Extracting Vector Performance
with Intel Compilers

Hideki Saito and CJ Lin
SSG/DPD/MCC/ICL
Intel Corporation
May 25, 2012
Legal Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

This document contains information on products in the design phase of development.

Cilk, Core Inside, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, the Intel Sponsors of Tomorrow logo, Intel StrataFlash, Intel vPro, Itanium, Itanium Inside, MCS, MMX, Pentium, Pentium Inside, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, XMM, 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.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or 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
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code

• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
There are different kinds of parallelism

Across multiple apps

Across multiple processes

Across multiple threads

Across multiple instructions

SIMD (Single Instruction Multiple Data)

This presentation covers SIMD --- often times synonymous to “vector” ---.
What is SIMD?

float *restrict A, *B, *C;
for(i=0;i<n;i++)
    A[i] = B[i] + C[i];

Scalar Code computes this one-element at a time.

    addss xmm1, xmm2

Vector (or SIMD) Code computes more than one element at a time. SIMD stands for Single Instruction Multiple Data.

[SSE] 4 elems at a time
    addps xmm1, xmm2

[AVX] 8 elems at a time
    vaddps ymm1, ymm2, ymm3

[MIC] 16 elems at a time
    vaddps zmm1, zmm2, zmm3
History of SIMD ISA extensions

Intel® Pentium® processor (1993)

MMX™ (1997) Illustrated with the number of 32bit data processed by one “packed” instruction

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 2012)
Why do we want to vectorize the code?

CPU has a lot of computation power in the form of SIMD unit.

XMM (128bit) can operate
- 16x chars
- 8x shorts
- 4x dwords/floats
- 2x qwords/doubles/float complex

YMM (256bit) can operate
- 32x chars
- 16x shorts
- 8x dwords/floats
- 4x qwords/doubles/float complex
- 2x double complex

MIC (512bit) can operate
- 16x chars/shorts (converted to int)
- 16x dwords/floats
- 8x qwords/doubles/float complex
- 4x double complex
Why do we want to vectorize the code?

Sometimes, much more powerful than simply replacing scalar op into equivalent vector op

```c
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
#define SAT2SI16(x) \n    MAX(MIN((x),32767), -32768)

void foo1(int n, short *A, short *B){
    int i;
    #pragma ivdep
    #pragma vector aligned
    for (i=0; i<n; i++) {
    }
}
```

```
movsx  r11d, [rdx+r9*2]
movsx  ebx, [r8+r9*2]
add    r11d, ebx
cmp    r11d, 32767
cmovge r11d, eax
cmp    r11d, -32768
cmovl  r11d, ecx
movdqa xmm0, [rdx+r9*2], r11w
inc    r9
cmp    r9, r10
jb     .B1.8
```

```
movdqa xmm0, [rdx+rax*2]
paddsw xmm0, [r8+rax*2]
movdqa [rdx+rax*2], xmm0
add    rax, 8
cmp    rax, r9
jb     .B1.4
```

**Speedup on Core2Duo 2.33GHz**

11 insts / 1 elem

6 insts / 8 elems
SIMD Programming Methodology: 10000ft view

Automatically using the compiler [w/ or w/o hints]
Intel® Cilk™ Plus (SIMD pragma, Elemental Function, Array Notation)
Vectorized Libraries
(Intel® Integrated Performance Primitives, Intel® Math Kernel Library, etc)
SIMD class libraries (wrappers for vector intrinsics)
ASM, Vector Intrinsics

Ease of use/maintenance
Performance is not on the same scale
For same performance, choose improved long-term productivity
Applicability
Vectorizer Architecture

Input: C/C++/FORTRAN source code

- Fully Automatic Analysis
- Vectorization Hints (ivdep/vector pragmas)
- Array Notation
- Elemental Function
- SIMD pragma

Express/expose vector parallelism

Vectorizer

Map vector parallelism to vector ISA

- Intel® SSE
- Intel® AVX
- Intel® MIC

Optimize and Code Gen

Vectorizer makes retargeting easy!

Express/expose vector parallelism
Write vector code only once and just recompile for new target!

```c
#define ABS(X) \n    ((X) >= 0? (X) : -(X))
int A[1000]; double B[1000];
void foo(int n){
    int i;
    for (i=0; i<n; i++){
        B[i] += ABS(A[i]);
    }
}
```

```
vpabsd  xmm0, [A+r9+rax*4]
vcvtdq2pd ymm1, xmm0
vaddpd  ymm2, ymm1, [B+r9+rax*8]
vmovupd  [B+r9+rax*8], ymm2
add     rax, 4
cmp     rax, rcx
jb      .B1.4
```

```
movq  xmm1, [A+r9+rax*4]
pxor  xmm0, xmm0
pcmpgtq  ymm0, xmm1
pxor  xmm1, xmm0
psubd  xmm1, xmm0
cvtdq2pd  xmm2, xmm1
addpd  xmm2, [B+r9+rax*8]
movaps  [B+r9+rax*8], xmm2
add     rax, 2
cmp     rax, rcx
jb      .B1.4
```
Are we all motivated enough now?
Let’s get started!
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code
• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
Writing Vector Code (trivial example)

Array Notation

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

Elemental Function

__declspec(vector(uniform(B,C), linear(i:1)))

float foo(float *B, float *C, int i) {
    return B[i]+C[i];
}

- **SIMD Pragma**

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

- **Auto-Vectorization**

for(i=0; i<N; i++){
    for(i=0; i<N; i++){
        A[i] = B[i]+C[i];
    }
    A[i] = foo(B, C, i);
}
Different Ways to Vectorize --- Why?

Array Notation

• Vector statements at the source level conveying the programmer intent.
• Concise semantics match vector execution
• Better reflection of Array operation in the algorithm.

• **SIMD Pragma**
  – Suitable for keeping the original loop form to minimize the amount of change.

Elemental Function

• Already have functions that work on one element at a time
• All of them aren’t inlined
• Inline code too complex for auto-vectorizer

• **Auto-Vectorization**
  – Not sure vectorization is beneficial
  – Leave the decision to the compiler
  – Don’t have time to change all the code.
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code

• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
What is Cilk™ Plus Array Notation?

Example: \( A[:] = B[:] + C[:] \)

An extension to C/C++

Perform operations on sections of arrays in parallel

Well suited for code that:
• performs per-element operations on arrays,
• without an implied order between them
• with an intent to execute in vector instructions
Array Notation

Use a “:” in array subscripts to operate on *multiple elements*

- A[:]
- A[lower_bound : length]
- A[lower_bound : length : stride]
Accessing an entire array

```c
float a[10];
...
    = a[:];
...
```

a: 0 1 2 3 4 5 6 7 8 9
Accessing a section of an array

float b[10];

.. = b[2:6];

// section operands can be variables also

..
Section of 2D Array

```c
float c[10][10];
..
  = c[::][5];
..
```

C:
Strided Section

float d[10];
.
= d[0:3:2];
.

d: 0 1 2 3 4 5 6 7 8 9
Operations on Array Sections

C/C++ operators

\[ d[::] = a[::] + (b[::] \times c[::]) \]

Function calls

\[ b[::] = \text{foo}(a[::]); \quad // \text{Call foo()} \text{ on each element of } a[] \]

Reductions combine array elements to get a single result

\[ \quad // \text{Add all elements of } a[] \]
\[ \text{sum} = \_\_\_\_\text{sec\_reduce\_add}(a[::]); \]
\[ \quad // \text{More reductions exist...} \]

If-then-else and conditional operators allow masked operations

\[
\begin{align*}
\text{if (mask[::])} & \{ \\
& \quad a[::] = b[::]; \quad // \text{If mask}[i] \text{ is true, } a[i]=b[i] \\
\} 
\end{align*}
\]
“The Fine Print”

If array size is not known, both lower-bound and length must be specified

A loop will be generated if vectorization is not possible

Unaligned data, and lengths not divisible by 4, are vectorized as much as possible, with loops to handle leftovers

Section ranks and lengths (“shapes”) must match. Scalars are OK.

\[
a[0:5] = b[0:6]; \quad // \text{No. Size mismatch.}
\]

\[
a[0:5][0:4] = b[0:5]; \quad // \text{No. Rank mismatch.}
\]

\[
a[0:5] = b[0:5][0:5]; \quad // \text{No. No 2D->1D reduction.}
\]

\[
a[0:x] = b[0:y]; \quad // \text{OK; LHS used as vector length}
\]

\[
a[0:4] = 5; \quad // \text{OK. 4 elements of A filled w/ 5.}
\]

\[
a[0:4] = b[i]; \quad // \text{OK. Fill with scalar } b[i].
\]

\[
a[10][0:4] = b[1:4]; \quad // \text{OK. Both are 1D sections.}
\]

\[
b[i] = a[0:4]; \quad // \text{No. Use reduction intrinsic.}
\]
Conditional Expressions

“If statement” creates a masked vector operation

```
if (A[::] > 0) {
    B[::] = 1;
}
```

“For each” element of A that is >0, set the corresponding element of B to 1.
Conditional statements

All array section shapes (ranks and lengths) in the conditional, and the “then” and “else” parts should be the same.

Scalars are OK of course.

```c
// reduction has a scalar result
if (__sec_reduce_all_nonzero(A[:]>10))
{
    A[:] = 10; // unmasked, all elems
}
else if (A[:] > 10)
{
    A[:] = 10; // masked, some elems
}
```

```c
int P, A[10], B[10][10];
if (P > 0) A[:] = P; // OK
if (A[:] > 10) {
    A[:] = 10; // OK
    // shapes don’t match if-cond
    B[0:2][5] = B[2][0:2]; // FAIL
}
```
Function Maps

A scalar function call is mapped to the elements of array section parameters by default:

```
a[:] = sin(b[:]);
a[:] = pow(b[:], c); // b[:]**c
a[:] = pow(c, b[:]); // c**b[:]
a[:] = foo(b[:])     // user defined function
```

Functions are mapped in parallel
No specific order on side effects
Compiler will use vector versions of functions if possible
• Built-in intrinsics, or user-defined elemental vector functions
If no vector version of function available, the scalar function is called one at a time for each element
Reductions

Reduction combines array section elements to generate a scalar result

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum
    // is 10
```

Nine built-in reduction functions supporting basic C data-types:
- add, mul, max, max_ind, min, min_ind, all_zero, all_non_zero, any_nonzero

Can define your own reduction function

```
type fn(type in1, type in2); // scalar reduction function
out = __sec_reduce(fn, identity_value, in[x:y:z]);
```

- Identity value is the starting value of the compiler-generated accumulator
- Built-in reductions provide best performance
Simple example: Dot product

Serial version

```c
float dot_product(unsigned int size, float A[size], float B[size])
{
    int i;
    float dp=0.0f;
    for (i=0; i<size; i++) {
        dp += A[i] * B[i];
    }
    return dp;
}
```

Array Notation Version

```c
float dot_product(unsigned int size, float A[size], float B[size])
{
    return __sec_reduce_add(A[:] * B[:]);
}
```
Implicit Index

Create expressions containing the array index value

__sec_implicit_index(0) represents the 1st rank section index
__sec_implicit_index(1) represents the 2nd rank section index

Examples:

// fill A with values 0,1,2,3,4....
A[:] = __sec_implicit_index(0);

// fill B[i][j] with i+j
B[:, :] = __sec_implicit_index(0) + __sec_implicit_index(1);

// fill the lower-left triangle of C with 1
C[0:n][0:__sec_implicit_index(0)] = 1;
Gather/Scatter

Take non-consecutive array elements and “gather” them into consecutive locations, or vice-versa. The indices of interest are in an index array

Gather

\[ c[:] = a[b[:]]; \quad // \text{gather elements of } a \text{ into } c, \]
\[ \quad // \text{according to index array } b \]

Scatter

\[ a[b[:]] = c[:]; \quad // \text{scatter elements of } c \text{ into } a, \]
\[ \quad // \text{according to index array } b \]
### Shift/Rotate

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b[:] = __sec_shift_right(a[:], shift_val, fill_val)</code></td>
<td>Shift elements in <code>a[:]</code> to the right/left by <code>shift_val</code>. The leftmost/rightmost element will get <code>fill_val</code> assigned.</td>
</tr>
<tr>
<td><code>b[:] = __sec_shift_left(a[:], shift_val, fill_val)</code></td>
<td></td>
</tr>
<tr>
<td><code>b[:] = __sec_rotate_right(a[:], rotate_val)</code></td>
<td>Rotate will circular-shift elements in <code>a[:]</code> to the right/left by <code>rotate_val</code>. Result is assigned to <code>b[:]</code>.</td>
</tr>
<tr>
<td><code>b[:] = __sec_rotate_left(a[:], rotate_val)</code></td>
<td></td>
</tr>
</tbody>
</table>

- Shift elements in `a[:]` to the right/left by `shift_val`.
- The leftmost/rightmost element will get `fill_val` assigned.
- Rotate will circular-shift elements in `a[:]` to the right/left by `rotate_val`.
- Result is assigned to `b[:]`.
- Argument `a[:]` is not modified.
Fixed-size Array Sections

**Long vector coding**

A[0:N] = B[0:N] + C[0:N];
D[0:N] = E[0:N] + A[0:N];

This is visually appealing, but may not be high performing.

Similar to C loops:
for(i=0; i<N; i++){
    A[i] = B[i] + C[i];
}
for(i=0; i<N; i++){
    D[i] = E[i] + A[i];
}

Use short-vector coding if you have data reuse between statements and N is big.

**Short vector coding**

```c
#define VLEN 4
for(i=0; i<N; i+=VLEN){
}
```

Similar C loop:
for(i=0; i<N; i+=VLEN){
    for(j=0; j<VLEN; i++)
        A[i+j] = B[i+j] + C[i+j];
    for(j=0; j<VLEN; i++)
        D[i+j] = E[i+j] + A[i+j];
}
Vector and Thread Parallelism Together
Write function in array notation that handles a size-$m$ “chunk” of data
Call the function on multiple chunks of data, in parallel, using multi-threading

```c
void saxpy_vec(int m, float a, float x[m], float y[m])
{
    y[:] += a * x[:]; // Vector code for size m data
}

void main(void) {
    int a[2048], b[2048];
    cilk_for (int i = 0; i < 2048; i +=256) {
        // Call function on size-256 chunks in parallel
        saxpy_vec(256, 2.0, &a[i], &b[i]);
    }
}
```
Time for a little break

You must be itching to see a more elaborate array notation example. Hold on. We’ll get there soon enough.
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code
• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
What is Elemental Function?

Write a function for one element.
Add __declspec(vector) to get vector code for it.
__declspec(vector)
float foo(float a, float b, float c, float d) {
    return a * b + c * d;
}

and obtain

vmulps ymm0, ymm0, ymm1
vmulps ymm2, ymm2, ymm3
vaddps ymm0, ymm0, ymm2
ret

Call it from auto-vec or SIMD loop
for(i=0;i<n;i++){
    A[i] = foo(B[i], C[i], D[i], E[i]);
}

Call it from Array Notation
A[:] = foo(B[:], C[:], D[:], E[:]);

Call it from Elemental Function
__declspec(vector)
float bar(float a, float b, float c, float d){
    return sinf(foo(a,b,c,d));
}

Call scalar version from scalar code
e = foo(a, b, c, d);
Concept of Elemental Function

Elemental functions allow the programmer to use scalar syntax to describe an operation on a single element, then apply that operation to arrays in parallel, utilizing both vector parallelism and core parallelism.

Similar to CUDA/OpenCL style kernel functions

The programmer:
1. Writes a standard C/C++ function with a scalar syntax
2. Annotates it with __declspec(vector)
3. Uses one of the parallel syntax choices to invoke the function

The compiler:
1. generates a short vector version of the function, that can operate on multiple elements at each invocation.
2. Invokes the function iteratively, applying multiple elements at a time, until all elements are processed
3. The instances of the elemental function can either execute on a single core, or can use the task scheduler and execute on multiple cores
Elemental Function: simple example

```c
__declspec(vector(linear(p:1)))
float add2(float *p){ return *p+2; }

for(i){ // caller side
  A[i] = add2(p+i);
}

Compare above to:
__declspec(vector)
float add2x(float *p){ return *p+2; }

And also to:
__declspec(vector)
float add2y(float x){ return x+2; }
```

Scalar code
- `movss xmm0, [_2il0floatpacket.6]`
- `addss xmm0, [rcx]`

Unmasked vector code
- `movups xmm0, [rax]`
- `addps xmm0, [_2il0floatpacket.1]`

Masked vector code
- `movups xmm1, [rax]`
- `addps xmm1, [_2il0floatpacket.4]`
- `andps xmm1, xmm0`
- `movaps xmm0, xmm1`
Elemental Function: syntax

__declspec(vector) // or:
__attribute__((vector))

Add optional clauses:

processor(name)
• pentium_4_sse3, ..., core_i7_sse4_2, core_i7_avx

vectorlength (num)
• vectorlengthfor(TYPE)
• If not set VL is selected by return value, then by args

linear (var:stride)

uniform (var)

mask/nomask

Multipledeclspecs are allowed for one function

Known limitations:

No indirect calls

No structs passed by value (by
ref – OK)

No synchronization

No “unsafe” calls

No EH constructs

No threading constructs
• E.g. _Cilk_spawn/_Cilk_for

No global scalar writes
**Vector Length**

How many vectorized copies of the function should execute together per function call?

As many as you can fit into the HW vector register

Constraints: this ratio must be determined consistently yet independently for the function declaration and its callers → cannot rely on the code inside the function, only return type and parameters

The cases of v_add_f and v_add_d are handled as expected.

In “oops”, most of the time is being spent in single precision, but the compiler cannot automatically use it as the “characteristic type” of the function

The clauses vectorlength and vectorlengthfor are provided for override

```c
__declspec(vector)
float v_add_f(float b, float c)
{
    return b+c;
}

__declspec(vector)
double v_add_d(double b, double c)
{
    return b+c;
}

__declspec(vector)
double oops(double e, double f)
{
    return 
sinf(float(e)*sinf(float(f))
}
```
**Elemental Function: Uniform/Linear clauses**

Why do we need them?

- Because “vector” loads and stores of IA chips are optimized for accessing immediately next elements in memory (e.g., [v]movups).

They are most useful when consumed in the address computation.

```c
__declspec(vector(uniform (a)))
void foo(float *a, int i);
```

- a is a pointer
- i is a vector of integers
- a[i] becomes gather/scatter.

```c
__declspec(vector(linear(i)))
void foo(float *a, int i);
```

- a is a vector of pointers
- i is a sequence of integers [i, i+1, i+2...]
- a[i] becomes gather/scatter.

```c
__declspec(vector(uniform(a),linear(i)))
void foo(float *a, int i);
```

- a is a pointer
- i is a sequence of integers [i, i+1, i+2...]
- a[i] is a unit-stride load/store ([v]movups).
Multiple versions

Multiple `declspec(vector)` lines are allowed for a single function

Each will result in another compiled version of the function

Example: the same function may be called with uniform / non uniform arguments

Avoiding the second line will deliver correct results but lose performance

If only the line with uniform is given, then for call sites where the actual arguments are not uniform, the compiler will call the scalar, not vector, version of the function!

```c
__declspec(vector)
__declspec(vector(uniform(b,c))

float vmul(float a, float b, float c)
{
    return sqrt(a)*sqrt(b) +
            sqrt(a)*sqrt(c) +
            sqrt(b)*sqrt(c);
}
```
## Invoking Elemental Functions

<table>
<thead>
<tr>
<th>Construct</th>
<th>Example</th>
<th>Semantics</th>
</tr>
</thead>
</table>
| Standard for loop | for (j = 0; j < N; j++) {  
  a[j] = my_ef(b[j]);  
}                                                                     | Single thread, auto vectorization                   |
| #pragma simd      | #pragma simd  
for (j = 0; j < N; j++) {  
  a[j] = my_ef(b[j]);  
}                                                                     | Single thread, Guaranteed to use the vector version |
| cilk for loop     | cilk_for (j = 0; j < N; j++) {  
  a[j] = my_ef(b[j]);  
}                                                                     | Both vectorization and concurrent execution          |
| Array notation    | a[:] = my_ef(b[:]);                                                   | Vectorization. Automatic concurrency is TBD          |
Array Notation calling Elemental Function

```c
// Plain C scalar function declared with __declspec(vector)
__declspec(vector) float saxpy(float a, float x, float y) {
    return (a * x + y);
}

Z[:] = saxpy(A, X[:], Y[:]); // Call scalar function with // array notation parameters
```

Compiler generates vector version of saxpy that processes 4/8/16 elements in parallel, using vector registers

saxpy is passed groups of elements from Z/X/Y in vector registers

Similar to sin(x[:]), but the user creates the function

Without __declspec(vector) tag, the function will be generated as a normal scalar routine

• Could still vectorize if it is inlined to the call site
Elemental Function and Inlining

Elemental functions are subject to automatic inlining decision, just like any function. Once inlined at a call site, “elemental function” notion is gone at that call site.

• Use explicit “no inline” technique if you do not want inlining.

```c
#pragma inline[recursive]
#pragma forceinline[recursive]
#pragma noinline
```

Function (elemental or not) w/ a loop isn’t inlined into a SIMD loop.

• Use explicit “force inline” technique if you want inlining.

```c
#pragma simd
for(i=0;i<n;i++){
    A[i] = foo(i);
}
__declspec(vector)
float foo(int n) {
    for (int i=0;i<n;i++){
        ....
    }
    ....
}
```
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code
• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
Pragma SIMD

#include <vector>

#pragma simd
for (int ray=0; ray < N; ray++) {
  float Color=0.0f, Opacity=0.0f;
  int len=0;
  int upper = raylen[ray];
  while (len < upper) {
    int voxel=ray+len;
    len++;
    if (visible[voxel]==0) continue;
    float O=opacity[voxel];
    if (O==0.0) continue;
    float Shading=O+1.0;
    Color += Shading*(1.0f-Opacity);
    Opacity+=O*(1.0f-Opacity);
    if (Opacity > THRESH) break;
  }
  color_out[ray]=Color;
}

OpenMP-like pragma for vector programming.

“go ahead and generate vector code” model

Additional semantics (private, reduction, linear, etc.) given to compiler via clauses.

<table>
<thead>
<tr>
<th>directive</th>
<th>hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>SIMD</td>
</tr>
<tr>
<td>thread</td>
<td>OpenMP</td>
</tr>
</tbody>
</table>
SIMD Pragma: definition

Top-level
• #pragma simd
• !DIR$ SIMD

Attached clauses to describe semantics / aid code generation
• vectorlength(VL)/vectorlengthfor(TYPE)
• private/firstprivate/lastprivate(var1[, var2, …])
• reduction(oper1:var1[, …][, oper2:var2[, …]])
• linear(var1[:step1][, var2[:step2], …])
• [no]assert
SIMD Pragma: simple examples

```c
void foo(int *A, int N, int n){
    int i;
    #pragma simd
    vectorlength(4)
    for (i=0; i<n; i++){
    }
}
```

#pragma simd not applicable if “0 < N < n”, but vectorization is still possible if N isn’t too small.

```c
short sum(float *A, int n){
    int i; short x = 0;
    #pragma simd
    reduction(+:x)
    for (i=0; i<n; i++){
        xt = x + A[i]*2
        x = xt + N;
    }
    return x;
}
```

Tell compiler “x” has sum-reduction semantics
Time for another break
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code
• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
Mandelbrot --- serial version

```cpp
for (i=0; i<n; i++) {
    complex<float> c = a[i];
    complex<float> z = c;
    int j = 0;
    while ((j < 255) && (abs(z)< limit)) {
        z = z*z + c;
        j++;
    }
    color[i] = j;
}
```
Mandelbrot with Array Notation

```c
void mandel(int n, complex c[],int cnt[],int max_cnt) {
    Complex<float> z[n]; int test[n];
    z[0:n] = 0;
    // Count how many times it takes to saturate each z
    for (int i = 0; i < max_cnt; i++) {
        test[0:n] = (abs(z[0:n] < 2.0);
        if (0 == __sec_reduce_add(test[0:n]))
            break;
        cnt[0:n] += test[0:n];
        z[0:n] = z[0:n]*z[0:n] + c[0:n];
    }
}

// Outer loop, could parallelize it with cilk_for
for (int i = 0; i < max_row; i++) {
    for (int j = 0; j < max_col; j+=8 ) {
        // Mandelbrot on 8 points at a time
        mandel(8, p[i]+j, points[i]+j, depth);
    }
}
```

If all points are saturated, quit

cnt[] is the result
Mandelbrot as an Elemental Function

```c
__declspec(vector(vectorlength(4),uniform(max_count)))
static float mandel(float re, float im, int max_count){
    int count=0, k;
    float zre=0, zim=0, zre1=0, absZ=0;
    for (k=0; k<max_count; k++){
        absZ=zre*zre+zim*zim;
        if (absZ >= 4.0f) { break; } 
        zre1=zre*zre-zim*zim+re;
        zim=2*zre*zim+im; zre=zre1; count++;
    }
    return count;
}
```

- `max_count` is uniform
- `re, im` are float[4]
- `count, zre, zim` and `absZ` are float[4]
- `k` iterates on 4 lanes simultaneously
- `break` masks out lanes, for which condition is met. If all lanes are masked the loop finishes
- masked version of function is also created for conditional call

```c
#pragma simd
for (j=0; j<NUM_COLS; j++)
res[i*NUM_COLS+j] = mandel(scale[i:4:], scale[j:4:], MAX_CNT);
```
Mandelbrot with `#pragma simd`

```cpp
#pragma simd
for (i=0; i<n; i++) {
    complex<float> c = a[i];
    complex<float> z = c;
    int j = 0;
    while ((j < 255) && (abs(z)< limit)) {
        z = z*z + c;
        j++;
    }
    color[i] = j;
}
```

Note: The pragma is applied to an outer loop with a while loop inside
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code
• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
Loop Invariant Inside the Loop

Loop invariant hoisting is harder for the compiler than a typical programmer might think.

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

Test case is intentionally contrived to have assumed alias.

```assembly
..B1.12:
    movss (%r15), %xmm0
    shufps $0, %xmm0, %xmm0
    addps (%rbp), %xmm0
    call __svml_sinf4
    addq $4, %r13
    movaps %xmm0, (%rbp)
    addq $16, %rbp
    cmpq %rbx, %r13
    jb ..B1.12
```

This is a bug we fixed in 13.0, but you get the idea.
Loop Invariant Inside the Loop (cont)

Even after that fix, don’t simply assume compiler can pull invariants out of loop.
Compiler can’t really tell a[0] is safe to read from, before the loop.

```c
float xx = a[0];
#pragma ivdep
for (i=0; i<n; i++){
    float x = 0.0f;
    if (b[i] > 0) {
        x = xx;
    }
    b[i] = sinf(x+b[i]);
}
```

```c
void foo1(float *a, float *b, int n){
    int i;
    #pragma ivdep
    for (i=0; i<n; i++){
        float x = 0.0f;
        if (b[i] > 0) {
            x = a[0];
        }
        b[i] = sinf(x+b[i]);
    }
}
```
Q-index gather/scatter versus D-index

Gather/Scatter: *(base + index_vector) or *(pointer_vector).
D-index: index is 32bit.
Q-index: index is 64bit.
D-index is more desirable
• MIC has HW support.
• vpgatherdd/vgatherdps fetch 2x more elements than vpgatherqd/ vgatherqps (AVX2).
• Fewer instructions on index compute

On ia32, it’s hard to get Q-index gather/scatter even if index is 64bit.
On intel64, it’s easy to fall into Q-index gather/scatter.
• Q-index if compiler can’t prove D-index is okay.

Unsigned 32bit indices?
• Typically harder to deal with. Signed is usually more optimizable.
Q-index gather/scatter versus D-index

```c
void foo0(int **p1, int *restrict p, int n){
    int i;
    #pragma vector always
    for (i=0; i<n; i++){
        p[i] = *(p1[i]); // Q-index
    }
}

void foo1(int *p1, int *p2, int *restrict p, int n){
    int i;
    #pragma vector always
    for (i=0; i<n; i++){
        p[i] = p1[p2[i]]; // D-index
    }
}

void foo2(int n, int *p1[n], int p[n]){
    p[:] = *(p1[:]); // Q-index
}

void foo3(int n, int p1[n], int p2[n], int p[n]){
    p[:] = p1[p2[:]]; // D-index
}

__declspec(vector)
int foo4(int *p){
    return *p; // Q-index
}

__declspec(vector(uniform(p)))
int foo5(int *p, int i){
    return p[i]; // Q-index...
}
```
Q-index gather/scatter versus D-index

struct X {
    float p, q;
}
X A[1000];
int inx[1000];
for(i) { // vectorize here!
    ... A[inx[i]].p ... // D-inx
    ... A[inx[i]].q ... // D-inx
}

for(i) { // vectorize here!
    X &a = A[inx[i]];
    ... a.p ... // Q-inx
    ... a.q ... // Q-inx
}

for(i) { // vectorize here!
    X *a = &A[inx[i]];
    ... a->p ... // Q-inx
    ... a->q ... // Q-inx
}
Letting the compiler know about unit-stride is very important.

```c
void foo(float *p, int x, int n) {
    int i;
    for (i=0; i<n; i+=x) {
        p[i]++;
    }
}
```

This is strided even if x is always 1 at runtime.

Also, compiler thinks x could be zero.

Optimize if x is frequently 1.

```c
void foo(float *p, int x, int n) {
    int i;
    if (x==1) {
        for (i=0; i<n; i++) {
            p[i]++; // unit stride!
        }
    } else {
        for (i=0; i<n; i+=x) {
            p[i]++;
        }
    }
}
```
Yes, Data Alignment Still Matters

Core2 and prior: not-aligned XMM load/store used to be broken into 8B/8B.

1st gen Core i3/5/7: No longer 8B/8B, vector load+op still needs aligned memref

movups (%rdi,%rax,4), %xmm1
addps %xmm0, %xmm1

2nd/3rd gen Core i3/5/7: not-aligned YMM load/store is broken into 16B/16B

vmovups (%rdi,%r8,4), %xmm1
vinsertf128 $1, 16(%rdi,%r8,4),
%ymm1, %ymm2
vaddps %ymm2, %ymm0, %ymm3

MIC: Aligned 512bit load/store is 1 instruction

All: aligned vector load/store is usually faster than non-aligned.
Did you align at allocation point?

Good Alignment
• 16B: Pentium4 to 1\textsuperscript{st} gen Core i7
• 32B: 2\textsuperscript{nd}/3\textsuperscript{rd} gen Core i7
• 64B: MIC

// Windows syntax
__declspec(align(16))
float A[1000];

// Linux syntax
__attribute__((aligned(16)))
float A[1000];

Aligned malloc
• _aligned_malloc()
• _mm_malloc()

Or you can use malloc() and massage the pointer yourself.

p1 = malloc(size + 16);
// allocate a bit more!
ptr = p1 & 0xFF…F0;
// massage
....
// use of ptr
....
free(p1); // free with right pointer.
Did you tell “pointer is aligned”?

Often times, you get better code if you also let the compiler know data is aligned.

• 3 version peel/vector/remainder becomes 2-version (w/o peel).
• Load+op is used
• Vector memref becomes one instruction instead of multiple.

__assume_aligned(p,16);
• p mod 16 = 0 at this point in the program.

memrefs are all aligned for the loop

#pragma vector aligned
for(i=0;i<n;i++) {
    A[i+t1]=B[i+t2]*C[i+t3];
}
Did you tell “we have good offset”?

I have too many loops for adding “vector aligned” pragmas. What do I do?

```c
for(i=0;i<n;i++) {
    p1[i+t1]=p2[i+t2]*p3[i+t3];
}
...
for(i=0;i<n;i++) {
    p1[i]=p2[i]*p3[i];
}
```

Use `__assume_aligned()` and `__assume()`.

```c
__assume_aligned(p1,16);
__assume_aligned(p2,16);
__assume_aligned(p3,16);
__assume(t1%4==0);
__assume(t2%4==0);
__assume(t3%4==0);
```

// We have a bug in C++
// __assume() handling.
// Until it’s fixed, use this.
`__assume_aligned((int*)t1, 4)`
Are we done if the code is vectorized?

Certainly not.

Vectorization is just one of the steps for optimizing your code.

Unit-stride load/store are more efficient than strided load/store or gather/scatter.

Q-index gather/scatter versus D-index?

Compiler successful in moving loop-invariant computation outside of loop?

Unit-stride load/store are better performing if aligned

Unnecessary converts between float/double?
Performance Tuning

Vectorizing the code is not the end of the story.

It’s actually the beginning: vector code has to be efficient, rather than inefficient vector code.

Disambiguation
- Not just for getting the loop vectorized.
- Can’t move load/store freely if aliased.

Alignment

Unroll

Trading Off Numerical Precision and Performance

Compiler has plenty of knobs to play with if you want to.
Align your data AND tell the compiler!!

Good array data alignment for
- Pentium 4 to Core i7: 16B
- AVX: 32B
- MIC: 64B

Data alignment directive (16B example)
- C/C++
  Windows: __declspec(align(16)) float A[1000];
  Linux/MacOS: float A[1000] __attribute__((aligned (16)));
- Fortran
  !DIR$ ATTRIBUTES ALIGN: 16:: A

Aligned malloc
- __aligned_malloc()
- _mm_malloc()

Data alignment assertion (16B example)
- C/C++:
  __assume_aligned(p,16);
- Fortran:  !DIR$ ASSUME_ALIGN ED A(1):16

Multiple of good number
- __assume(n%4==0)

Aligned loop assertion
- C/C++: #pragma vector aligned
- Fortran: !DIR$ VECTOR ALIGNED

Align your data AND tell the compiler!!
Align your data AND tell the compiler!!

Good array data alignment

- Pentium 4 to Core i7: 16
- AVX: 32B
- MIC: 64B

Data alignment directive (16B example)

- C/C++
  - Windows: __declspec(align(16)) float A[1000];
  - Linux/MacOS: float A[1000] __attribute__((aligned(16)));
- Fortran
  - _DIR$ ATTRIBUTES ALIGN: 16:: A
- _aligned_malloc()
- _mm_malloc() (16)

Data alignment assertion (16B example)

- C/C++:
  - __assume_aligned(p, 16);
- Fortran: _DIR$ ASSUME_ALIGN

Multiple of good number

- __assume(n%4==0)

Aligned loop assertion

- C/C++: #pragma vector aligned
- Fortran: _DIR$ VECTOR ALIGNED

Compiler will know these vector memrefs are all 16B aligned.
Help compiler with memref analysis

Do not change the base pointer. Loop-invariant pointers are easier for the compiler.

This can easily confuse the compiler as the code complexity increases.

```c
float *restrict p1 = ... 
float *restrict q1 = ... 
for(i){
    p1[i] = ... q1[i]... 
    // or
    *(p1 + ... i ...) = 
    .... *(q1 + ... i ...)
}
```
Help compiler with loop trip count

This can easily confuse the compiler as the code complexity increases --- size() could be reading some heap memory that is “considered” aliased.

No-brainer way to tell the compiler that the loop bounds don’t change in the loop no matter what the aliasing situation is.

```c
for(i=0; i<size(); i++){
    ...
}

int trip = size();
for(i=0; i<trip; i++){
    ...
}
```
Common Pitfalls

Even if all of your variables are float, 1.0 and sin(), for example, are still double precision in C/C++.

• Use 1.0f and sinf() if you are writing single precision code.

• Compiler fixes your perf bug but that’s not always possible ➔ lots of FP converts.

```c
float x[N], y[N];
for (i = 0; i < N; i++)
    y[i] = sin(x[i] + 1.0) + 1.0;
```
Common Pitfalls

Even if all of your variables are float, 1.0 and sin(), for example, are still double precision in C/C++.

- Use 1.0f and sinf() if you are writing single precision code.
- Compiler fixes your perf bug but that’s not always possible ➔ lots of FP converts.

```c
float x[N], y[N];
for (i = 0; i < N; i++)
    y[i] = sin(x[i] + 1.0) + 1.0
```
Common Pitfalls

Even if all of your variables are float, 1.0 and sin(), for example, are still double precision in C/C++.

- Use 1.0f and sinf() if you are writing single precision code.
- Compiler fixes your perf bug but that’s not always possible → lots of FP converts.

```
float x[N], y[N];
for (i = 0; i < N; i++)
    y[i] = sinf(x[i] + 1.0f) + 1.0f
```

“unsigned int” is different from “int” + non-negative.

- “unsigned int” allows wraparound.
  - max_uint+1=zero.
  - Difficult for the optimizers to deal with
- Wraparound of “int” is undefined.
  - Compiler is allowed to assume values won’t wraparound.
  - Can assume easier to optimize contiguous range of values.

```
for(i=0;i<N;i++)
y[i]=sinf(x[i]+1.0f)+1.0f
```
Common Pitfalls

Even if all of your variables are float, 1.0 and sin(), for example, are still double precision in C/C++.

- Use 1.0f and sinf() if you are writing single precision code.
- Compiler fixes your perf bug but that’s not always possible → lots of FP converts.

“unsigned int” is different from “int” + non-negative.

- “unsigned int” allows wraparound.
  - max_uint+1=zero.
  - Difficult for the optimizers to deal with
- Wraparound of “int” is undefined.
  - Compiler assumes values won’t wraparound.
  - Can assume contiguous range of values.

```c
float x[N], y[N];
for (i = 0; i < N; i++)
    y[i] = sin(x[i] + 1.0) + 1.0
```

Please use signed int.
Common Pitfalls (cont)

Pointer is 64bit on Intel64
Avoid pointer increments
• Better for dependence analysis too
Long is 64bit on
Intel64 platforms running Linux
Mixing 64bit and 32bit integers
leads to
• Sign extend/Zero extend
• Truncation
Also, packed 64bit integer HW support
is not as good as 32bit.
• Lower compute bandwidth
• Emulation code, or loss of vectorization

for (i = 0; i < N; i++)
  *x++ = *y++;

for (i = 0; i < N; i++)
  x[i] = y[i];

float *p1=...; *p2=...
int i=...
p1 = p1 + i; p1=p1 + i;
p2 = p2+2*i;

p1 != p2
compiler assumes
the worst case
void foo(float *a, float *b1, 
          float *b2, float *b3, int n){
    int i;
    for (i=0; i<n; i++) {
        a[i] += b1[i]+b2[i]+b3[i];
    }
}

Compiler has to think a[] is aliased to b1[], b2[], and b3[].
• Use “float *restrict a”

Use restrict pointer (C/C++, requires –restrict flag)

#pragma IVDEP (C/C++), !DIR$ IVDEP (FORTRAN)

-ansi-alias flag (especially on C++ code, but helps on C, too)
• Enables data type based disambiguation (e.g., stores to double don’t change pointer)
Numerical Precision and Performance

Compiler default is “fast-mode” --- one-size-fits-all.

Math Libs

• Scalar is 0.5ulp
• Vector is 4ulp
• Controllable by
  - -fimf-absolute-error=value[ :funclist]
  - -fimf-accuracy-bits=bits[ :funclist]
  - -fimf-arch-consistency=value[ :funclist]
  - -fimf-max-error=ulps[ :funclist]
  - -fimf-precision=value[ :funclist]
  - -fimf-domain-exclusion=classlist[ :funclist]

32b FP divide

• Scalar is divss
• Vector is N-R with rcpps

Control DIV and SQRT precision

-[no-]prec-div (Linux/MacOS)
-[no-]prec-sqrt (Linux/MacOS)
/Qprec-div[-] (Windows)
/Qprec-sqrt[-] (Windows)

• Rewrite “/val” as “*(1/val)”
Agenda

Introduction to Vectorization and Vectorizer

Ways to Write Vector Code
• Array Notation
• Elemental Function
• Loop Vectorization

Case Study

Optimizations Beyond Vectorizing the Code

Summary
Summary

We have learned

• Why vectorization is important.
• How the vectorizer helps improve productivity
• Vectorizing with Array Notation
• Vectorizing with Elemental Function
• Vectorizing with SIMD Pragma
• Writing optimizable code that enables more optimization, including auto-vectorization
• Optimizing beyond just getting vector code
Call to Action

Get back to your application code

Find the hot spots that are vectorizable

Apply one or more techniques learned today
Thank You!