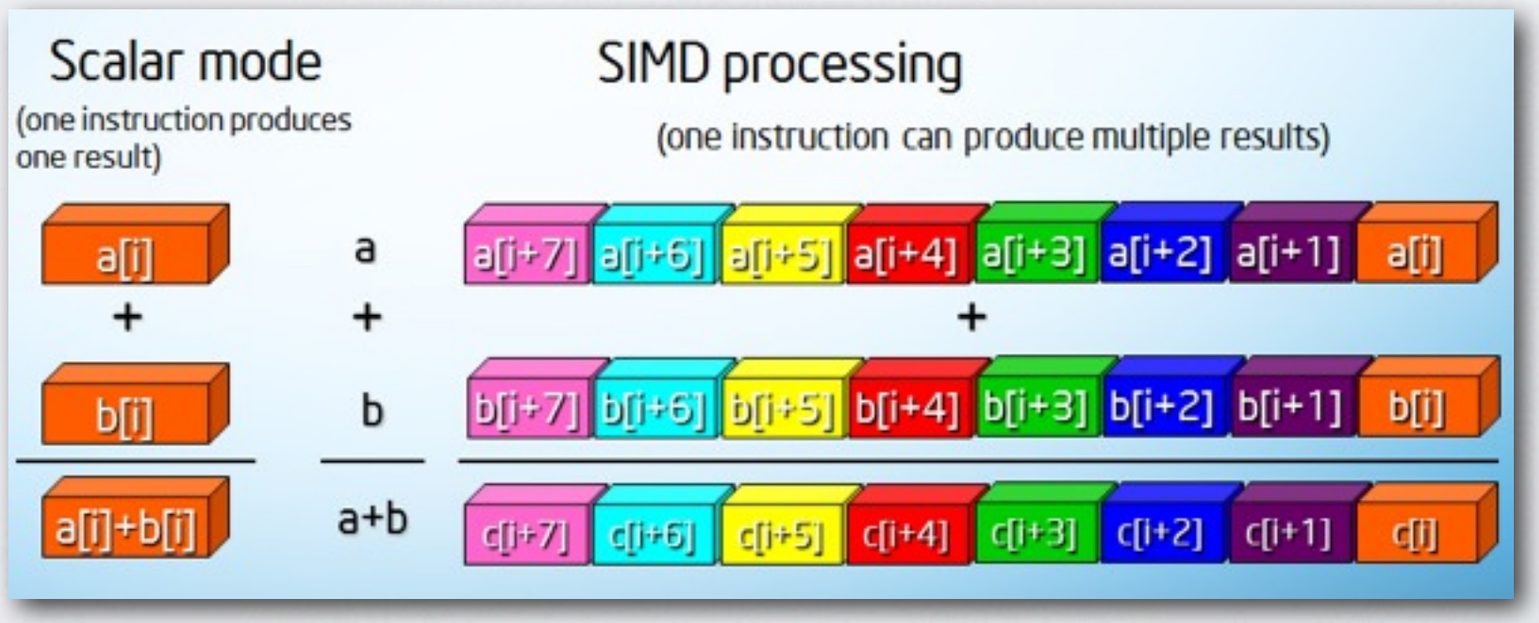
VECTORIZATION ON X86

Roberto A. Vitillo (CRD)

10/10/2013 LabTech

2.6 GHz SB-EP CPU, 8 cores 8 add + 8 mul FLOPs/cycle/core	Theoretical performance in GFLOPS	Theoretical performance / maximum
multi-threaded and vectorized	166.4	100.0%
multi-threaded and not vectorized	20.8	12.5%
serial and vectorized	20.8	12.5%
serial and not vectorized	2.6	1.6%

- **S**ingle **I**nstruction, **M**ultiple **D**ata:
  - processor throughput is increased by handling multiple data in parallel (MMX, SSE, AVX, ...)
  - vector of data is packed in one large register and handled in one operation
  - SIMD instructions are energy-efficient
  - exploiting SIMD is even more of importance for the Xeon PHI and future architectures



blogs.intel.com

- Vector sizes keep increasing but programmers continue to use simple abstraction of hw registers.

```
void multiplyMatrices(Float32x4List A, Float32x4List B, Float32x4List R) {
    var a0 = A[0];
    var a1 = A[1];
    var a2 = A[2];
    var a3 = A[3];

    var b0 = B[0];
    R[0] = b0.xxxx * a0 + b0.yyyy * a1 + b0.zzzz * a2 + b0.wwww * a3;
    var b1 = B[1];
    R[1] = b1.xxxx * a0 + b1.yyyy * a1 + b1.zzzz * a2 + b1.wwww * a3;
    var b2 = B[2];
    R[2] = b2.xxxx * a0 + b2.yyyy * a1 + b2.zzzz * a2 + b2.wwww * a3;
    var b3 = B[3];
    R[3] = b3.xxxx * a0 + b3.yyyy * a1 + b3.zzzz * a2 + b3.wwww * a3;
}
```

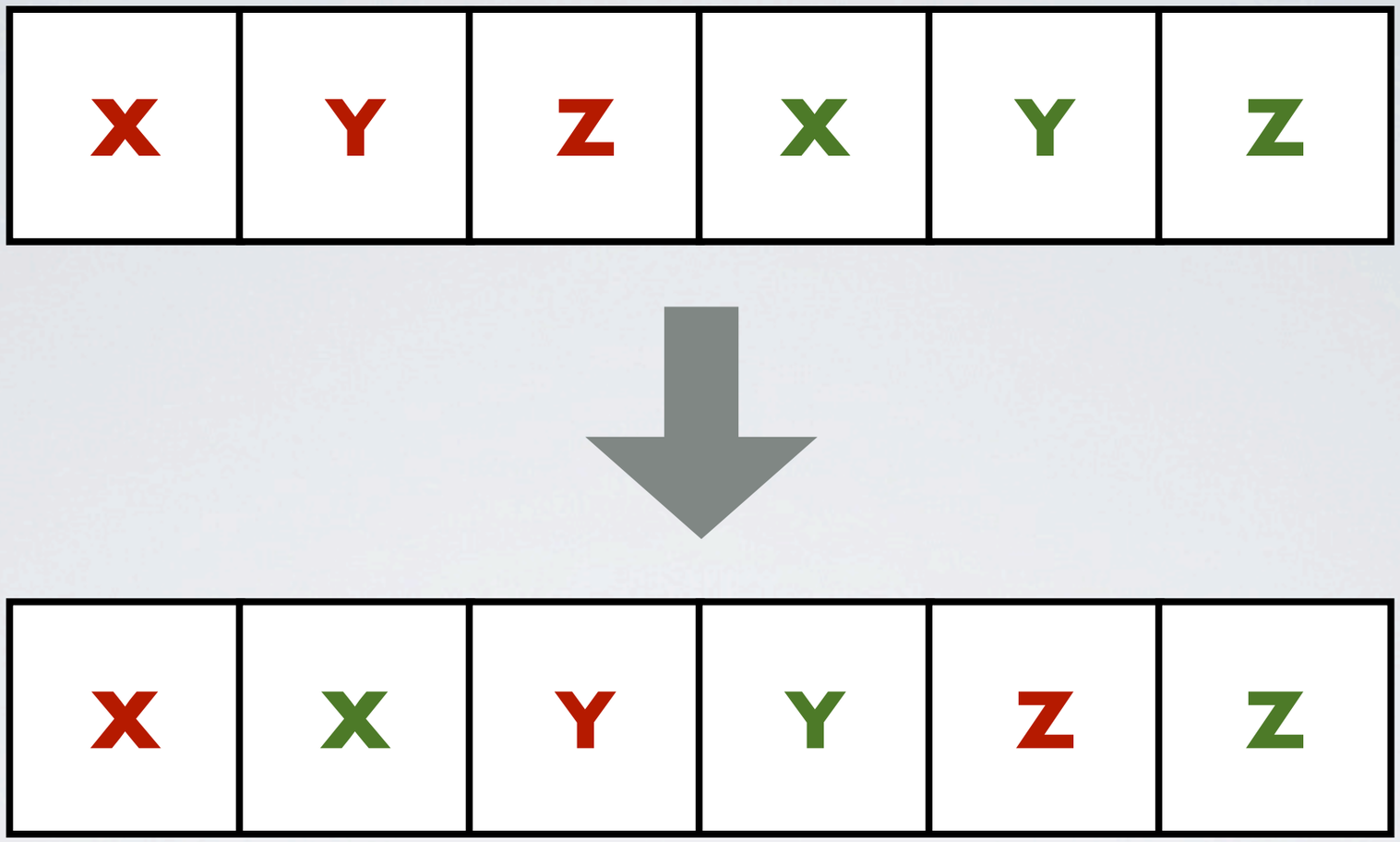
DART 2013

```
#ifdef ID_WIN_X86_SSE2_INTRIN

const __m128 mask_keep_last = __m128c( _mm_set_epi32( 0xFFFFFFFF, 0x00000000, 0x00000000, 0x00000000 ) );

for ( int i = 0; i < numJoints; i += 2, inFloats1 += 2 * 12, inFloats2 += 2 * 12, outFloats += 2 * 12 ) {
    __m128 m1a0 = _mm_load_ps( inFloats1 + 0 * 12 + 0 );
    __m128 m1b0 = _mm_load_ps( inFloats1 + 0 * 12 + 4 );
    __m128 m1c0 = _mm_load_ps( inFloats1 + 0 * 12 + 8 );
    __m128 m1a1 = _mm_load_ps( inFloats1 + 1 * 12 + 0 );
    __m128 m1b1 = _mm_load_ps( inFloats1 + 1 * 12 + 4 );
    __m128 m1c1 = _mm_load_ps( inFloats1 + 1 * 12 + 8 );

    __m128 m2a0 = _mm_load_ps( inFloats2 + 0 * 12 + 0 );
    __m128 m2b0 = _mm_load_ps( inFloats2 + 0 * 12 + 4 );
    __m128 m2c0 = _mm_load_ps( inFloats2 + 0 * 12 + 8 );
    __m128 m2a1 = _mm_load_ps( inFloats2 + 1 * 12 + 0 );
    __m128 m2b1 = _mm_load_ps( inFloats2 + 1 * 12 + 4 );
    __m128 m2c1 = _mm_load_ps( inFloats2 + 1 * 12 + 8 );

    __m128 tj0 = _mm_and_ps( m1a0, mask_keep_last );
    __m128 tk0 = _mm_and_ps( m1b0, mask_keep_last );
    __m128 tl0 = _mm_and_ps( m1c0, mask_keep_last );
    __m128 tj1 = _mm_and_ps( m1a1, mask_keep_last );
    __m128 tk1 = _mm_and_ps( m1b1, mask_keep_last );
```

DOOM 3 2005

Technology	PROS	CONS
Autovectorization	requires minimal code changes	unreliable*
Intel SIMD Directives	work well for specific use cases	might change program meaning, GCC mainline support missing
OpenCL	clear conceptual model	unreliable*, requires code restructuring
Vectorized Libraries, e.g. MKL, Eigen	programmer doesn't need to know low level details	
Cilk Plus Array Notation	mostly reliable	GCC mainline support missing
ISPC	mostly reliable, clear conceptual model	proper PHI support missing, requires code restructuring

* unless significant time is invested inspecting the generated assembly and deciphering compiler messages



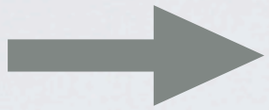
- Addressing modes
  - SSEx and AVX1 do not support strided accesses pattern and gather-scatter accesses which force the compiler to generate scalar instructions
  - Even when "fancy" access patterns are supported (e.g. IMCI and AVX2) a significant penalty is paid
  - Convert your arrays of structures to structures of arrays

cac.cornell.edu

- Memory alignment
  - Unaligned memory access may generate scalar instructions for otherwise vectorizable code
  - Vectorized loads from unaligned memory suffer from a penalty
  - Align data in memory to the width of the SIMD unit

```
void foo(double *a, double *b)
{
    for(int i = 0; i < SIZE; i++)
    {
          a[i] += b[i];
    }
}
```

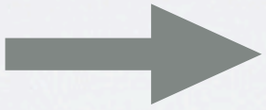
what we would like:



```
1a8:   vmovapd ymm0,YMMWORD PTR [rdi+rax*1]
1ad:   vaddpd ymm0,ymm0,YMMWORD PTR [rsi+rax*1]
1b2:   vmovapd YMMWORD PTR [rdi+rax*1],ymm0
1b7:   add    rax,0x20
1bb:   cmp    rax,0x186a0
1c1:   jne    1a8 <foo2+0x8>
```

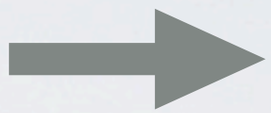
- *SIZE* is a multiple of the vector lane width
- Alignment of arrays unknown to the compiler
- GCC has to check if the arrays overlap
- If they do, scalar addition is performed
  - otherwise loop is partially vectorized

what we actually get: with autovectorization:



```
0:  lea    rax,[rsi+0x20]
4:  cmp    rdi,rax
7:  jb     4b <foo+0x4b>

9:  xor    eax,eax
b:  nop    DWORD PTR [rax+rax*1+0x0]
10: vmovupd xmm0,XMMWORD PTR [rsi+rax*1]
15: vmovupd xmm1,XMMWORD PTR [rdi+rax*1]
1a: vinsertf128 ymm0,ymm0,XMMWORD PTR [rsi+rax*1+0x10],0x1
22: vinsertf128 ymm1,ymm1,XMMWORD PTR [rdi+rax*1+0x10],0x1
2a: vaddpd ymm0,ymm1,ymm0
2e: vmovupd XMMWORD PTR [rdi+rax*1],xmm0
33: vextractf128 XMMWORD PTR [rdi+rax*1+0x10],ymm0,0x1
3b: add    rax,0x20
3f: cmp    rax,0x186a0
45: jne    10 <foo+0x10>
47: vzeroupper
4a: ret

4b: lea    rax,[rdi+0x20]
4f: cmp    rsi,rax
52: jae    9 <foo+0x9>
54: xor    eax,eax
56: nop    WORD PTR cs:[rax+rax*1+0x0]
60: vmovsd xmm0,QWORD PTR [rdi+rax*1]
65: vaddsd xmm0,xmm0,QWORD PTR [rsi+rax*1]
6a: vmovsd QWORD PTR [rdi+rax*1],xmm0
6f: add    rax,0x8
73: cmp    rax,0x186a0
79: jne    60 <foo+0x60>
```

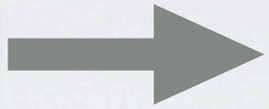
```
void foo(double * restrict a, double * restrict b)
{
    for(int i = 0; i < SIZE; i++)
    {
          a[i] += b[i];
    }
}
```



```
80:   push   rbp
81:   mov    r9,rdi
84:   and    r9d,0x1f
88:   mov    rbp,rsp
8b:   push   r12
8d:   shr    r9,0x3
91:   neg    r9
94:   push   rbx
95:   mov    edx,r9d
98:   and    rsp,0xffffffffffffffe0
9c:   add    rsp,0x20
a0:   and    edx,0x3
a3:   je     185 <foo1+0x105>
a9:   xor    eax,eax
ab:   mov    r8d,0x1869f
b1:   nop    DWORD PTR [rax+0x0]
b8:   vmovsd xmm0,QWORD PTR [rdi+rax*8]
bd:   mov    r10d,r8d
c0:   sub    r10d,eax
c3:   lea    r11d,[rax+0x1]
c7:   vaddsd xmm0,xmm0,QWORD PTR [rsi+rax*8]
cc:   vmovsd QWORD PTR [rdi+rax*8],xmm0
d1:   add    rax,0x1
d5:   cmp    edx,eax
d7:   ja     b8 <foo1+0x38>
d9:   mov    r12d,0x186a0
df:   mov    r8,r9
e2:   sub    r12d,edx
e5:   and    r8d,0x3
e9:   mov    edx,r12d
ec:   shr    edx,0x2
ef:   lea    ebx,[rdx*4+0x0]
f6:   test   ebx,ebx
f8:   je     140 <foo1+0xc0>
fa:   shl    r8,0x3
fe:   xor    eax,eax
100:  xor    ecx,ecx
102:  lea    r9,[rdi+r8*1]
106:  add    r8,rsi
109:  nop    DWORD PTR [rax+0x0]
110:  vmovupd xmm0,XMMWORD PTR [r8+rax*1]
116:  add    ecx,0x1
119:  vinsertf128 ymm0,ymm0,XMMWORD PTR [r8+rax*1+0x10],
0x1
120:
121:  vaddpd ymm0,ymm0,YMMWORD PTR [r9+rax*1]
127:  vmovapd YMMWORD PTR [r9+rax*1],ymm0
12d:  add    rax,0x20
131:  cmp    ecx,edx
133:  jb     110 <foo1+0x90>
135:  add    r11d,ebx
138:  sub    r10d,ebx
13b:  cmp    r12d,ebx
13e:  je     179 <foo1+0xf9>
140:  movsxd r11,r11d
143:  sub    r10d,0x1
147:  lea    rdx,[r11*8+0x0]
14e:
14f:  add    r11,r10
152:  lea    rcx,[rdi+r11*8+0x8]
157:  lea    rax,[rdi+rdx*1]
15b:  add    rdx,rsi
15e:  xchg   ax,ax
160:  vmovsd xmm0,QWORD PTR [rax]
164:  vaddsd xmm0,xmm0,QWORD PTR [rdx]
168:  add    rdx,0x8
16c:  vmovsd QWORD PTR [rax],xmm0
170:  add    rax,0x8
174:  cmp    rax,rcx
177:  jne    160 <foo1+0xe0>
179:  lea    rsp,[rbp-0x10]
17d:  pop    rbx
17e:  pop    r12
180:  pop    rbp
181:  vzeroupper
184:  ret
185:  mov    r10d,0x186a0
18b:  xor    r11d,r11d
18e:  jmp    d9 <foo1+0x59>
193:  data32 data32 data32 nop WORD PTR cs:[rax
+rax*1+0x0]
```

- GCC knows now that the arrays do not overlap
- It doesn't know if the arrays are aligned
  - peel loop to align the memory accesses
  - remainder loop when trip-count is not a multiple of the vector lane width

```
void foo(double * restrict a, double * restrict b)
{
   double *x = __builtin_assume_aligned(a, 16);
   double *y = __builtin_assume_aligned(b, 16);

   for(int i = 0; i < SIZE; i++)
   {
        x[i] += y[i];
   }
}
```



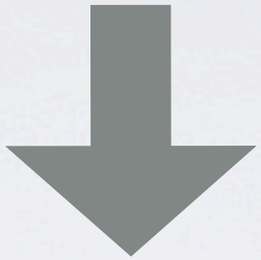
```
1a8:   vmovapd ymm0,YMMWORD PTR [rdi+rax*1]
1ad:   vaddpd ymm0,ymm0,YMMWORD PTR [rsi+rax*1]
1b2:   vmovapd YMMWORD PTR [rdi+rax*1],ymm0
1b7:   add    rax,0x20
1bb:   cmp    rax,0x186a0
1c1:   jne    1a8 <foo2+0x8>
```

- GCC finally generates optimal code
- Be ready to babysit the compiler with options and directives if you want to use autovectorization

- SIMD directives in ICC help in loop nests with no dependencies and branches
  - HPC friendly code
  - Xeon PHI is well supported as advertised
- Suggested pragmas:
  - #pragma unroll(N)
  - #pragma vector aligned
  - #pragma vector nontemporal
  - #pragma simd assert

```
for (i=0; i<count; i++){
    for (y=1; y < height-1; y++) {
        int c = 1 + y*WIDTHP+1;
        int n = c-WIDTHP;
        int s = c+WIDTHP;
        int e = c+1;
        int w = c-1;
        int nw = n-1;
        int ne = n+1;
        int sw = s-1;
        int se = s+1;
#pragma simd
#pragma vector nontemporal
        for (x=1; x < width-1; x++) {
            fout[c] = diag * fin[nw] +
            diag * fin[ne] +
            diag * fin[sw] +
            diag * fin[se] +
            next * fin[w] +
            next * fin[e] +
            next * fin[n] +
            next * fin[s] +
            ctr  * fin[c];

            // increment to next location
            c++;n++;s++;e++;w++;nw++;ne++;sw++;se++;
        }
    }
    REAL *ftmp = fin;
    fin = fout;
    fout = ftmp;
}
```

```
__global__ void mxm(float *X, float *Y){
const int i = blockIdx.x*blockDim.x + threadIdx.x;

    if(X[i] != 0)
            X[i] = X[i] - Y[i];
    else
            Y[i] = X[i] * Y[i];
}
```

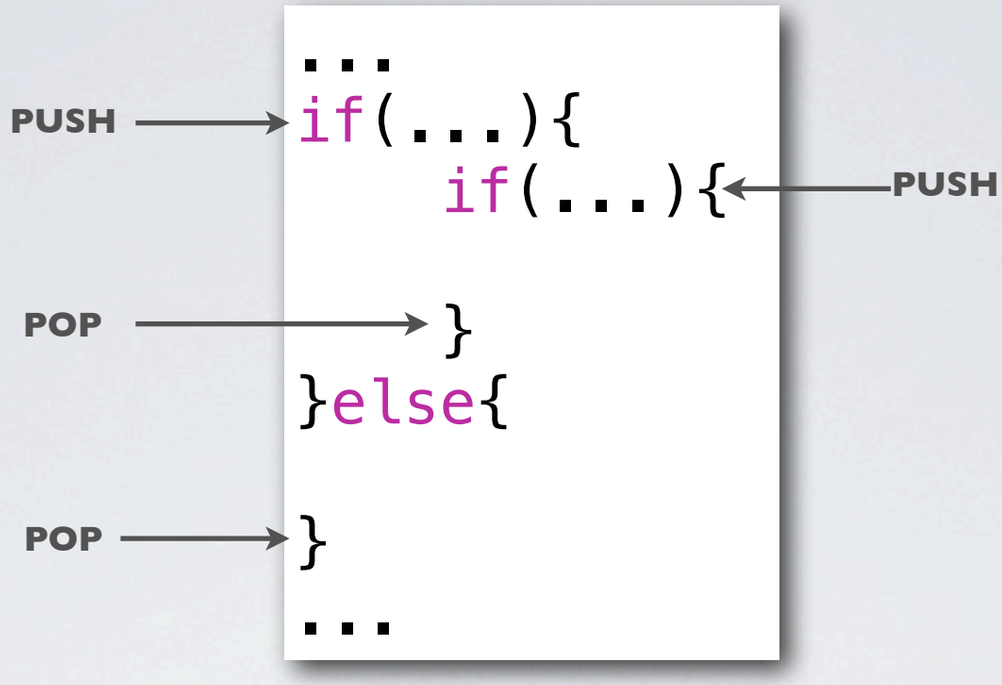


even though scalar code is generated, the hardware implicitly vectorizes it (SIMT)

```
/*0058*/         FSETP.NEU.AND P0, PT, R2, RZ, PT;
/*0060*/   @!P0 FMUL R3, R2, R0;
/*0068*/    @P0 FADD R0, R2, -R0;
/*0070*/   @!P0 ST.E [R6], R3;
/*0078*/    @P0 ST.E [R4], R0;
/*0080*/        EXIT ;
```

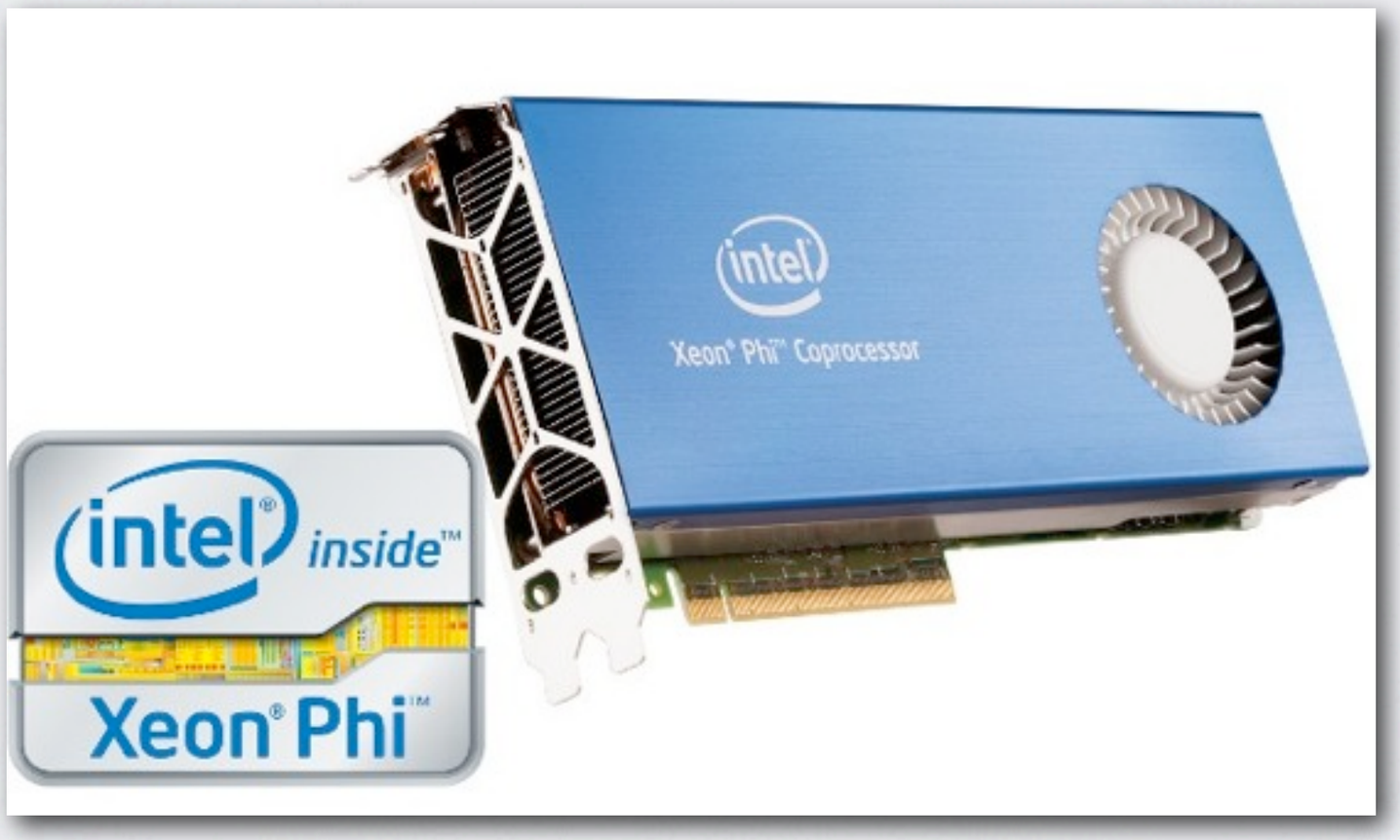
instead of branching, instructions are predicated

- Scalar Loads/stores in GPUs are automatically coalesced into vector loads/stores by the hardware
- Predicated instructions are generated for simple branching constructs



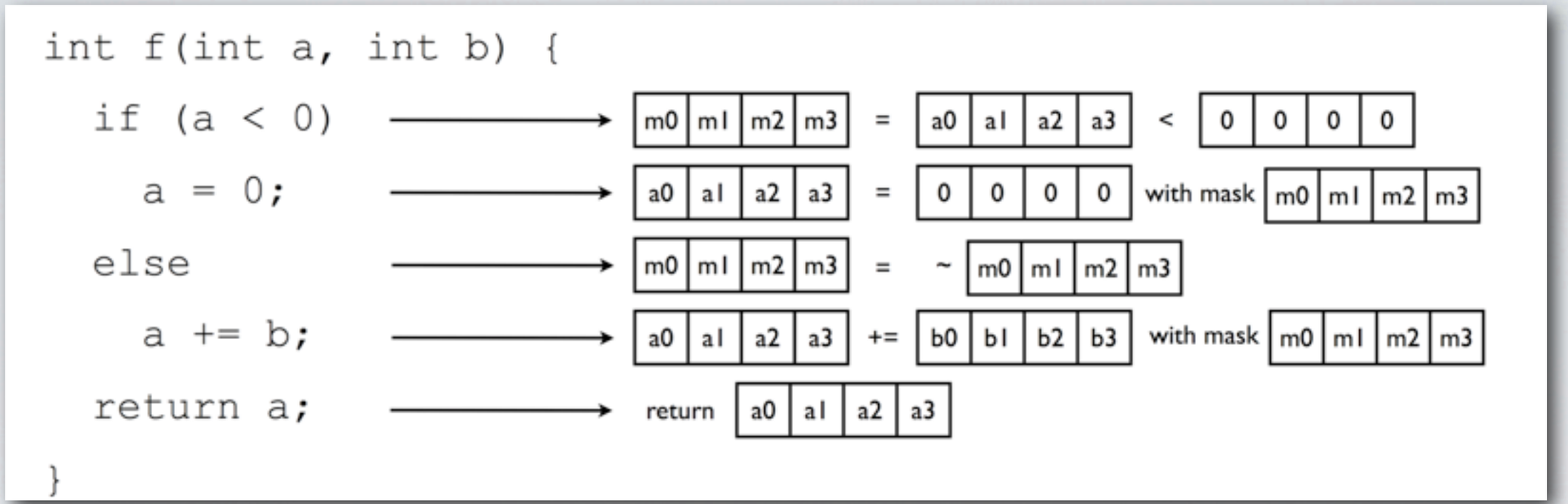
- Branch Synchronization Stack (BSS)
  - entries consists of a mask (1bit) that determines which lane commits its results
  - comes in handy for nested branching constructs
- Instruction markers
  - push/pop a mask on the BSS when a branch diverges/converges into/from multiple execution paths
- In other words, writing a vectorizing compiler for a GPU is an easier task than writing one for CPU

- Opmask registers are expressed using the notation "k1" through "k7".
- Instructions supporting conditional vector operation are expressed by attaching the notation {k[1-7]} next to the destination operand
- E.g.: `VADDPS zmm1 {k1}{z}, zmm2, zmm3`



- Intel SPMD Program Compiler (ISPC) extends a C-based language with "single program, multiple data" (SPMD) constructs
- An ISPC program describes the behavior of a single program instance
  - even though a "gang" of them is in reality being executed
  - gang size is usually no more than 2-4x the native SIMD width of the machine
- For CUDA affectionados
  - ISPC Program is similar to a CUDA thread
  - An ISPC gang is similar to a CUDA warp

Execution of a SPMD program with a gang size of 4



- Observations:
  - diverging control flow reduces the utilization of vector instructions
  - vectorization adds masking overhead

uniform variable is shared among program instances

make function available to be called from application code

foreach expresses a parallel computation

each program instance has a private instance of a non-uniform variable (a.k.a. varying variable)

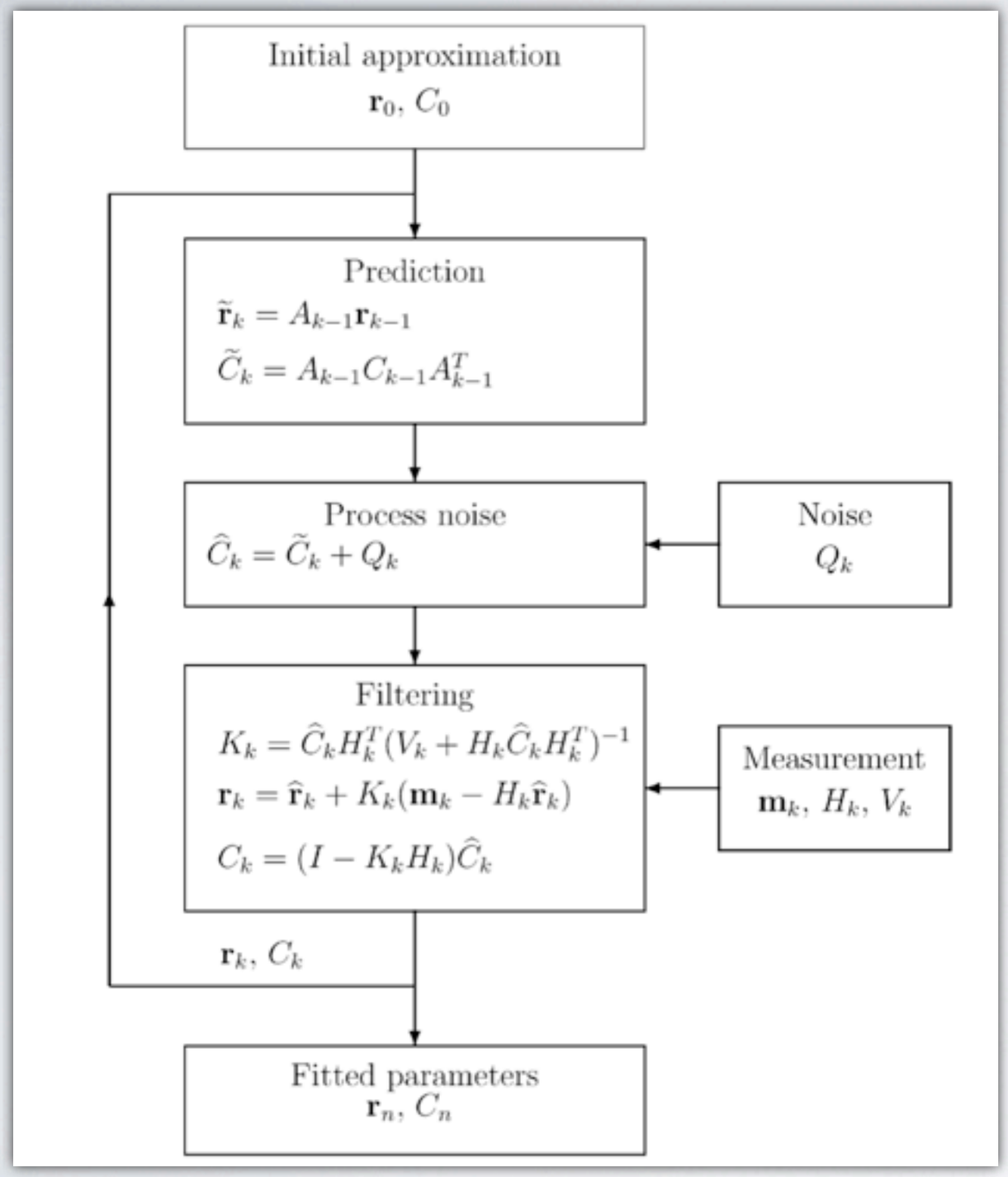
```
export void simple(uniform float vin[], uniform float vout[],
                   uniform int count) {
    foreach (index = 0 ... count) {
        float v = vin[index];
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);
        vout[index] = v;
    }
}
```

simple.ispc, compiled with ispc

```
#include <stdio.h>
#include "simple.h"

int main() {
    float vin[16], vout[16];
    for (int i = 0; i < 16; ++i)
        vin[i] = i;

    simple(vin, vout, 16);

    for (int i = 0; i < 16; ++i)
        printf("%d: simple(%f) = %f\n", i, vin[i], vout[i]);
}
```

main.c, compiled with GCC

ispc function is called like any other function from the C/C++ application

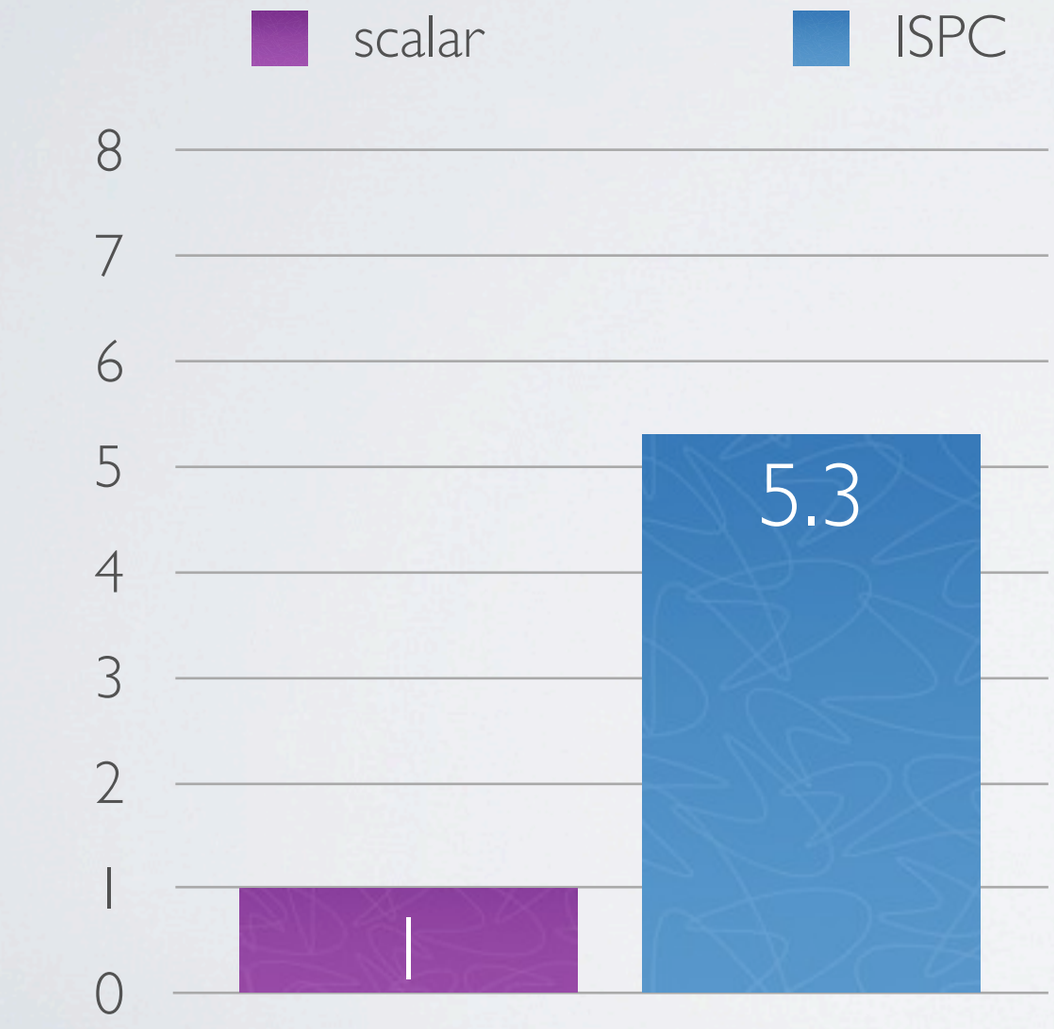


- The Kalman filter method is intended for finding the optimum estimation r of an unknown vector $\mathbf{r}^t$ according to the measurements m_k, k= 1 ...n, of the vector $\mathbf{r}^t$.

- Plenty of linear algebra operations so it's a good use case for vectorization.

- MKL is not an option since small dimensions are involved

```
export void startFilter(uniform KalmanFilter * uniform filter,
                        uniform KalmanFilterParameter * uniform param){
    foreach(i = 0 ... filter->ntracks){
        filterTrack(filter, param, i);
    }
}

inline void filterTrack(uniform KalmanFilter * uniform filter,
                        uniform KalmanFilterParameter * uniform param,
                        int i){
    ...
    for(uniform int h = 0; h < param->max_hit_count; h++){
        if(h >= param->hit_count[i])
            continue;

        predictState(filter, param, h, i);
        predictCovariance(filter, param, h, i);

        if(param->hits[h].is2Dim[i]){
            ...
            correctGain2D(filter, i);
            correctState2D(filter, i);
            correctCovariance2D(filter, i);
        }else{
            ...
            correctGain1D(filter, i);
            correctState1D(filter, i);
            correctCovariance1D(filter, i);
        }
    }
   ...
}
```



KalmanFilter speedup (double precision), Ivy Bridge

- Use vectorized math libraries like MKL or Eigen wherever you can
- On Xeons use ISPC to vectorize your code
  - compatible across different compilers, architectures and operating systems
- On Xeons/Xeons PHI use SIMD Directives and Cilk Plus to vectorize your code
  - if you can use the ICC compiler
  - if you have "nice" loop nests (HPC-code like)