Vanquishing SIMD Hurdles: Look Back and Forward - What are we up to?

Xinmin Tian
Principal Engineer
Mobile Computing and Compilers, DPD/SSG
Intel Corporation, Santa Clara, CA

The 27th Workshop on Languages and Compilers for Parallel Computing 2014 - Keynote
Agenda

Why SIMD?

Taking a Look Back: Cray* and Intel® Pentium® 4 (90nm) SSE3

Vanquishing SIMD Parallelism Hurdles
- Function vectorization
- Outer loop vectorization
- Mixed data type vectorization
- Sophisticated Idioms Vectorization
- Less-than-full-vector Vectorization
- Alignment Optimizations for Intel® MIC architecture
- Small Matrix 2-D Vectorization

Xeon Phi™ Suitability -- Keys for High Performance

Looking Forward: What are we up to?

Recipe and Vision: Close to Medal Performance
Why SIMD? SIMD Scaling vs. Frequency Scaling

- Wider SIMD – Linear increase in area and power
- Wider superscalar – Quadratic increase in area and power
- Higher frequency – Cubic increase in power

With SIMD we can go faster with less power!
All future processors will have multiple cores with SIMD instructions.
Look back three and half decades!
Look back: Compiler Vectorization in 1978

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

CACM 1978

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

Livermore loop #1
Small loop, simple data and control flow

Compiler auto-vectorization becomes reality through dependency analysis

Compiler vectorization “solved” in 1978
2004 Intel® Pentium® 4 SSE3 on 90nm

Complex Multiplication with SSE3: \((a + ib)(c + id) = (ac - bd) + i(ad + bc)\)

Memory

Ops:
- 3 SIMD loads
- 1 SIMD store
- 3 Arithmetic Ops
- 1 shuffle Ops

Ops not available in SSE2
- movddup
- addsubpd

Performance can be improved up to \(~75\%\), SPEC2000FP/168.wupwise 10-15\%
Vanquishing SIMD Hurdles!
SIMD Vectorization Hurdles

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

Two fundamental problems
Data divergence
Control divergence

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

#pragma omp declare simd
float sfoo(float x)
{
    ... ...
}

Scalar C function
sfoo(x0)->r0  sfoo(x1)->r1  sfoo(x2)->r2  sfoo(x3)->r3
sfoo(x4)->r4  sfoo(x5)->r5  sfoo(x6)->r6  sfoo(x7)->r7
... ...
Scalar execution
vfoo(x0...x3)->r0...r3
vfoo(X4...X7)->r4...r7
... ...
Vector execution

_vector128 vfoo(_vector128 vx)
{
    ... ...
}

Vector C function
Vortex Code: Using SIMD-Enabled Function

#pragma omp simd // simd pragma for outer-loop at call-site of SIMD-function
for (int i = beg*16; i < end*16; ++i)
    particleVelocity_block(px[i], py[i], pz[i],
                          destvx + i, destvy + i, destvz + i, vel_block_start, vel_block_end);

#pragma omp declare simd linear(velx, vely, velz) uniform(start, end) aligned(velx:64, vely:64, velz:64)
static void particleVelocity_block(const float posx, const float posy, const float posz,
                                  float *velx, float *vely, float *velz, int start, int end) {

    for (int j = start; j < end; ++j) {
        const float del_p_x = posx - px[j];
        const float del_p_y = posy - py[j];
        const float del_p_z = posz - pz[j];
        const float dxn = del_p_x * del_p_x + del_p_y * del_p_y + del_p_z * del_p_z + pa[j] * pa[j];
        const float dxctau_i = del_p_y * tz[j] - ty[j] * del_p_z;
        const float dyctau_i = del_p_z * tx[j] - tz[j] * del_p_x;
        const float dzctau_i = del_p_x * ty[j] - tx[j] * del_p_y;
        const float dst = 1.0f / std::sqrt(dxn);
        const float dst3 = dst * dst * dst;
        *velx -= dxctau_i * dst3;
        *vely -= dyctau_i * dst3;
        *velz -= dzctau_i * dst3;
    }
}
Recursive Function Vectorization

```c
#pragma omp declare simd
int binsearch(int key, int lo, int hi) {
    int ans;
    if (lo > hi) {
        ans = -1;
    } else {
        int mid = lo + ((hi - lo) >> 1);
        int t = sortedarr[mid];
        if (key == t) {
            ans = mid;
        } else if (key > t) {
            ans = binsearch(key, mid + 1, hi);
        } else {
            ans = binsearch(key, lo, mid - 1);
        }
    }
    return ans;
}
#pragma omp simd
for (int i=0; i<M; i++) {
    ans[i] = binsearch(keys[i], 0, N-1);
};
```
typedef float complex fcomplex;
const uint32_t max_iter = 3000;
#pragma omp declare simd uniform(max_iter), simdlen(16)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
    uint32_t count = 1; fcomplex z = c;
    while ((cabsf(z) < 2.0f) && (count < max_iter)) {
        z = z * z + c; count++;
    }
    return count;
}
uint32_t count[ImageWidth][ImageHeight];

...... .... ........
for (int32_t y = 0; y < ImageHeight; ++y) {
    float c_im = max_imag - y * imag_factor;
    #pragma omp simd safelen(16)
    for (int32_t x = 0; x < ImageWidth; ++x) {
        fcomplex in_vals_tmp = (min_real + x * real_factor) + (c_im * 1.0iF);
        count[y][x] = mandel(in_vals_tmp, max_iter);
    }
}
... ... .... ....

Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows* Server 2012 R2 Datacenter.
Compiler options: ; SSE4.2: -O3 –Qipo –QxSSE4.2 or AVX2: -O3 –Qipo –QxCORE-AVX2. For more information go to http://www.intel.com/performance
Mandelbrot: ~2000x Speedup on Xeon Phi™ -- Isn’t it Cool?

```c
#pragma omp declare simd uniform(max_iter), simdlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
  uint32_t count = 1; fcomplex z = c;
  while ((cabsf(z) < 2.0f) && (count < max_iter)) {
    z = z * z + c; count++;
  }
  return count;
}
```

Intel Xeon Phi™ system, Linux64, 61 cores running 244 threads at 1GHz; 32 KB L1, 512 KB L2 per core. Intel C/C++ Compiler Internal build.

Mandelbrot Normalized Speedup with OMP PAR+SIMD on Xeon Phi(TM)
Outer Loop Vectorization

**SPEC CPU2006 410.bwaves**

```plaintext
do l=1,nb
    y(l,i,j,k)=0.0d0
  do m=1,nb
    y(l,i,j,k)=y(l,i,j,k)
    +a(l,m,i,j,k)*x(m,i,j,k)
    +axp(l,m,i,j,k)*x(m,ip1,j,k)
    +ayp(l,m,i,j,k)*x(m,i,jp1,k)
    +azp(l,m,i,j,k)*x(m,i,jp1,k)
    +axm(l,m,i,j,k)*x(m,im1,j,k)
    +aym(l,m,i,j,k)*x(m,i,jm1,k)
    +azm(l,m,i,j,k)*x(m,i,jkm1)
  enddo
enddo
```

**Inner m-loop** is not fit for vectorization --- 7 strided + 7 unit-stride mem-refs

**Outer l-loop** is better vectorization candidate --- 9 unit-stride mem-refs and 7 broadcasts
C++ Explicit Vectorization via OpenMP 4.0 SIMD

```c
#pragma omp declare simd linear(z:40) uniform(L, N, Nmat) linear(k)
float path_calc(float *z, float L[][VLEN], int k, int N, int Nmat)

#pragma omp declare simd uniform(L, N, Nopt, Nmat) linear(k)
float portfolio(float L[][VLEN], int k, int N, int Nopt, int Nmat)

... ... ...
for (path=0; path<NPATH; path+=VLEN) {
    /* Initialise forward rates */
    z = z0 + path * Nmat;
    #pragma omp simd linear(z:Nmat)
    for(int k=0; k < VLEN; k++) {
        for(i=0;i<N;i++) {
            L[i][k] = L0[i];
        }
        /* LIBOR path calculation */
        float temp = path_calc(z, L, k, N, Nmat);
        v[k+path] = portfolio(L, k, N, Nopt, Nmat);
    }
    /* move pointer to start of next block */
    z += Nmat;
}
... ... ...
```

Libor (Normalized)

Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows® Server 2012 R2 Datacenter. Compiler options; SSE4.2: -O3 -Qipo -QxSSE4.2 or AVX2: -O3 -Qipo -QxCORE-AVX2. For more information go to http://www.intel.com/performance
C++ Explicit Vectorization via OpenMP 4.0 SIMD / Cilk™ Plus: SIMD Performance

Normalized SIMD Speedup on Intel® Xeon® CPU

Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threadining On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows® Server 2012 R2 Datacenter. Compiler options: -O3 –Qipo –QxSSE4.2 or AVX2: -O3 –Qipo –QxCORE-AVX2. For more information go to http://www.intel.com/performance
Mixed Data Type Vectorization

```c
void foo(int n, float *A, double *B){
    int i;
    float t = 0.0f;
    #pragma ivdep
    for (i=0; i<n; i++) {
        A[i] = t; B[i] = t; t += 1.0f;
    }
}
```

Naïve: use full vectors 4 != 2. Give up (bad)

Match the number of elements.
- A[i] = ... for 2 or 4 elements at a time
- B[i] = ... for 2 or 4 elements at a time

mixed.c(5) (col. 3): remark: LOOP WAS VECTORIZED.

Match: 2=2. Good
Match: 4=2x2. Good
int index_0 = 0;
for(int k0=0; k0<count0; k0++) {
    TYPE X1 = *(Pos0 + k0);
    TYPE Y1 = *(Pos0 + k0 + count0);
    TYPE Z1 = *(Pos0 + k0 + 2*count0);
    #pragma loop_count min(220) avg (300) max (380)
    #pragma ivdep
    for(int k1=0; k1<count1; k1+=1) {
        TYPE X0 = *(Pos1 + k1);
        TYPE Y0 = *(Pos1 + k1 + count1);
        TYPE Z0 = *(Pos1 + k1 + 2*count1);
        TYPE diff_X = (X0 - X1);
        TYPE diff_Y = (Y0 - Y1);
        TYPE diff_Z = (Z0 - Z1);
        TYPE norm_2 = (diff_X*diff_X) + (diff_Y*diff_Y) + (diff_Z*diff_Z);
        if ( (norm_2 >= rmin_2) && (norm_2 <= rmax_2))
            Packed[index_0++] = norm_2;
    }
}

- Performance gain close to 10X with SIMD vectorization using vcompress
- Index_0 is getting updated under a condition – not linear (named it as monotonic index)
  - Currently this cannot be expressed using simd clauses
  - Users can use __simd_compress_*(...) functors for SIMD loop.
  - Extensions to simd-clause-syntax to express this idiom is WIP
float foo(float *y, int n) {
    int k; float x = 10.0f;
    for (k = 0; k < n; k++) {
        x = x + fsqrt(y[k]);
    }
    return x;
}

misalign = &y[0] & 63
peeledTripCount = (63 - misalign)/sizeof(float)
x = 10.0f;
do k0 = 0, peeledTripCount-1 // peeling loop
    x = x + fsqrt(y[k0])
endo
x1_v512 = (m512)0
x2_v512 = (m512)0
mainTripCount = n - ((n - peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
    x1_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1:16]), x1_v512)
    x2_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1+16:16]), x2_v512)
endo
// perform vector add on two vector x1_v512 and x2_v512
x1_v512 = _mm512_add_ps(x1_v512, x2_v512);

// perform horizontal add on all elements of x1_v512, and
// the add x for using its value in the remainder loop
x = x + _mm512_hadd_ps(x1_v512)
do k2 = mainTripCount, n // Remainder loop
    x = x + fsqrt(y[k2])
endo
Less-Than-Full-Vector Vectorization

misalign = &y[0] & 63
peeledTripCount = (63 - misalign) / sizeof(float)
x = 10.0f;
// create a vector: <0,1,2,…15>
k0_v512 = _mm512_series_pi(0, 1, 16)

// create vector: all 16 elements are peeledTripCount
peeledTripCount_v512 =
    _mm512_broadcast_pi32(peeledTripCount)
x1_v512 = (m512)0
x2_v512 = (m512)0

// create vector: all 16 elements has the same value n
n_v512 = _mm512_broadcast_pi32(n)
step_v512 = _mm512_broadcast_pi32(16)

mainTripcount = n - ((n - peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
    x1_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1:16]), x1_v512)
x2_v512 = _mm512_add_ps( _mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo

// create a vector: <mainTripCount, mainTripCount+1 ... mainTripCount+15>
k2_v512 = _mm512_series_pi(mainTripCount, 1, 16)

// perform horizontal add on 8 elements and final reduction sum to write
// the result back to x.
x = x + _mm512_hadd_ps(x1_v512)
Alignment Optimizations for Intel® MIC Architecture

MIC alignment requirements

✓ Simple load/store instructions require the alignment to be known at compile time (64-byte aligned)

✓ SIMD load/store instructions including gather/scatter require at least element size alignment. Misaligned elements will cause a fault.

✓ No special unaligned load/store instructions, the compiler uses unpacking loads and packing stores which are capable of dealing with unaligned (element-aligned) memory locations

✓ The faulting nature of masked memory access instructions adds extra complexity to those instructions addressing data outside paged memory, and may fail even if actual data access is masked out. The exceptions are gather/scatter instructions.

Alignment schemes for MIC

✓ Aggressive data alignment optimizations with alignment peeling and multi-versioning.

✓ For unmasked unaligned (element-aligned) vector loads/stores, the compiler uses unpacking/packing load and store instructions (safe and better than using gather/scatter).

✓ For unaligned masked and/or converting loads/stores, the compiler uses gather/scatter instructions instead for safety, even though this degrades performance.

✓ With –opt-assume-safe-padding knob, unaligned masked and/or converting load/store operations are emitted as unpacking loads/packing stores.
An Alignment Example

```c
void foo(float *x, float *y, float *z, int n) {
    #pragma omp simd
    for (int k=0; k<n; k++) {
        x[k] = x[k] * (y[k] + z[k]);
    }
}
```

Assume array x, y, and z were allocated using malloc such as:
```
x = (float *)malloc(sizeof(float) * n);
```
then they should be changed by the user to say:
```
x = (float *)malloc(sizeof(float) * n + 64);
```

Vectorized alignment peeling loop body with
–opt-assume-safe-padding

```asm
... ...
..B2.8:     # Preds ..B2.7 Latency 53
    vmovaps %zmm1, %zmm4   #8.13 c1
    vmovaps %zmm1, %zmm5   #8.20 c5
    vloadunpacklps (%rsi,%r10,4), %zmm4(%k1) #8.13 c9
    vloadunpacklps (%rdx,%r10,4), %zmm5(%k1) #8.20 c17
    vloadunpacklps (%r10,%r10,4), %zmm4(%k1) #8.13 c13
    vloadunpacklps (%r10,%r10,4), %zmm5(%k1) #8.20 c21
    vloadunpacklps 64(%rsi,%r10,4), %zmm4(%k1) #8.13 c25
    vloadunpacklps 64(%rdx,%r10,4), %zmm5(%k1) #8.20 c29
    vloadunpacklps 64(%r10,%r10,4), %zmm4(%k1) #8.13 c33
    vaddps %zmm5, %zmm4, %zmm7 #8.20 c37
    vmulps %zmm7, %zmm6, %zmm8 #8.5 c45
    vpackstorelps %zmm8, (%rdx,%r10,4){%k1} #8.5 c49
    vpackstorehps %zmm8, 64(%rdx,%r10,4){%k1} #8.5 c53
    movb %al, %al #8.5 c53
...
```

Vectorized alignment peeling loop body without
–opt-assume-safe-padding

```asm
... ...
..B2.8:     # Preds ..B2.7 Latency 57
    vmovaps .L_210floatpacket.10(%rip), %zmm8 #8.5 c1
    lea (%rdi,%r13), %r14
    vmovaps %zmm1, %zmm5
    kmov %k4, %k2
    vgatherdps (%r14,%zmm8,4), %zmm4(%k2) #8.13
    jkd ...L14, %k2 # Prob 50%
    vgatherdps (%r14,%zmm8,4), %zmm4(%k2) #8.13
    jknzd ...L15, %k2 # Prob 50%
    vmovaps %zmm1, %zmm6
    kmov %k4, %k3
    lea (%rdx,%r13), %r14
    lea (%rdi,%r13), %r12
    vgatherdps (%r14,%zmm8,4), %zmm5(%k3) #8.20
    jkd ...L16, %k3 # Prob 50%
    vgatherdps (%r14,%zmm8,4), %zmm5(%k3) #8.20
    jknzd ...L17, %k3 # Prob 50%
    vaddps %zmm5, %zmm4, %zmm7 #8.20 c37
    kmov %k4, %k1
    vgatherdps (%r12,%zmm8,4), %zmm6(%k1) #8.5
    jkd ...L18, %k1 # Prob 50%
    vgatherdps (%r12,%zmm8,4), %zmm6(%k1) #8.5
    jknzd ...L19, %k1 # Prob 50%
    vmulps %zmm7, %zmm6, %zmm9 #8.5 c49
    nop #8.5 c53
    vscatteredps %zmm9, (%rdx,%zmm8,4){%k4} #8.5
    jkd ...L20, %k4 # Prob 50%
    vscatteredps %zmm9, (%r12,%zmm8,4){%k4} #8.5
    jknzd ...L21, %k4 # Prob 50%
...
```
Small Matrix 2-D Vectorization

\[
\begin{align*}
\text{do } x &= 1, 4 \\
\quad &\text{do } y = 1, 4 \\
\qquad &\text{do } z = 1, 4 \\
\quad &\quad C(x, y) = C(x, y) + A(x, z) \times B(z, y) \\
\text{enddo} \\
\text{enddo} \\
\text{endo}
\end{align*}
\]

Note: Fortran is column-major.

\[
\begin{align*}
\text{zmm0} &= \langle A[4,4], \ldots, A[3,1], A[2,1], A[1,1] \rangle \\
\text{zmm1} &= \langle B[4,4], \ldots, B[3,1], B[2,1], B[1,1] \rangle \\
&\ldots
\end{align*}
\]

Vectorizing x-loop and y-loop to achieve 16-way SIMD parallelism
Small Matrix 2-D Vectorization

\[
\text{do } z = 1, 4 \\
\text{do } y = 1, 4 \\
\text{do } x = 1, 4 \\
C(x, y) = C(x, y) + A(x, z) \times B(z, y) \\
\text{enddo} \\
\text{enddo} \\
\text{enddo}
\]

To achieve 16-way SIMD parallelism for float-point MUL and ADD, need to transpose matrix B.
Small Matrix 2-D Vectorization

Intel® Xeon Phi™ SIMD Pseudo code after 2-D vectorization

\[ A_{v512} = A \]
\[ B_{v512} = B \]

// Transpose matrix B
\[ B'_{v512} = \_mm512_mask_shuf128x32(B'_{v512}, 0x8421, B_{v512}, \_MM_PERM_DCBA, \_MM_PERM_DCBA) \]
\[ B'_{v512} = \_mm512_mask_shuf128x32(B'_{v512}, 0x4218, B_{v512}, \_MM_PERM_CBAD, \_MM_PERM_ADCB) \]
\[ B'_{v512} = \_mm512_mask_shuf128x32(B'_{v512}, 0x2184, B_{v512}, \_MM_PERM_BADC, \_MM_PERM_BADC) \]
\[ B'_{v512} = \_mm512_mask_shuf128x32(B'_{v512}, 0x1842, B_{v512}, \_MM_PERM_ADCB, \_MM_PERM_CBAD) \]

// Load the first column of A_{v512} and broadcast that column to each of the remaining three columns
\[ t1_{v512} = \_mm512_swizzle_ps(A_{v512}, \_MM_SWIZ_REG_AAAA) \]

// Load the second column of A_{v512} and broadcast that column to each of the remaining three columns
\[ t2_{v512} = \_mm512_extload_ps(B'_{v512}[0:4], \_MM_FULLUPC_NONE, \_MM_BROADCAST_4X16, 0) \]
\[ C_{v512} = \_mm512_mul_ps(t1_{v512}, t2_{v512}) \] // z = 1

// Load the second column of B'_{v512} and broadcast that column to each of the remaining three columns
\[ t2_{v512} = \_mm512_extload_ps(B'_{v512}[4:4], \_MM_FULLUPC_NONE, \_MM_BROADCAST_4X16, 0) \]

// Add the existing values of C_{v512} with the product of t1_{v512} and t2_{v512} and store result in C_{v512}
\[ C_{v512} = \_mm512_madd213_ps(t1_{v512}, t2_{v512}, C_{v512}) \] // z = 2

Scalar 4x4 matrix multiply computation (single precision case)

128 memory loads, 64 multiplies, 64 additions, 16 memory stores.

Small matrix 2-D vectorization

2 SIMD loads from memory, 4 shuffles, 4 swizzles, 4 extloads, 4 multiplies, 3 additions, 1 SIMD store to memory

~14x reduction in number of operations.
## Performance Studies

<table>
<thead>
<tr>
<th>System Parameters</th>
<th>Intel® Xeon Phi™ Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chips</td>
<td>1</td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>61 and 244</td>
</tr>
<tr>
<td>Frequency</td>
<td>1 GHz</td>
</tr>
<tr>
<td>Data caches</td>
<td>32 KB L1, 512 KB L2 per core</td>
</tr>
<tr>
<td>Power Budget</td>
<td>300 W</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>7936 MB</td>
</tr>
<tr>
<td>Memory Technology</td>
<td>GDDR5</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>2.75 (GHz) (5.5 GT/s)</td>
</tr>
<tr>
<td>Memory Channels</td>
<td>16</td>
</tr>
<tr>
<td>Memory Data Width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Peak Memory Bandwidth</td>
<td>352 GB/s</td>
</tr>
<tr>
<td>SIMD vector length</td>
<td>512 bits</td>
</tr>
</tbody>
</table>
Performance Results on Xeon Phi™

Normalized SIMD Vectorization Speedup

<table>
<thead>
<tr>
<th></th>
<th>No Vectorization</th>
<th>Vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D 5x5 Convolution</td>
<td>1.00</td>
<td>10.39x</td>
</tr>
<tr>
<td>Back Projection</td>
<td>1.00</td>
<td>4.27x</td>
</tr>
<tr>
<td>N-body</td>
<td>1.00</td>
<td>10.52x</td>
</tr>
<tr>
<td>Radar (1D Convolution)</td>
<td>1.00</td>
<td>12.45x</td>
</tr>
<tr>
<td>Tree Search</td>
<td>1.00</td>
<td>2.25x</td>
</tr>
</tbody>
</table>

Normalized SIMD Vectorization Speedup

<table>
<thead>
<tr>
<th></th>
<th>No Vectorization</th>
<th>Main Vector + Peeling/Remainder scalar loop</th>
<th>Main Vector + Peeling/Remainder vector loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>intAdd short trip-count loop</td>
<td>1.00</td>
<td>2.89x</td>
<td>6.57x</td>
</tr>
<tr>
<td>floatAdd short trip-count loop</td>
<td>1.00</td>
<td>3.32x</td>
<td>7.68x</td>
</tr>
<tr>
<td>doubleAdd short trip-count loop</td>
<td>1.00</td>
<td>3.06x</td>
<td>3.28x</td>
</tr>
</tbody>
</table>

Normalized Gain from Data Alignment

<table>
<thead>
<tr>
<th></th>
<th>unaligned memory accesses</th>
<th>aligned memory accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>intAdd short trip-count loop</td>
<td>1.00</td>
<td>1.45x</td>
</tr>
<tr>
<td>floatAdd short trip-count loop</td>
<td>1.00</td>
<td>1.41x</td>
</tr>
<tr>
<td>doubleAdd short trip-count loop</td>
<td>1.00</td>
<td>1.32x</td>
</tr>
</tbody>
</table>

Normalized SIMD Vectorization Gain/Loss

<table>
<thead>
<tr>
<th></th>
<th>No Vectorization</th>
<th>Conventional Vectorization</th>
<th>Small Matrix 2-D Vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision Single Matrix Multiply</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Single Precision Paired Matrix Multiply</td>
<td>1.00</td>
<td>1.06</td>
<td>1.04x</td>
</tr>
<tr>
<td>Double Precision Single Matrix Multiply</td>
<td>1.00</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Double Precision Paired Matrix Multiply</td>
<td>1.00</td>
<td>0.93</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Xeon Phi™ Suitability -- Keys for High Performance

• Advice for successful programming on MIC:
  • Program with lots of threads that use vectors with your preferred programming languages and parallelism models

• Any restructuring of code to take advantage of parallelism generally carries over to Xeon as well

• Application should be able to:
  • Take advantage of all threads (~240) on MIC and exhibit good scaling
  • Make efficient use of all vector resources
    • 16X single-precision, 8X double
  • Make efficient use of memory bandwidth through judicious use of caches
    • 512 KB L2 cache, 32KB L1 cache per core
Look Forward: what are we up to?
Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (#pragma vector, ...)

Explicit Vector Programming
(OpenMP* 4.0 SIMD, Cilk Plus SIMD, Array Notation)

SIMD intrinsic class
(e.g.: F32vec, F64vec, ...)

Vector intrinsic
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

Assembler code
(e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
function mandelx4(c_re4, c_im4, max_iterations) {
    var z_re4  = c_re4;
    var z_im4  = c_im4;
    var four4  = float32x4.splat (4.0);
    var two4   = float32x4.splat (2.0);
    var count4 = int32x4.splat (0);
    var one4   = int32x4.splat (1);

    for (var i = 0; i < max_iterations; ++i) {
        var z_re24 = SIMD.float32x4.mul (z_re4, z_re4);
        var z_im24 = SIMD.float32x4.mul (z_im4, z_im4);

        var mi4    = SIMD.float32x4.lessThanOrEqual (SIMD.float32x4.add (z_re24, z_im24), four4);
        // if all 4 values are greater than 4.0, there’s no reason to continue
        if (mi4.signMask === 0x00) {
            break;
        }

        var new_re4 = SIMD.float32x4.sub(z_re24, z_im24);
        var new_im4 = SIMD.float32x4.mul(SIMD.float32x4.mul (two4, z_re4), z_im4);
        z_re4       = SIMD.float32x4.add(c_re4, new_re4);
        z_im4       = SIMD.float32x4.add(c_im4, new_im4);
        count4      = SIMD.int32x4.add(count4, SIMD.int32x4.and (mi4, one4));
    }
    return count4;
}
## SIMD Application Areas

<table>
<thead>
<tr>
<th>Science</th>
<th>Media</th>
<th>Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equation solving</td>
<td>Video processing</td>
<td>Games</td>
</tr>
<tr>
<td>Giga-FLOP/sec</td>
<td>Giga-bits/sec</td>
<td>Giga-Triangles/sec</td>
</tr>
<tr>
<td>Vectors</td>
<td>SIMD</td>
<td>Vertex processor</td>
</tr>
<tr>
<td>Fortran auto-vectorization</td>
<td>Assembly language, C with Intrinsics</td>
<td>Shaders</td>
</tr>
</tbody>
</table>

Historically, different architectures and programming models
Hey Buddy! I googled this piece of code, added #pragma omp simd ..., compiled with Intel® Compiler and now it works well and gets 8x speedup with SIMD! Our end of year bonus will be very nice, so what are you doing tonight?

Hey Buddy, I had this formula to prove my theory, my student implemented it with C++ and #pragma omp simd..., compiled with Intel® Compiler. Now it works and get ~8x speedup with SIMD, But, I was expecting ~15x speedup! Yuck@. Tell me what I need to do to get 15x?

- Programmers want to just express their intent
- Programmers need to control details of execution in order to achieve peak performance
- «LOOP WAS VECTORIZED» is only the beginning!
// Sample.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "library.h"

#define MAX_COUNT 1000

int main(int argc, char**) {
    void generate(int *data, int upperbound) {
        for (int i = 0; i < upperbound; ++i) { ... 
    }

    void print(int *data, int upperbound) {
        ... 
    }

    int a[10];
    ... 

    void foo1() {
        cout << data[1] << " 
        ... 
    }

    int n[MAX_COUNT];
    int b[MAX_COUNT];
    ... 

generate(a, MAX_COUNT);
Recipe: Close to Medal SIMD Performance

- All calculations in the registers
  - Good sign: “vpaddw xmm8,xmm6,xmm8”

- Narrowest possible data type
  - Good sign: “vpaddb xmm8,xmm12,xmm5”.

- Effective pack and saturate
  - Good sign: “vpackuswb xmm12,xmm12,xmm8”

- Use most efficient AVX2 cross lane operations
  - Good sign: “vpermq ymm2,ymm7,0D8h”
Looking forward more to Coming

- Programmer-defined vector variant functions
  ```cpp
  __declspec(vector_variant(implements(MyMax<float>)))  // function name MyMax needs
  MS128 MyMaxVec(MS128 v_a, MS128 vec_b) {              // to be specialized for “float” type
    MS128 tmp = _mm_max_ps(vec_a, vec_b);
    return tmp;
  }
  ```
- Language extensions to mix scalar and vector code
  ```cpp
  #pragma omp ordered [simd]
  ```
- Virtual function and function pointer in SIMD context
- Change data layout of aggregate structures
  - From AoS to SoA
- vcompress/vexpand, vconflict: cross iteration communication
- New SIMD Hardware support
Vision: Explicit SIMD Programming

The reality:

- There is **no one single solution** that would make all programmers happy after decades of trying.
- There is **no free lunch** for effectively utilizing SIMD HW in multicore CPUs, accelerators and GPUs.
- There are **many emerging programming models** for multicore CPUs, accelerators and GPUs.
- Programming languages and compilers are driven by hardware and application
- The incremental approach of applying the learnings from HPC and graphics is working

Simple programming language extensions for computational use of SIMD Hardware

Portable and consistent SIMD programming model across CPU, Coprocessors and GPUs
Legal Disclaimer & Optimization Notice

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

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

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

Optimization Notice

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
References

- “Compiling C++ SIMD Extensions for Function and Loop Vectorization” by Xinmin Tian, et.al, at IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2012.
- “Practical SIMD Vectorization Techniques for Intel® Xeon Phi Coprocessors” by Xinmin Tian, et.al. At IEEE 27th International IPDPS Workshops, 2013.
- “Performance Essentials with OpenMP* 4.0 Vectorization” by Bob Chesebrough, Intel Corporation.
- “Vectorization/Parallelization” in the Intel Compiler by Peng Tu, Intel Corporation.