

# 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

# 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

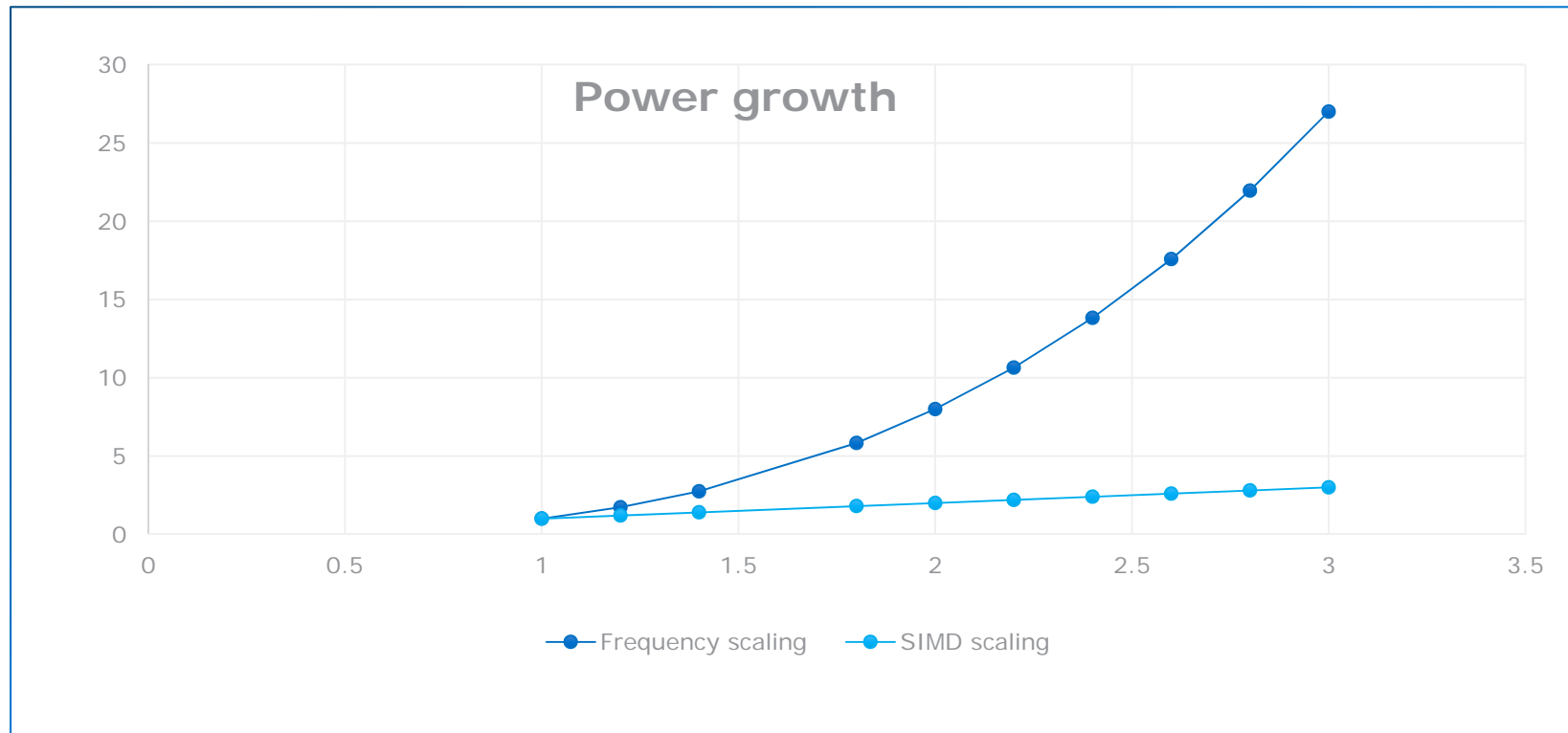
Xeon Phi™ Suitability -- Keys for High Performance

Looking Forward: What are we up to?

Recipe and Vision: Close to Medal Performance



# Why SIMD? SIMD Scaling vs. Frequency Scaling

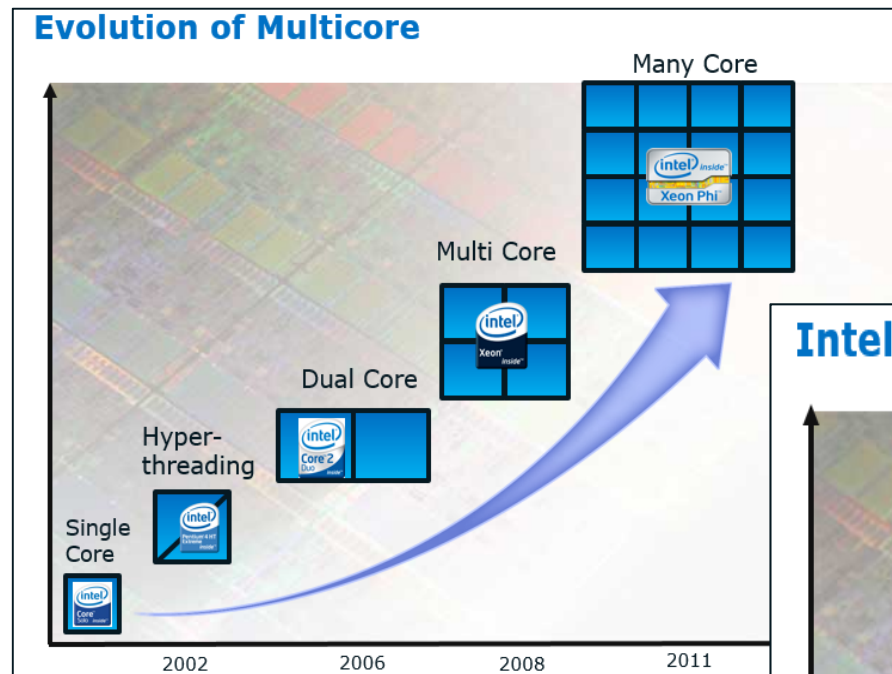


- ✓ Wider SIMD – Linear increase in area and power
- ✓ Wider superscalar – Quadratic increase in area and power
- ✓ Higher frequency – Cubic increase in power

With SIMD we can go faster with less power!

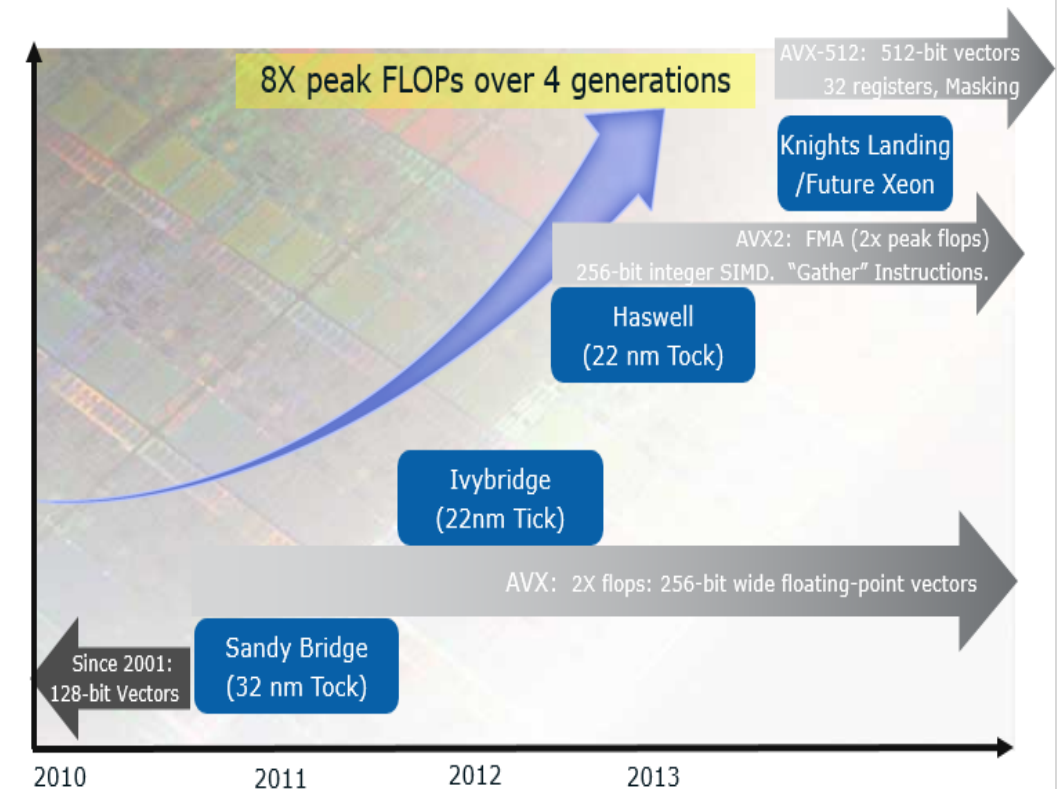


# Evolution of MultiCore and Intel® AVX Extensions



**All future processors will have multiple cores with SIMD instructions**

## Intel® Advanced Vector Extensions



**LCPC'2014**



Look back three and half  
decades!



# Look back: Compiler Vectorization in 1978

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

CACM 1978

```
C
C*****
C***  KERNEL 1    HYDRO FRAGMENT
C*****
C
cdir$ 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

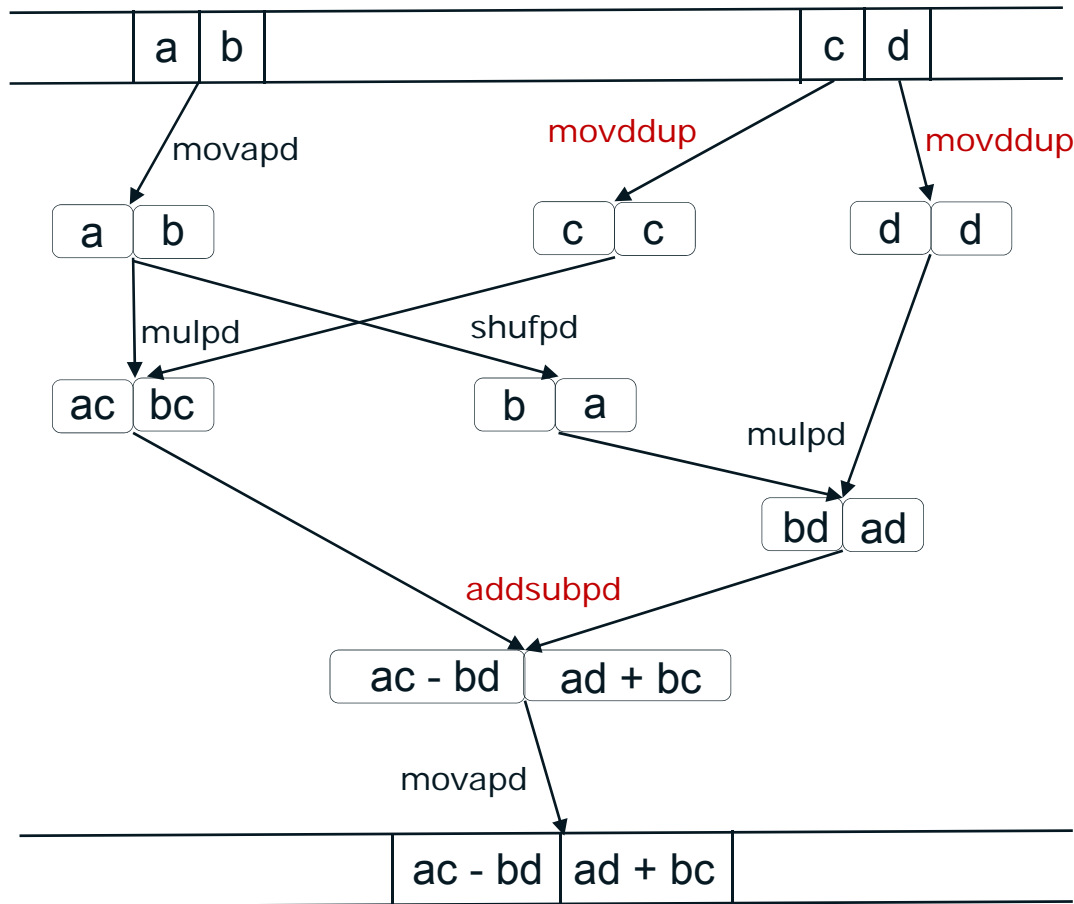
Compiler vectorization "solved" in 1978



LCPC'2014

# 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**

Memory

Performance can be improved up to ~75%, SPEC2000FP/168.wupwise 10-15%





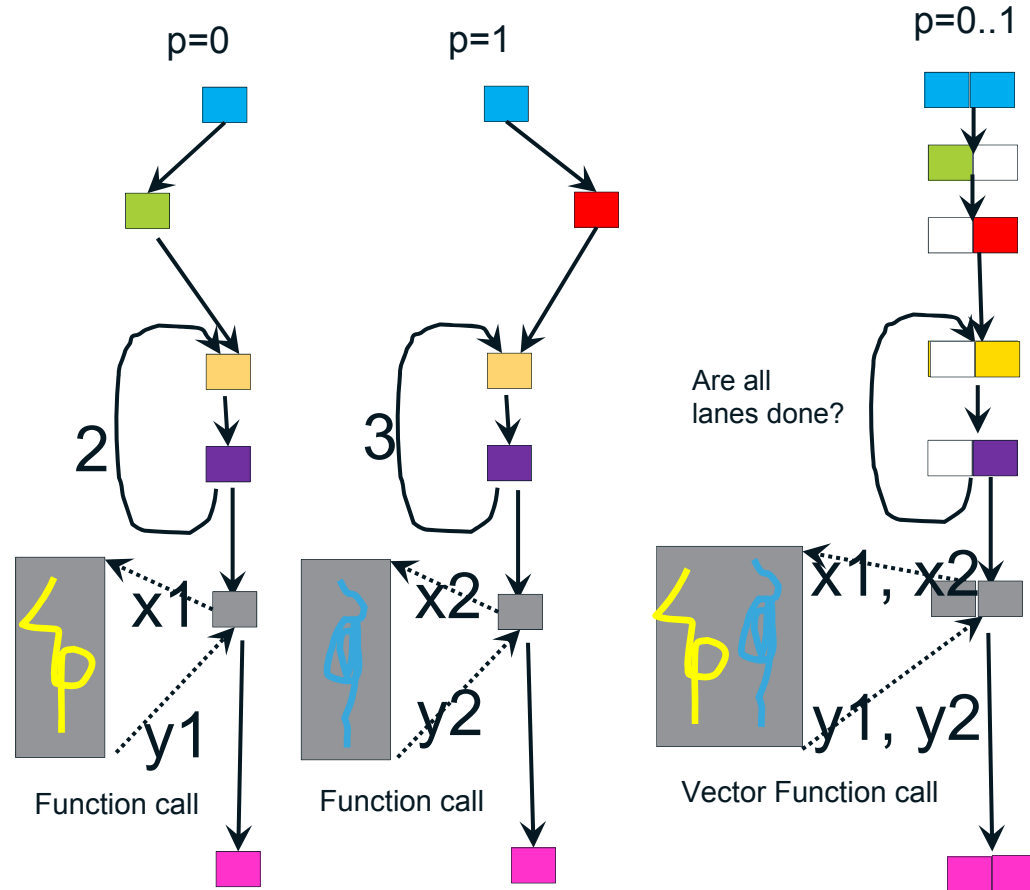
Vanquishing SIMD Hurdles !



# 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

Compiler created

```
__m128 vfoo(__m128 vx)
```

```
{....
```

```
}
```

Vector C function

sfoo(x0)->r0

sfoo(x1)->r1

sfoo(x2)->r2

sfoo(x3)->r3

sfoo(x4)->r4

... ..

Scalar execution

sfoo(x0)->r0

sfoo(x1)->r1

sfoo(x2)->r2

sfoo(x3)->r3

sfoo(x4)->r4

sfoo(x5)->r5

sfoo(x6)->r6

sfoo(x7)->r7

sfoo(x8)->r8

sfoo(x9)->r9

... ..

... ..

... ..

vfoo(x0...x3)->r0...r3

vfoo(X4...X7)->r4...r7

... ..

Vector execution



LCPC'2014

# 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);
```

```
#pragama 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 dxctau1 = del_p_y * tz[j] - ty[j] * del_p_z;  
        const float dyctau1 = del_p_z * tx[j] - tz[j] * del_p_x;  
        const float dzctau1 = 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      -= dxctau1 * dst3;  
        *vely      -= dyctau1 * dst3;  
        *velz      -= dzctau1 * dst3;  
    }  
}
```

KNC performance improvement  
over 2X going  
from inner to outer-loop vectorization



LCPC'2014

# Recursive Function Vectorization

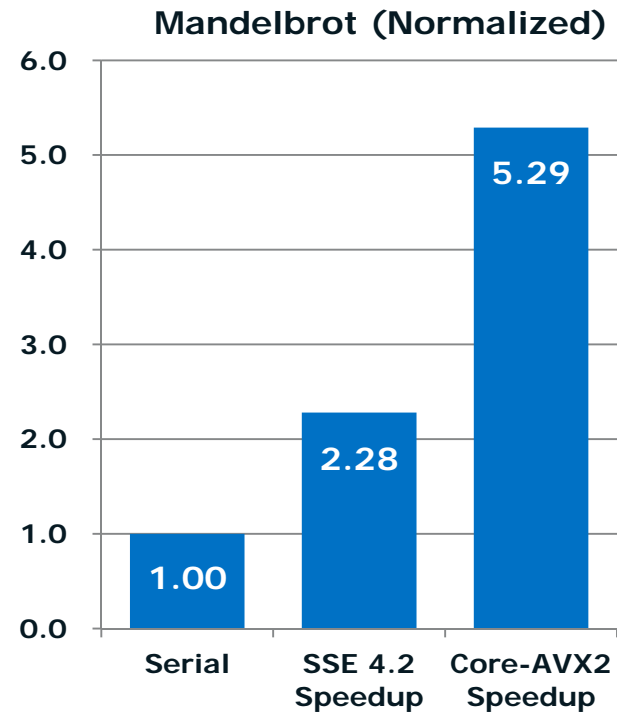
```
#pragma omp declare simd
int binsearch(int key, int lo, int hi) {
    int ans;
    if ( lo > hi) {
        ans = -1;
    } else {
        int mid = lo + ((hi - lo) >> 1);
        int t = sortedarr[mid];
        if (key == t) {
            ans = mid;
        } else if ( key > t) {
            ans = 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);
};
```



# C++ Explicit Vectorization via OpenMP 4.0 SIMD

```
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?

```
#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;
}
```

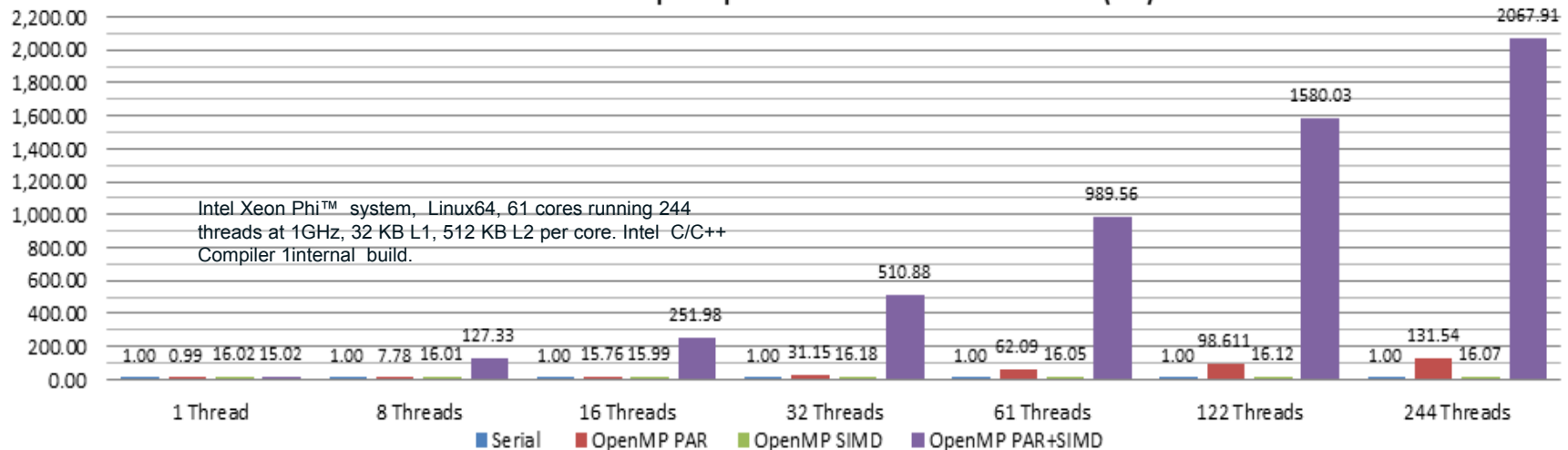
```
#pragma omp parallel for schedule(guided)
```

```
for (int32_t y = 0; y < ImageHeight; ++y) {
    float c_im = max_imag - y * imag_factor;
```

```
#pragma omp simd safelen(32)
```

```
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);
}
}
```

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





# Outer Loop Vectorization

## SPEC CPU2006 410.bwaves

```
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,j,kp1)
      +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,j,km1)
  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

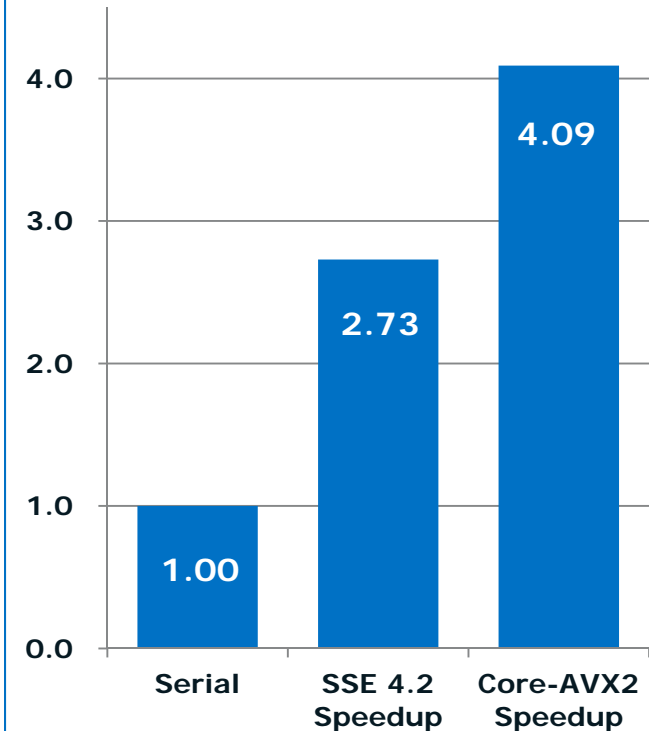
```
#pragma omp declare simd linear(z:40) uniform(L, N, Nmat) linear(k)
float path_calc(float *z, float L[][VLEN], int k, int N, int Nmat)

#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] = LO[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)

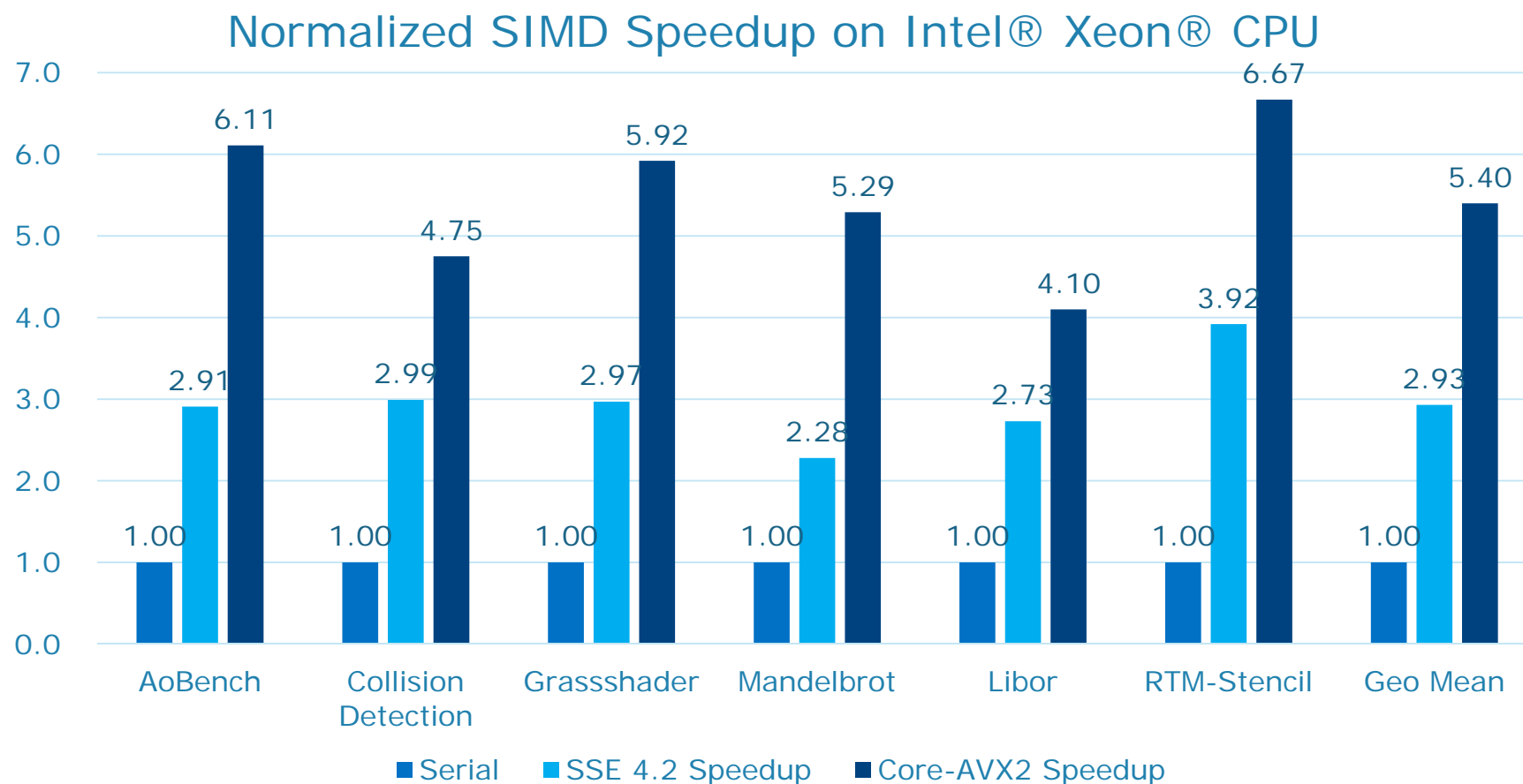


Configuration: Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz system (4 cores with Hyper-Threading On), running at 3.50GHz, with 32.0GB RAM, L1 Cache 256KB, L2 Cache 1.0MB, L3 Cache 8.0MB, 64-bit Windows® Server 2012 R2 Datacenter. Compiler options: SSE4.2: -O3 -Qipo -QxSSE4.2 or AVX2: -O3 -Qipo -QxCORE-AVX2.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



LCPC'2014

# C++ Explicit Vectorization via OpenMP 4.0 SIMD / Cilk™ Plus: SIMD Performance



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>



# Mixed Data Type Vectorization

```
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;
```

```
    }
```

```
}
```

a3	a2	a1	a0
----	----	----	----

b1	b0
----	----

a3	a2	a1	a0
----	----	----	----

b1	b0
----	----

a3	a2	a1	a0
----	----	----	----

b1	b0
----	----

b3	b2
----	----

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

# TPCF Code with Compress Idiom on Xeon Phi™

```
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



# Less-Than-Full-Vector Vectorization

```
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])
enddo
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)
enddo
// 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])
enddo
```





# 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
do k0 = 0, peeledTripCount-1, 16
    // generate mask for vectorizing peeling loop
    mask = _mm512_compare_pi32_mask_lt(k0_v512,
        peeledTriPCount_v512)
    x1_v512 = _mm512_add_ps_mask(
        _mm512_fsqrt(y[k0:16]), x1_v512, mask)
enddo
```

```
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)
```

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

```
do k2 = mainTripCount, n, 16 // vectorized remainder loop
    mask = _mm512_compare_pi32_mask_lt(k2_v512, n_v512)
    x1_v512 = _mm512_add_ps_mask(
        _mm512_fsqrt(y[k2:16]), x1_v512, mask)
    k2_v512 = _mm512_add_ps(k2_v512, step_v512)
enddo
```

```
x1_v512 = _mm512_add_ps(x1_v512, x2_v512);
```

```
// 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

```
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

```
... ..  
..B2.8:                                # Preds ..B2.7 Latency 53  
    vmovaps    %zmm1, %zmm4            #8.13 c1  
    vmovaps    %zmm1, %zmm5            #8.20 c5  
    vmovaps    %zmm1, %zmm6            #8.5 c9  
    vloadunpacklps (%rsi,%r10,4), %zmm4{%k1} #8.13 c13  
    vloadunpacklps (%rdx,%r10,4), %zmm5{%k1} #8.20 c17  
    vloadunpacklps (%rdi,%r10,4), %zmm6{%k1} #8.5 c21  
    vloadunpackhps 64(%rsi,%r10,4), %zmm4{%k1} #8.13 c25  
    vloadunpackhps 64(%rdx,%r10,4), %zmm5{%k1} #8.20 c29  
    vloadunpackhps 64(%rdi,%r10,4), %zmm6{%k1} #8.5 c33  
    vaddps      %zmm5, %zmm4, %zmm7      #8.20 c37  
    vmulps      %zmm7, %zmm6, %zmm8      #8.5 c41  
    nop         #8.5 c45  
    vpackstorelps %zmm8, (%rdi,%r10,4){%k1} #8.5 c49  
    vpackstorehps %zmm8, 64(%rdi,%r10,4){%k1} #8.5 c53  
    movb        %al, %al                 #8.5 c53  
... ..
```

Don't need to read the ASM code details

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

```
... ..  
..B2.8:                                # Preds ..B2.7 Latency 57  
    vmovaps    .L_2il0floatpacket.10(%rip), %zmm8 #8.5 c1  
    vmovaps    %zmm1, %zmm4            #8.13 c5  
    lea        (%rsi,%r13), %r14        #8.13 c5  
    vmovaps    %zmm1, %zmm5            #8.20 c9  
    kmov       %k4, %k2                #8.13 c9  
    vgatherdps (%r14,%zmm8,4), %zmm4{%k2} #8.13  
    jkzd       ..L14, %k2               # Prob 50% #8.13  
    vgatherdps (%r14,%zmm8,4), %zmm4{%k2} #8.13  
    jknzd      ..L15, %k2               # Prob 50% #8.13  
    vmovaps    %zmm1, %zmm6            #8.5 c21  
    kmov       %k4, %k3                #8.20 c21  
    lea        (%rdx,%r13), %r14        #8.20 c25  
    lea        (%rdi,%r13), %r12        #8.5 c25  
    vgatherdps (%r14,%zmm8,4), %zmm5{%k3} #8.20  
    jkzd       ..L16, %k3               # Prob 50% #8.20  
    vgatherdps (%r14,%zmm8,4), %zmm5{%k3} #8.20  
    jknzd      ..L17, %k3               # Prob 50% #8.20  
    vaddps     %zmm5, %zmm4, %zmm7      #8.20 c37  
    kmov       %k4, %k1                #8.5 c37  
    vgatherdps (%r12,%zmm8,4), %zmm6{%k1} #8.5  
    jkzd       ..L18, %k1               # Prob 50% #8.5  
    vgatherdps (%r12,%zmm8,4), %zmm6{%k1} #8.5  
    jknzd      ..L19, %k1               # Prob 50% #8.5  
    vmulps     %zmm7, %zmm6, %zmm9      #8.5 c49  
    nop        #8.5 c53  
    vscatterdps %zmm9, (%r12,%zmm8,4){%k4} #8.5  
    jkzd       ..L20, %k4               # Prob 50% #8.5  
    vscatterdps %zmm9, (%r12,%zmm8,4){%k4} #8.5  
    jknzd      ..L21, %k4               # Prob 50% #8.5  
... ..
```



# Small Matrix 2-D Vectorization



```
do x = 1, 4
  do y = 1, 4
    do z = 1, 4
      C(x, y) = C(x, y) + A(x, z) * B(z, y)
    enddo
  enddo
enddo
```

Note: Fortran is column-major.

zmm0 = <A[4,4], ... .., A[3,1], A[2,1], A[1,1]>

zmm1 = <B[4,4], ... .., B[3,1], B[2,1], B[1,1]>

.....

```
do z = 1, 4
```

```
  do y = 1, 4
```

```
    do x = 1, 4
```

```
      C(x, y) = C(x, y) + A(x, z) * B(z, y)
```

```
    enddo
```

```
  enddo
```

```
enddo
```



Vectorizing x-loop and y-loop  
to achieve 16-way SIMD parallelism



# Small Matrix 2-D Vectorization

```
do z = 1, 4
```

```
  do y = 1, 4
```

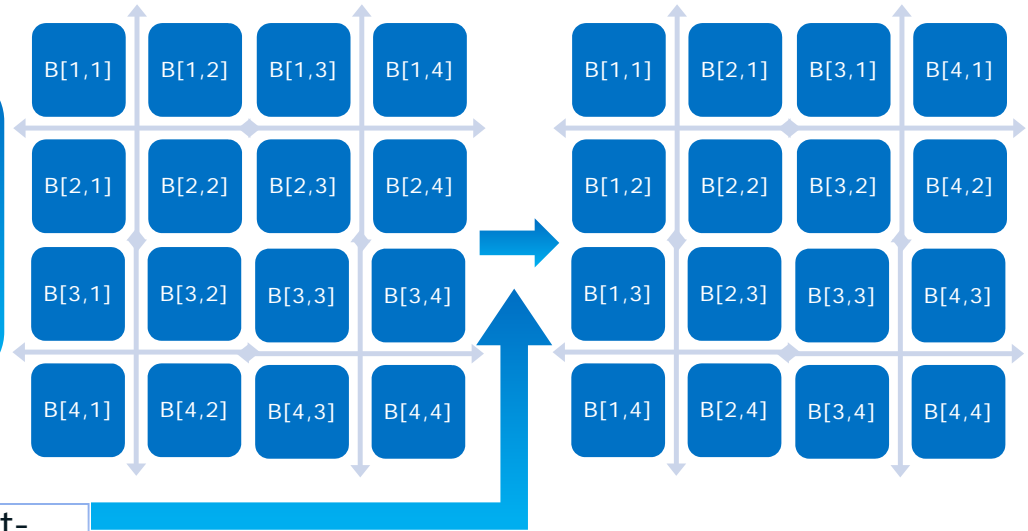
```
    do x = 1, 4
```

```
      C(x, y) = C(x, y) + A(x, z) * B(z, y)
```

```
    enddo
```

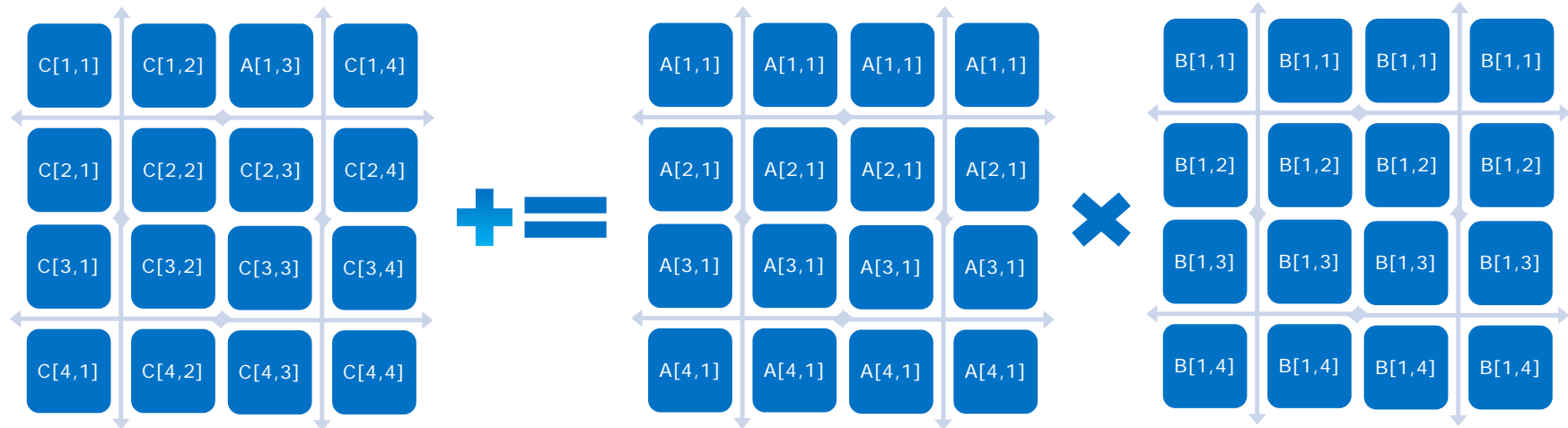
```
  enddo
```

```
enddo
```



To achieve 16-way SIMD parallelism for float-point MUL and ADD, need to transpose matrix B

z= 1, broadcast A[1:4,z] to 4 SIMD 128-bit Lanes  
broadcast B[z,1:4] to 4 SIMD 128-bit Lanes





# 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 first column of B'_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 A_v512 and broadcast that column to each of the remaining three columns
t1_v512 = _mm512_swizzle_ps(A_v512, _MM_SWIZ_REG_BBBB)
// 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.**



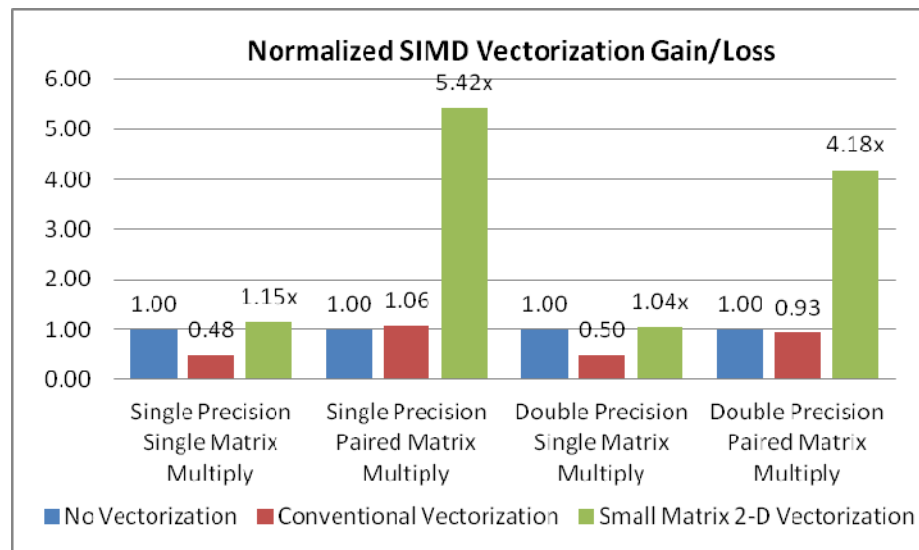
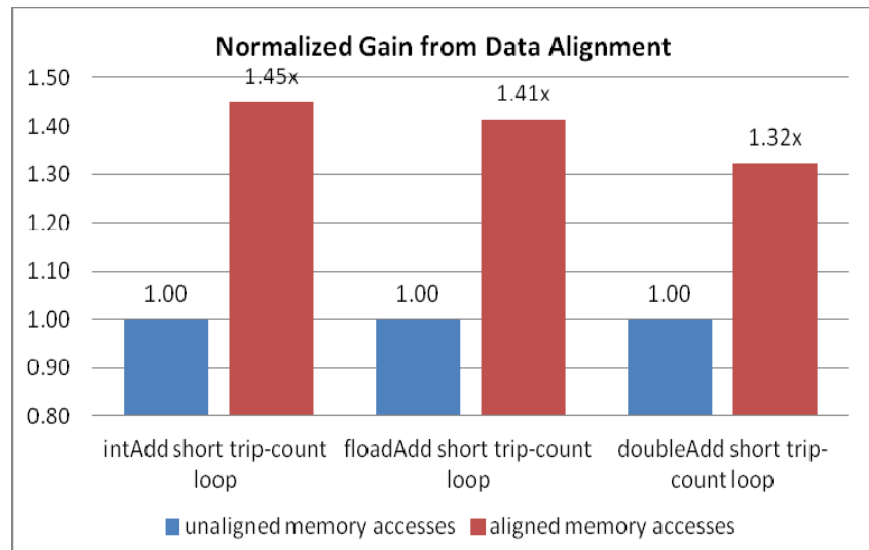
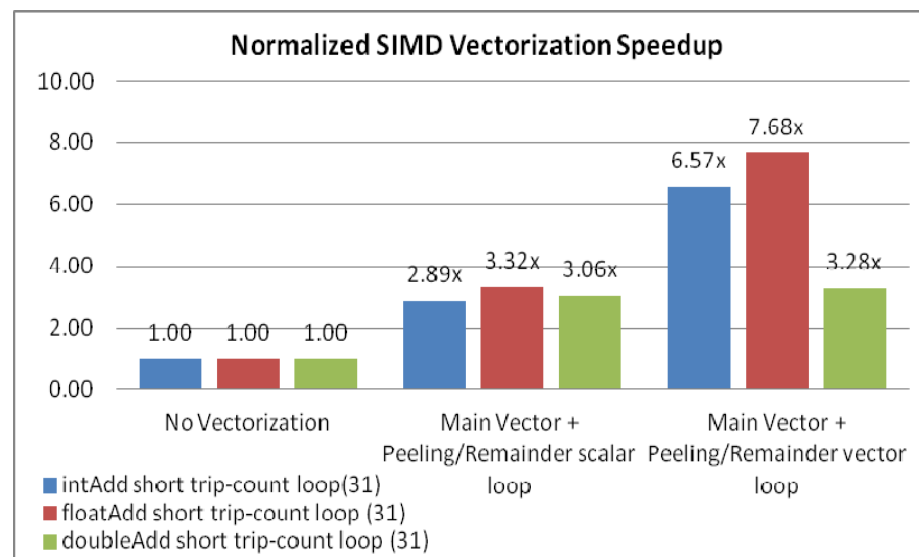
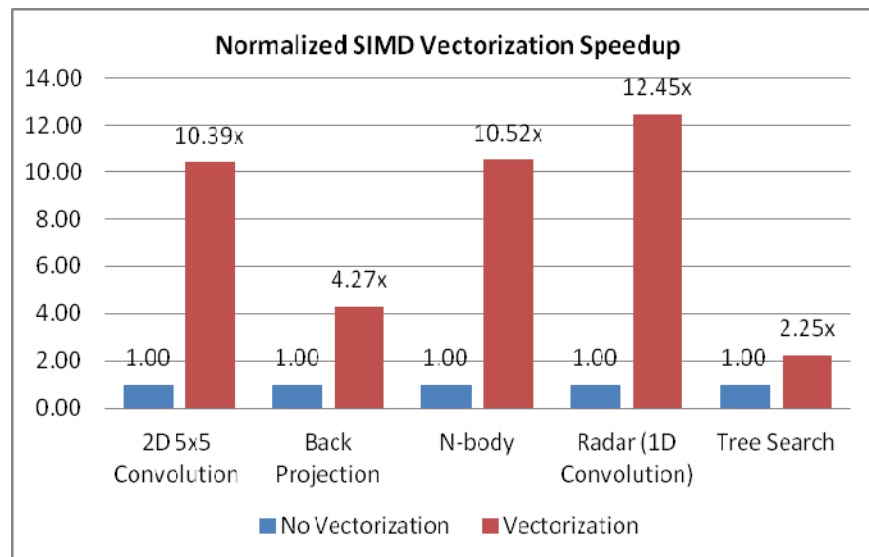


# Performance Studies

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



# Performance Results on Xeon Phi™



# 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

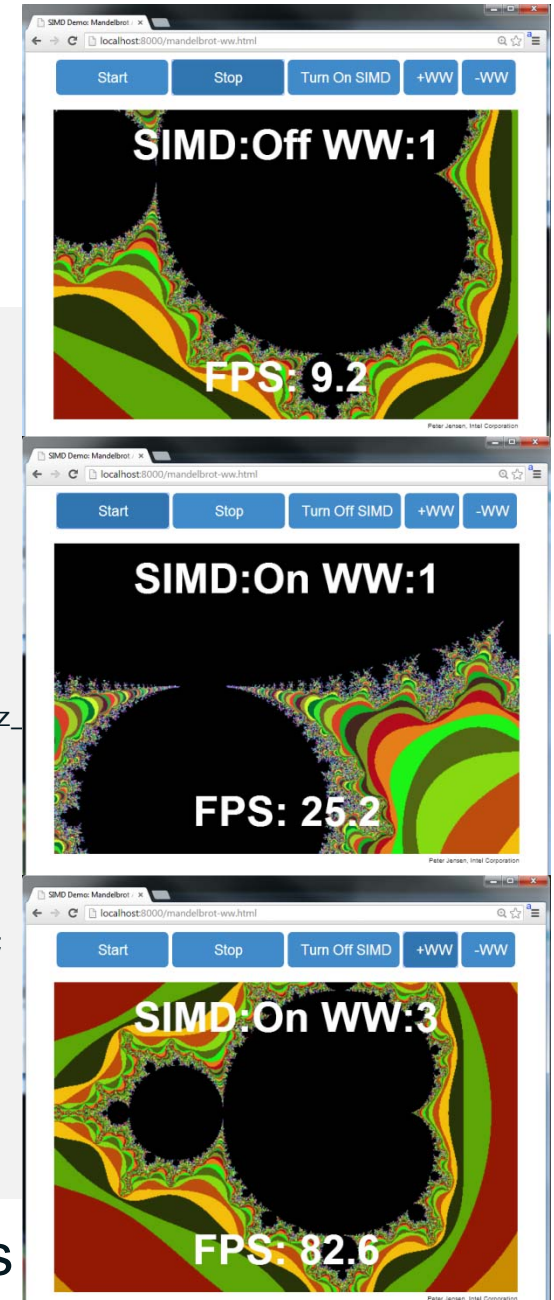


# SIMD.JS – The API

## Mandelbrot – SIMD.JS API version

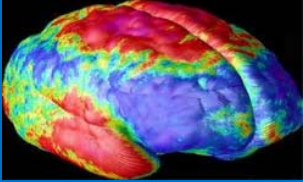


```
function mandelx4(c_re4, c_im4, max_iterations) {  
    var z_re4 = c_re4;  
    var z_im4 = c_im4;  
    var four4 = float32x4.splat(4.0);  
    var two4 = float32x4.splat(2.0);  
    var count4 = int32x4.splat(0);  
    var one4 = int32x4.splat(1);  
  
    for (var i = 0; i < max_iterations; ++i) {  
        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;  
}
```

Initial support for float32x4 and int32x4 operations





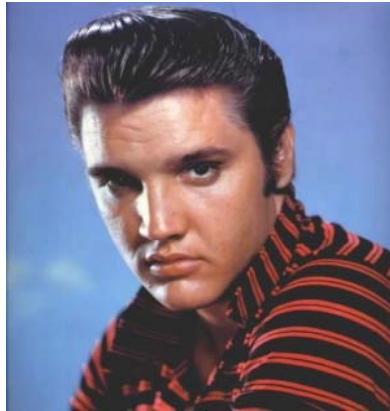
# SIMD Application Areas

		
Science	Media	Graphics
Equation solving	Video processing	Games
Giga-FLOP/sec	Giga-bits/sec	Giga-Triangles/sec
Vectors	SIMD	Vertex processor Pixel processor
Fortran auto-vectorization	Assembly language, C with Intrinsics	Shaders

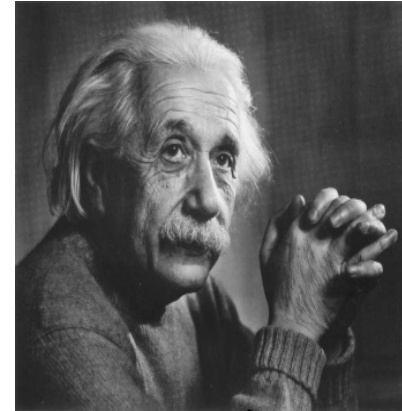
Historically, different architectures and programming models



# Not All Programmers Are Equal!



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!



# Programmer Friendly Optimization Report

Annotated source listing with compiler optimization reports

File: C:\Users\Sample\Sample.cpp

## Quick Navigation

[foo1\(\)](#)  
[foo2\(\)](#)  
[generate\(int \\*, int\)](#)  
[main\(int, TCHAR \\*\\*\)](#)  
[print\(int \\*, int\)](#)

```
1 // Sample.cpp : Defines the entry point for the console application.
2 //
3
4 #include "stdafx.h"
5 #include "library.h"
6 #define MAX_COUNT 1000
7
8 void generate(int *data, int upperbound)
9 {
10     for (int i = 0; i < upperbound; ++i)
11
12     LOOP BEGIN at C:\Users\Sample\Sample.cpp(10,2)
13     <Peeled>
14     LOOP END
15
16     LOOP BEGIN at C:\Users\Sample\Sample.cpp(10,2)
17     remark #15388: vectorization support: reference data has aligned access [ C:\Users\Sample\Sample.cpp(12,3) ]
18     remark #15300: LOOP WAS VECTORIZED
19     remark #15442: entire loop may be executed in remainder
20     remark #15449: unmasked aligned unit stride stores: 1
21     remark #15475: --- begin vector loop cost summary ---
22     remark #15476: scalar loop cost: 4
23     remark #15477: vector loop cost: 0.750
24     remark #15478: estimated potential speedup: 4.810
25     remark #15479: lightweight vector operations: 3
26     remark #15488: --- end vector loop cost summary ---
27     LOOP END
28
29     LOOP BEGIN at C:\Users\Sample\Sample.cpp(10,2)
30     <Remainder>
31     LOOP END
32
33     {
34         data[i] = i;
35     }
36 }
37
38 void print(int* data, int upperbound)
39 {
40     for (int i = 0; i < upperbound; ++i)
41
42     LOOP BEGIN at C:\Users\Sample\Sample.cpp(18,2)
43     remark #15344: loop was not vectorized: vector dependence prevents vectorization
44     remark #15382: vectorization support: call to function std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>> *, int) cannot be ve
45     remark #15382: vectorization support: call to function std::operator<<<std::char_traits<char>>(std::basic_ostream<char, std::char_traits<char>> &, const char *) cannot be vectorized [
46     LOOP END
47
48     {
49         cout << data[i] << " ";
50     }
51     cout << endl;
52 }
53
54 void foo1()
55 {
56     int a[MAX_COUNT];
57     int b[MAX_COUNT];
58
59     generate(a, MAX_COUNT);
60 }
```



# 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: “**vpckuswb** 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

```
__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

```
#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

- [“The Future of SIMD”](#) by Geoff Lowney, Keynote at 2014 Workshop on Programming models for SIMD/Vector Processing, Feb. 2014.
- [“Compiling C++ SIMD Extensions for Function and Loop Vectorization”](#) by Xinmin Tian, et.al, at IEEE 26<sup>th</sup> 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 27<sup>th</sup> International IPDPS Workshops, 2013.
- [“Performance Essentials with OpenMP\\* 4.0 Vectorization”](#) by Bob Chesebrough, Intel Corporation.
- [“Explicit Vector Programming”](#) by Milind Girkar, Intel Corporation.
- [“Vectorization/Parallelization” in the Intel Compiler](#) by Peng Tu, Intel Corporation.



