At most 2 lsb incorrect
• Behaves according to C99
• 30-50% faster than HA

Enhanced Performance (EP)
• ~1/2 incorrect bits
• 30-50% faster than LA

#include <mkl_vml.h>

int main()
{
    double in[1000];
    double out[1000];
    vmlSetMode(VML_EP)
    vdExp(1000, int, out);
}

## Checklist

### C/C++

- Do you know restrict, const-ness and ansi-aliasing?
- Have you estimated aliasing impact (–no-alias)?
- Do you know about O2 vs. O3, IPO, and PGO?
- Did you optimization reports?
- Are you using pragmas?

### Fortran

- Do you know pure, elemental, and array sections?
- Are you using InterProcedural Optimization, or PGO?
- Are you using the -align arraynbyte option?
- Did you optimization reports?
- Did you ever use a directive e.g., non-temporal?
Resources

http://software.intel.com/mic-developer
• Developer’s Quick Start Guide
• Programming Overview
• New User Forum at


Intel® Composer XE 2013 for Linux* User and Reference Guides

Intel Premier Support  https://premier.intel.com
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 © 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core,
VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

**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