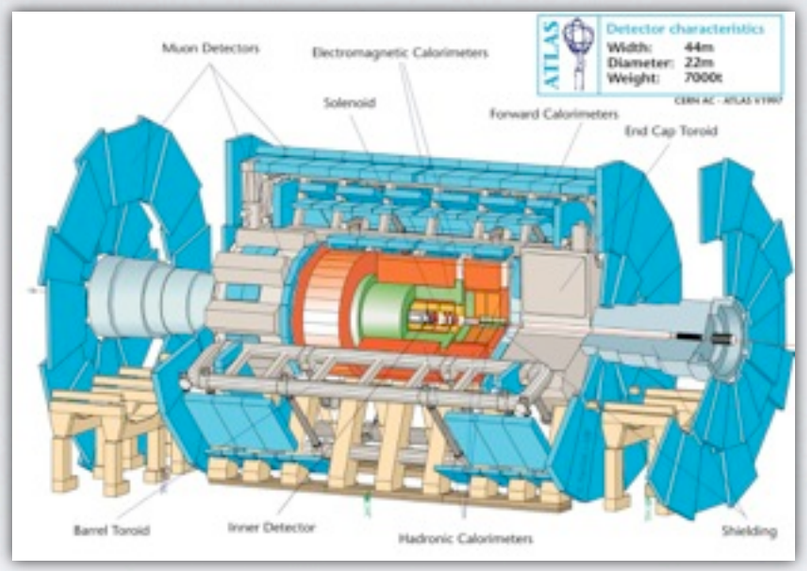
VECTORIZATION IN ATLAS

Roberto A. Vitillo (LBNL)

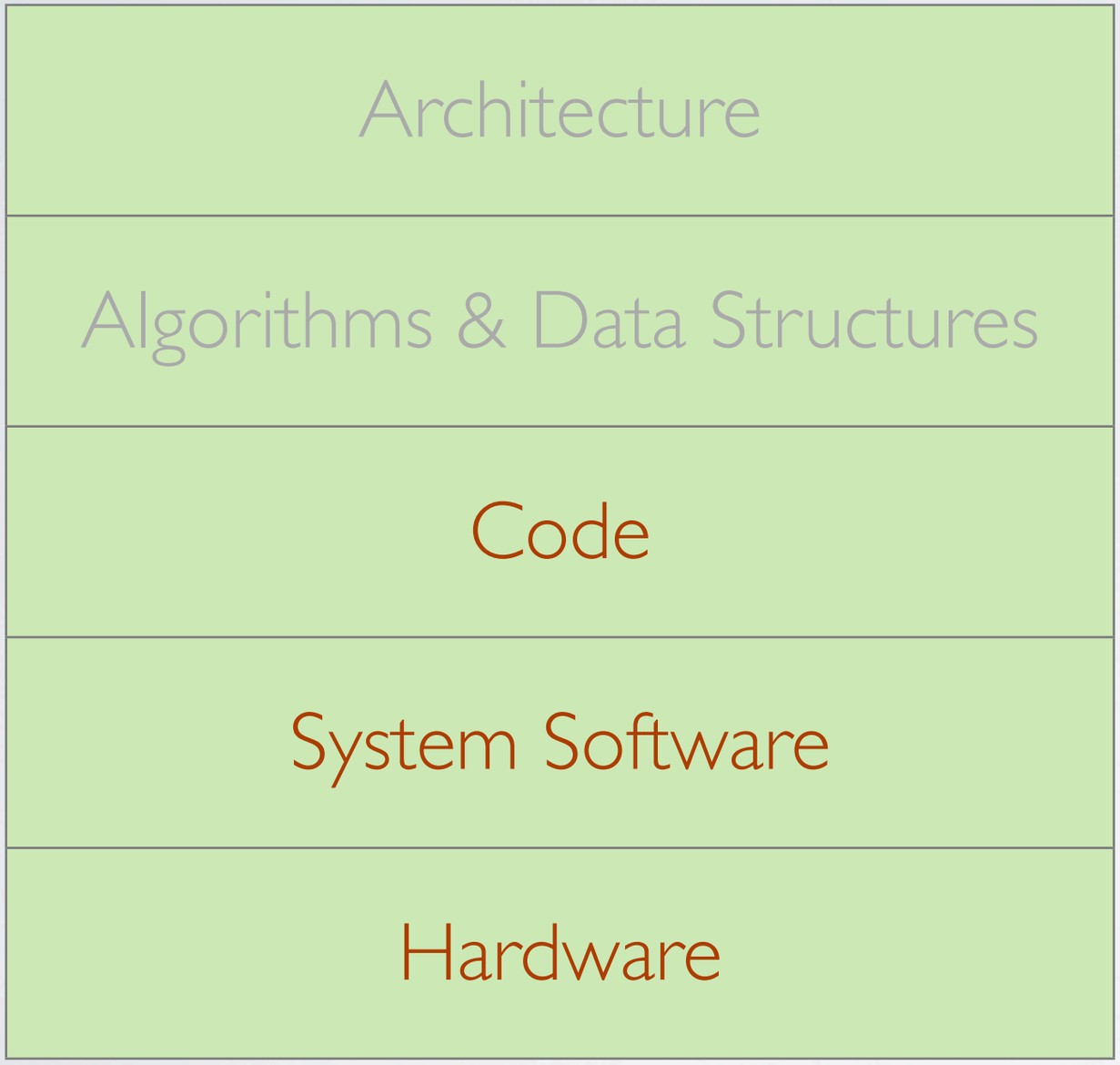
8/7/2013 NERSC, App readiness meeting

THE ATLAS EXPERIMENT



- ATLAS is one of the six particle physics experiment at the Large Hadron Collider (LHC)
- Proton beams interact in the center of the detector
- The ATLAS detector has been designed to provide the most complete information of the ~1000 particles that emerge from the proton collisions
- ATLAS acts as a huge digital camera composed of ~100 Million of channels which provide a "picture" of about 1 Mbyte defined as event
- Each event is fully analyzed to obtain the information needed for the physics analysis
- The software behind ATLAS consists of millions of LOC; a monstrosity to optimize!

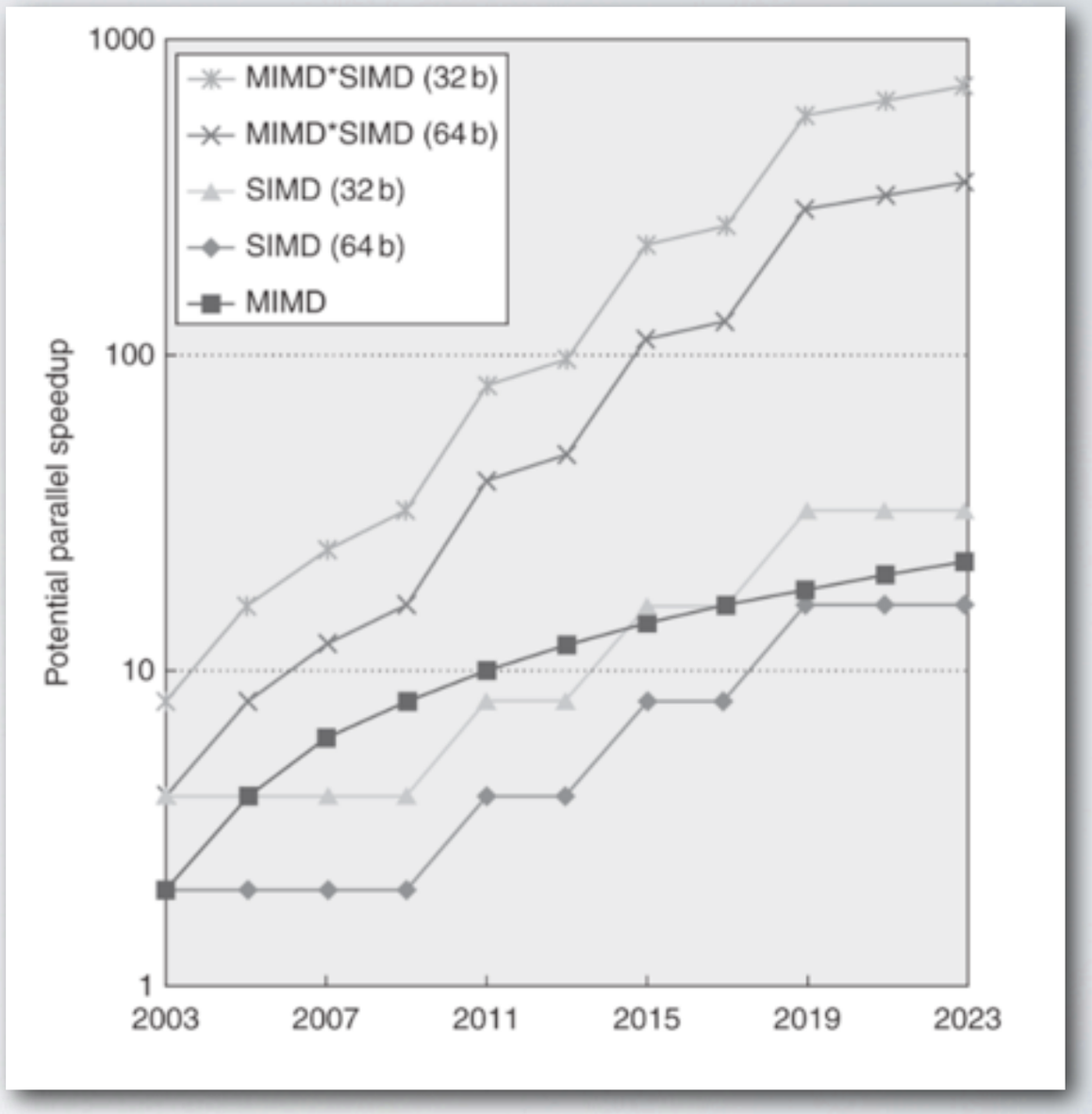
DESIGN LEVELS



- Independent changes on different levels causes speedups to multiply
- Clock speed free lunch is over
  - hardware parallelism is increasing
- How can parallelism be exploited at the bottom layers?

EXPLOITING SIMD

- **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
  - exploiting SIMD is even more of importance for the Xeon PHI and future architectures



Source: "Computer Architecture, A Quantitative Approach"

EXPLOITING SIMD

ATLAS USE CASES

- Linear algebra operations
  - small rectangular matrices (e.g. 2x5, 3x5, 3x6, ...) for error propagation
  - small square matrices (e.g. 3x3, 4x4, 5x5, ...) for transforms in geometry calculations
  - separate use of matrices up to ~25x25 in a few places
- Hot loops that invoke transcendental functions
- Other numerical hotspots, e.g. Kalman Filter in Tracking

EXPLOITING SIMD

APPROACHES

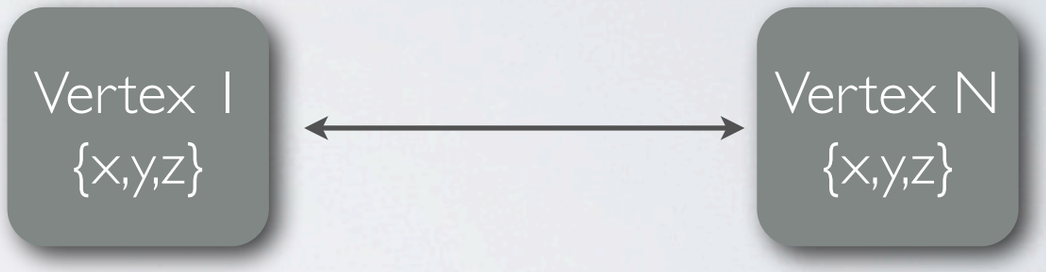
1. hand-tuning numerical hotspots

2. vectorized linear algebra libraries

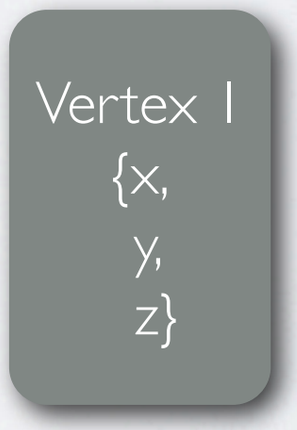
3. autovectorization

4. language extensions

Horizontal Vectorization



Vertical Vectorization



APPROACH 1

HAND TUNING NUMERICAL HOTSPOTS

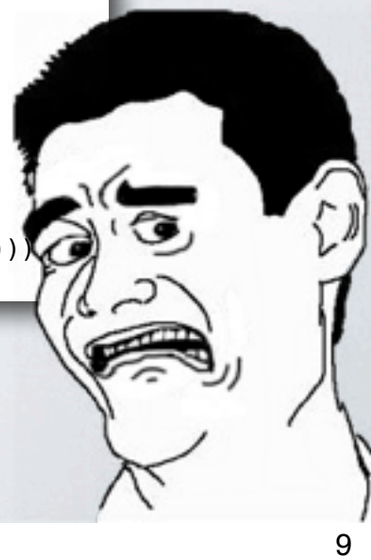
HANDTUNING

- GOoDA told us that most cycles are spent in *RungeKuttaPropagator::rungeKuttaStep*
- Most nested loop accounts for ~50% of its cycles
  - contains lots of floating point operations
- Good candidate for vectorization
  - autovectorization failed

```
for(int i=0; i<42; i+=7) {
    double* dR   = &P[i];
    double* dA   = &P[i+3];

    double dA0   = H0[ 2]*dA[1]-H0[ 1]*dA[2];
    double dB0   = H0[ 0]*dA[2]-H0[ 2]*dA[0];
    double dC0   = H0[ 1]*dA[0]-H0[ 0]*dA[1];

    if(i==35) {dA0+=A0; dB0+=B0; dC0+=C0;}

    double dA2   = dA0+dA[0];
    double dB2   = dB0+dA[1];
    double dC2   = dC0+dA[2];

    double dA3   = dA[0]+dB2*H1[2]-dC2*H1[1];
    double dB3   = dA[1]+dC2*H1[0]-dA2*H1[2];
    double dC3   = dA[2]+dA2*H1[1]-dB2*H1[0];

    if(i==35) {dA3+=A3-A00; dB3+=B3-A11; dC3+=C3-A22;}

    double dA4   = dA[0]+dB3*H1[2]-dC3*H1[1];
    double dB4   = dA[1]+dC3*H1[0]-dA3*H1[2];
    double dC4   = dA[2]+dA3*H1[1]-dB3*H1[0];

    if(i==35) {dA4+=A4-A00; dB4+=B4-A11; dC4+=C4-A22;}

    double dA5   = dA4+dA4-dA[0];
    double dB5   = dB4+dB4-dA[1];
    double dC5   = dC4+dC4-dA[2];

    double dA6   = dB5*H2[2]-dC5*H2[1];
    double dB6   = dC5*H2[0]-dA5*H2[2];
    double dC6   = dA5*H2[1]-dB5*H2[0];

    if(i==35) {dA6+=A6; dB6+=B6; dC6+=C6;}

    dR[0]+=(dA2+dA3+dA4)*S3; dA[0]=(dA0+dA3+dA3+dA5+dA6)*.33333333;
    dR[1]+=(dB2+dB3+dB4)*S3; dA[1]=(dB0+dB3+dB3+dB5+dB6)*.33333333;
    dR[2]+=(dC2+dC3+dC4)*S3; dA[2]=(dC0+dC3+dC3+dC5+dC6)*.33333333;
}
```

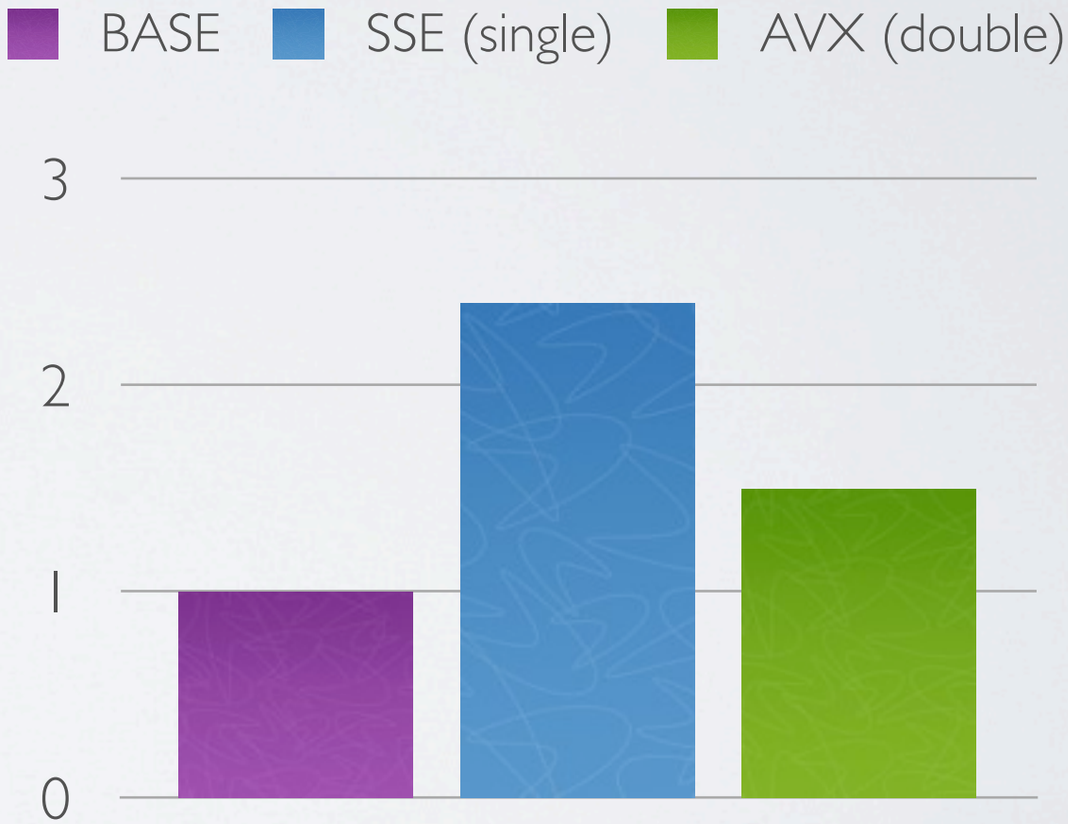
HANDTUNING

```
for(int i = 0; i < 42; i+=7){
    __m256d dR = _mm256_loadu_pd(&P[i]);

    __m256d dA = _mm256_loadu_pd(&P[i + 3]);
    __m256d dA_201 = CROSS_SHUFFLE_201(dA);
    __m256d dA_120 = CROSS_SHUFFLE_120(dA);

    __m256d d0 = _mm256_sub_pd(_mm256_mul_pd(H0_201, dA_120), _mm256_mul_pd(H0_120, dA_201));

    if(i==35){
        d0 = _mm256_add_pd(d0, V0_012);
    }

    __m256d d2 = _mm256_add_pd(d0, dA);
    __m256d d2_201 = CROSS_SHUFFLE_201(d2);
    __m256d d2_120 = CROSS_SHUFFLE_120(d2);

    __m256d d3 = _mm256_sub_pd(_mm256_add_pd(dA, _mm256_mul_pd(d2_120, H1_201)), _mm256_mul_pd(d2_201, H1_120));
    __m256d d3_201 = CROSS_SHUFFLE_201(d3);
    __m256d d3_120 = CROSS_SHUFFLE_120(d3);

    if(i==35){
        d3 = _mm256_add_pd(d3, _mm256_sub_pd(V3_012, A_012));
    }

    __m256d d4 = _mm256_sub_pd(_mm256_add_pd(dA, _mm256_mul_pd(d3_120, H1_201)), _mm256_mul_pd(d3_201, H1_120));

    if(i==35){
        d4 = _mm256_add_pd(d4, _mm256_sub_pd(V4_012, A_012));
    }

    __m256d d5 = _mm256_sub_pd(_mm256_add_pd(d4, d4), dA);
    __m256d d5_201 = CROSS_SHUFFLE_201(d5);
    __m256d d5_120 = CROSS_SHUFFLE_120(d5);

    __m256d d6 = _mm256_sub_pd(_mm256_mul_pd(d5_120, H2_201), _mm256_mul_pd(d5_201, H2_120));

    if(i==35){
        d6 = _mm256_add_pd(d6, V6_012);
    }

    _mm256_storeu_pd(&P[i], _mm256_add_pd(dR, _mm256_mul_pd(_mm256_add_pd(d2, _mm256_add_pd(d3, d4)), S3_012)));
    _mm256_storeu_pd(&P[i + 3], _mm256_mul_pd(C_012, _mm256_add_pd(d0, _mm256_add_pd(d3, _mm256_add_pd(d3, _mm256_add_pd(d5, d6))))));
}
```



HANDTUNING

- Tested on a Sandy Bridge-EP CPU
- SSE version with single precision intrinsics is 2.4x faster
- AVX version with double precision intrinsics is 1.5x faster
  - slower than SSE in this particular example because of costly cross lane permutations
  - not as mature as SSE
  - AVX2 (Haswell) will change that



HANDTUNING

- Hand-vectorizing may be suitable for vertical vectorization or when maximum speed-up is required and/or other approaches fail
- Using compiler intrinsics or inline assembly is not ideal
- Options:
  - C++ Vector Class Library
    http://www.agner.org/optimize/vectorclass.zip
  - VC Library (recommended)
    http://code.compeng.uni-frankfurt.de/projects/vc

VC LIBRARY

- Implements vector classes that abstract the SIMD registers
  - vector register size is not directly exposed
  - memory abstraction allows to handle uniformly arrays of any size
  - masked ops syntax

```
void testVc(){
  Vc::Memory<double_v, SIZE> x;
  Vc::Memory<double_v, SIZE> y;

  for(int i = 0; i < x.vectorsCount(); i++){
      y.vector(i) = cos(x.vector(i) * 3);
  }
}
```

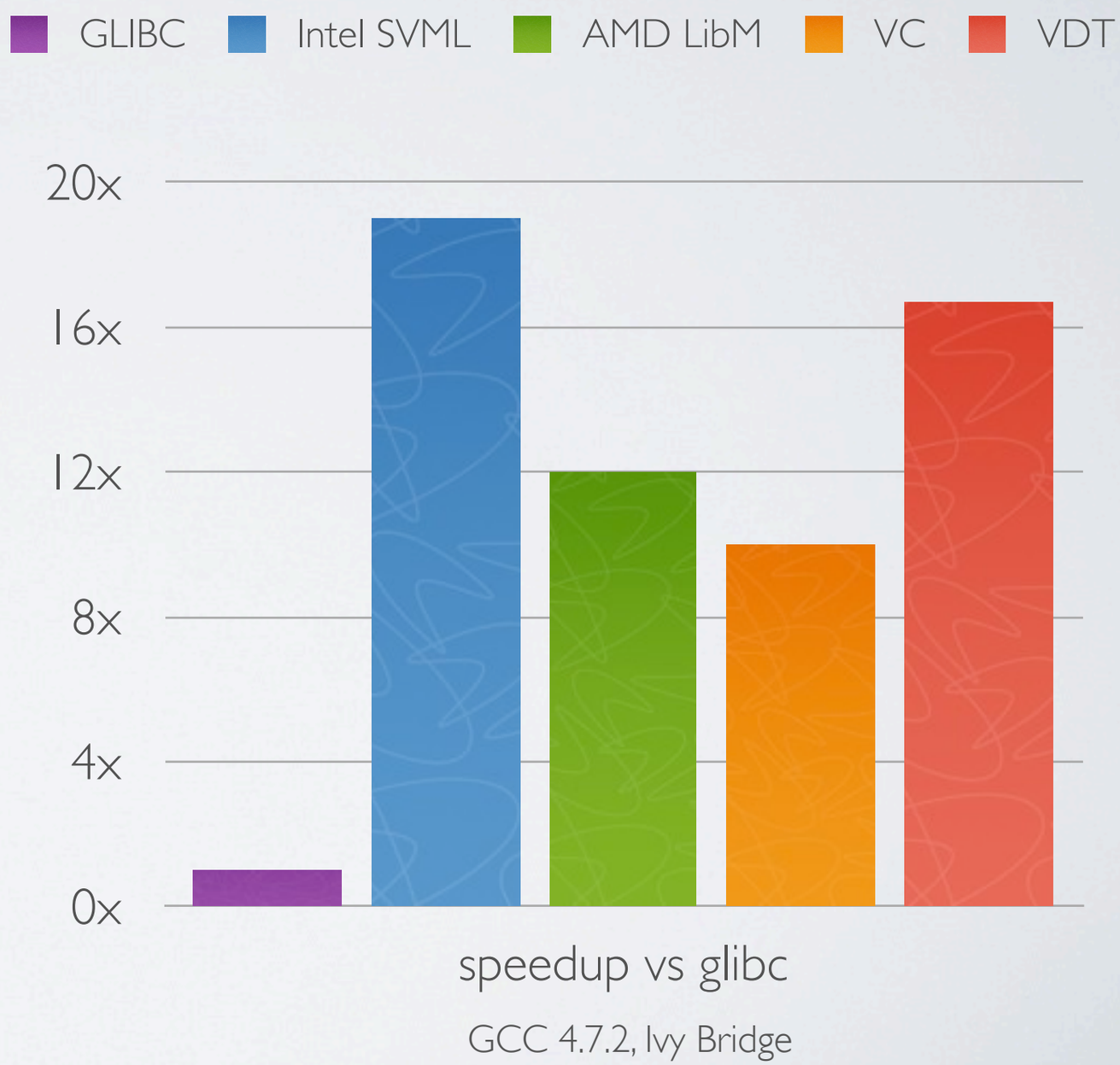
- Transcendental functions are implemented within the library
- Vectors implemented with SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX
- Implementation is chosen at compile-time

TRANSCENDENTAL FUNCTIONS

VECTOR PERFORMANCE

Accuracy				
GLIBC 2.17	SVML 11.0.1	AMD LibM 3	VC 0.6.7-dev	VDT 0.2.3
2 ulp	2-4 ulp	1 ulp	1160 ulp	2 ulp

- Test performed applying cos() on an array of 100 doubles
- GLIBC
  - repeatedly calls scalar function on vector
- AMD LibM
  - supports only SSE2 for non-AMD processors
- VDT
  - accuracy comparable to SVML, see: http://indico.cern.ch/contributionDisplay.py?contribId=4&sessionId=9&confId=202688



APPROACH 2

VECTORIZED LINEAR ALGEBRA LIBRARIES

(FOR SMALL MATRICES)

MATRIX MULTIPLICATION

- Showcasing vertical vectorization
- Simplest possible example: 4x4 double precision matrix multiplication
  - matrix fits in two cache lines
  - AVX supports vectors of 4 doubles
- *OptimizedMult* vectorized without horizontal sums
- Speedup of 3 vs nonvectorized *BasicMult*

BasicMult

```
for(int i = 0; i < 16; i+=4){
   for(int j = 0; j < 4; j++){
        z[i+j] = x[i] * y[j] +       \
                 x[i+1] * y[4 + j] + \
                 x[i+2] * y[8 + j] + \
                 x[i+3] * y[12 + j];
   }
}
```

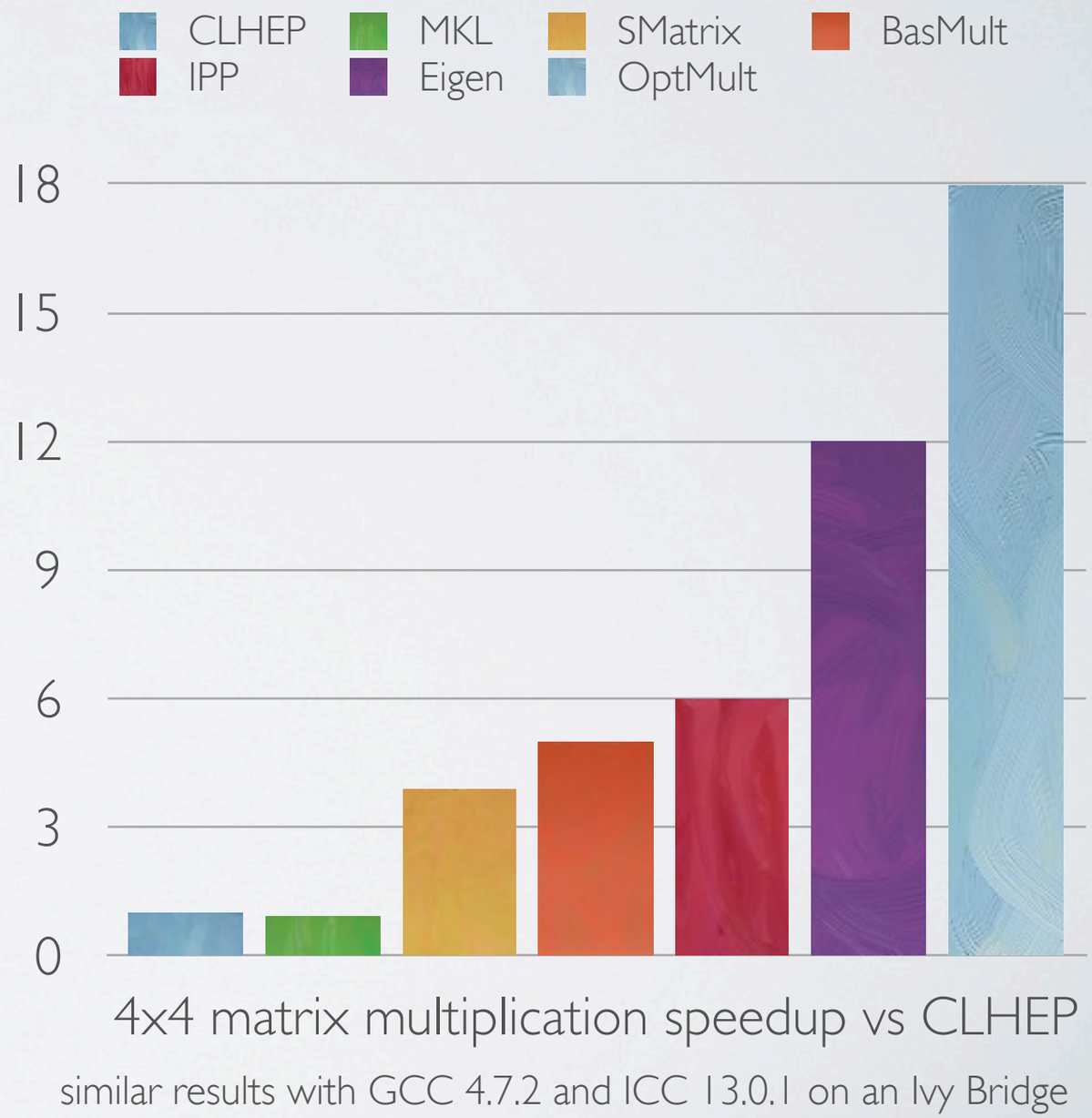
OptimizedMult

```
for(int i = 0; i < 16; i+=4){
    Vec4d r1 = Vec4d(x[i]) * Vec4d(y);

    for(int j = 1; j < 4; j++){
        r1 += Vec4d(x[i+j]) * Vec4d(&y[j*4]);
    }

    r1.store(&z[i]);
}
```

MATRIX MULTIPLICATION

SQUARE MATRICES

- Eigen3 doesn't support yet AVX
- CLHEP provides a generic interface for any-dimension matrix
- MKL is optimized for large matrices and BLAS operations: $C = \alpha AB + \beta C$
- SMatrix operations are not vectorized
- Benefits of template expressions are not shown in this simple example
- OptMult represents the maximum speedup that can be achieved

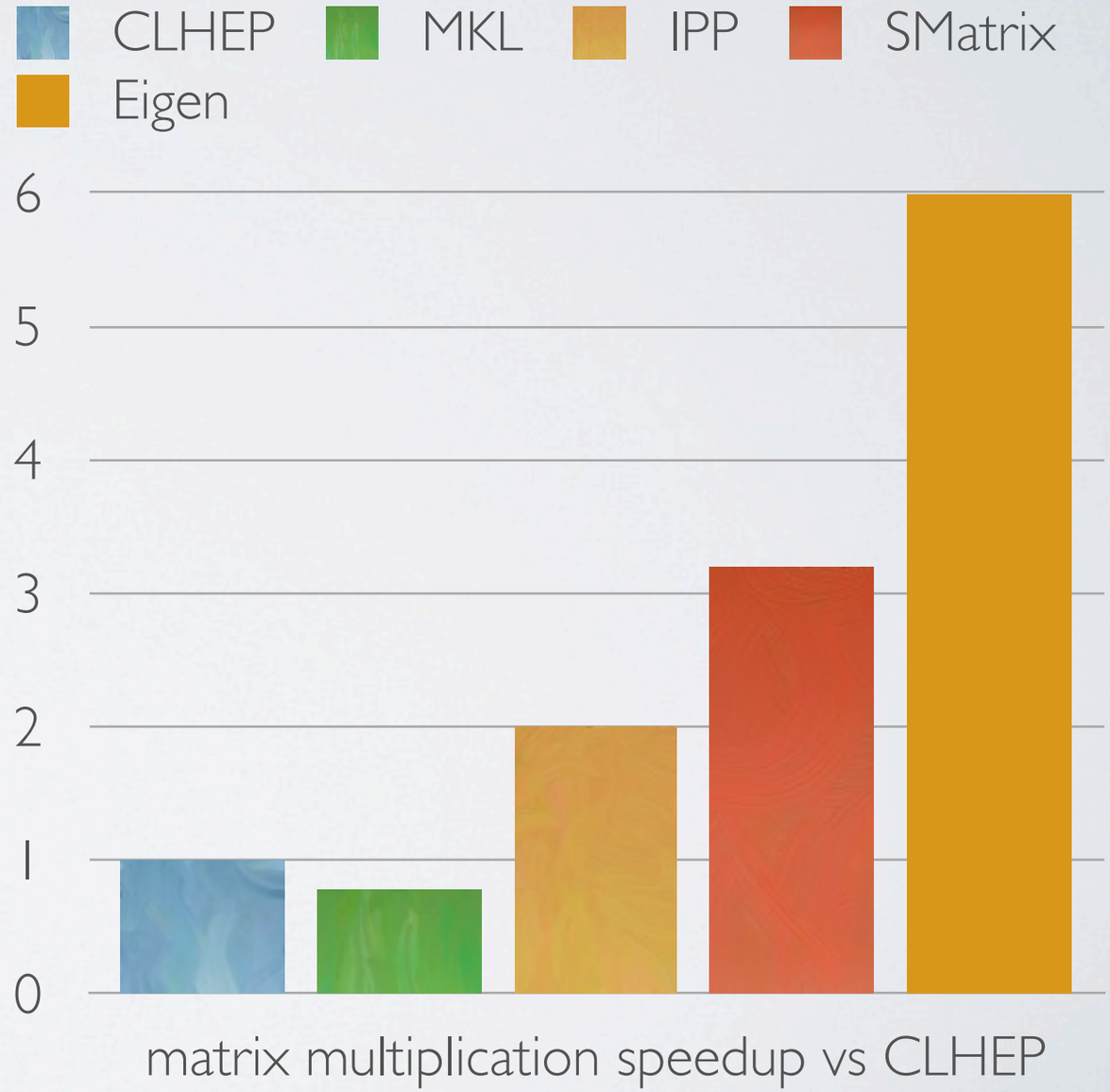


similar results with GCC 4.7.2 and ICC 13.0.1 on an Ivy Bridge

MATRIX MULTIPLICATION

RECTANGULAR MATRICES

- Evaluating $A_{5\times3} \times B_{3\times5}$
- None of the libraries is using vectorized code!
  - vectorization is not trivial in this case
  - alternative: horizontal vectorization



similar results with GCC 4.7.2 and ICC 13.0.1 on an Ivy Bridge

APPROACH 3

AUTOVECTORIZATION

AUTOVECTORIZATION CAVEATS

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

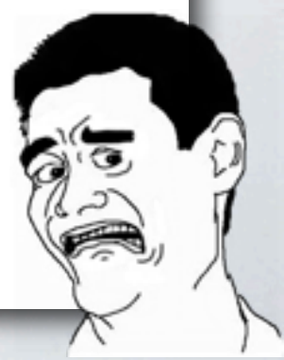
what we would like:

```
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,0xc3500
1c1:   jne    1a8 <foo2+0x8>
```

- Alignment of arrays unknown to the compiler
- GCC has to check if the arrays overlap but doesn't check if the arrays are aligned!
- If they do, scalar addition is performed
  - otherwise loop is only partially vectorized

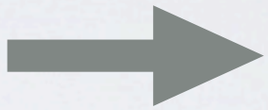
what we actually get:

```
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,0xc3500
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,0xc3500
79: jne    60 <foo+0x60>
```



AUTOVECTORIZATION CAVEATS

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

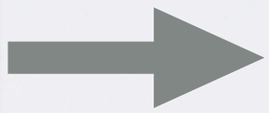


```
80:    push   rbp
81:    mov    r9,rdi
84:    and    r9d,0x1f
88:    mov    rbp,rsp
8b:    push   r12
8d:    shr    r9,0x3
91:    neg    r9
94:    push   rbx
95:    mov    edx,r9d
98:    and    rsp,0xfffffffffffffe0
9c:    add    rsp,0x20
a0:    and    edx,0x3
a3:    je     185 <foo1+0x105>
a9:    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 but...
- It doesn't know if the arrays are aligned and doesn't check for it
  - loop only partially vectorized

AUTOVECTORIZATION CAVEATS

```
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,0xc3500
1c1:   jne    1a8 <foo2+0x8>
```

- GCC finally generates optimal code
- Don't assume that the compiler generates efficient vector code
- For ICC see: http://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

LAR CALIBRATION

- Input for test: Barrel-presampler calibration run
- According to VTune, the hottest hotspot in this job is the wave convolution method
  - called from inside a fit method (many million times)
- Autovectorizing the inner loop of the convolution
  - requires -fassociative-math, -fno-signed-zeros and -fno-trapping-math
  - CPU time 121 -> 76 seconds (~ 40% faster)
  - result is not identical; relative diff about 1e-7

APPROACH 4

LANGUAGE EXTENSIONS

CILK PLUS

- Introduces
  - array notations, data parallelism for arrays or sections of arrays
  - SIMD pragma, specifies that a loop is to be vectorized
  - elemental functions, i.e. functions that can be vectorized when called from within an array notation or a *#pragma simd loop*
- Supported by recent ICC releases and a GCC branch
- Offers other features not related to vectorization, see: http://software.intel.com/en-us/intel-cilk-plus

```
// Copies x[i..i+n-1] to z[i..i+n-1]
z[i:n] = x[i:n];

// Sets z[i..i+n-1] to twice the
// corresponding elements in x[i+1..i+n]
z[i:n] = 2*x[i+1:n];
```

```
_declspec (vector)
float v_add(float x, float y){
    return x+y;
}

void foo(){
#pragma simd
  for(int j = 0; j < N; ++j){
    r[j] = v_add(a[j],b[j]);
  }
}
```

OPENMP 4

- Based on the SIMD directives of Cilk Plus
- Provides data sharing clauses
- Vectorizes functions by promoting scalar parameters to vectors and replicates control flow using elemental functions
- Up to 4x speedup vs autovectorization
  http://iwomp-2012.caspur.it/sites/iwomp-2012.caspur.it/files/Klemm_SIMD.pdf
- OpenMP 4.0 specification was released recently

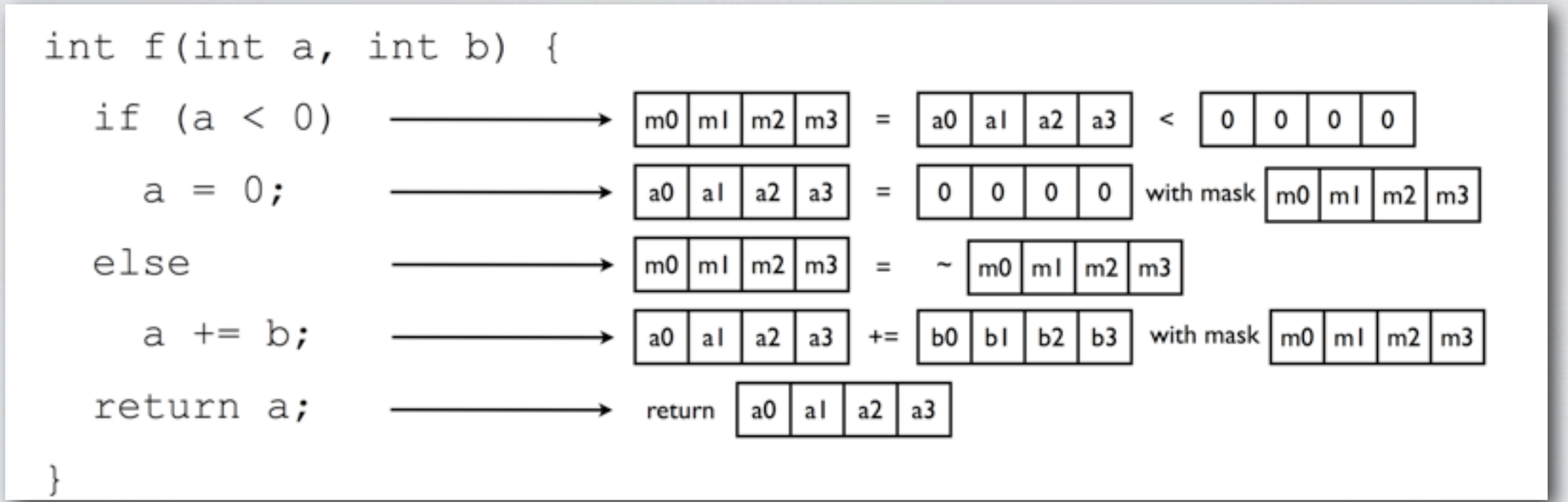
  http://openmp.org/wp/openmp-specifications/

```
#pragma omp simd nomask
float sqdiff(float x1, float x2){
    return (x1 - x2) * (x1 - x2);
}

void euc_dist(){
...
#pragma omp parallel simd for
    for(int i = 0; i < N; i++){
        d[i] = sqrt(sqdiff(x1[i], x2[i]) +
                    sqdiff(y1[i], y2[i]));
    }
}
```

MEET ISPC

- 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

SPMD PARADIGM



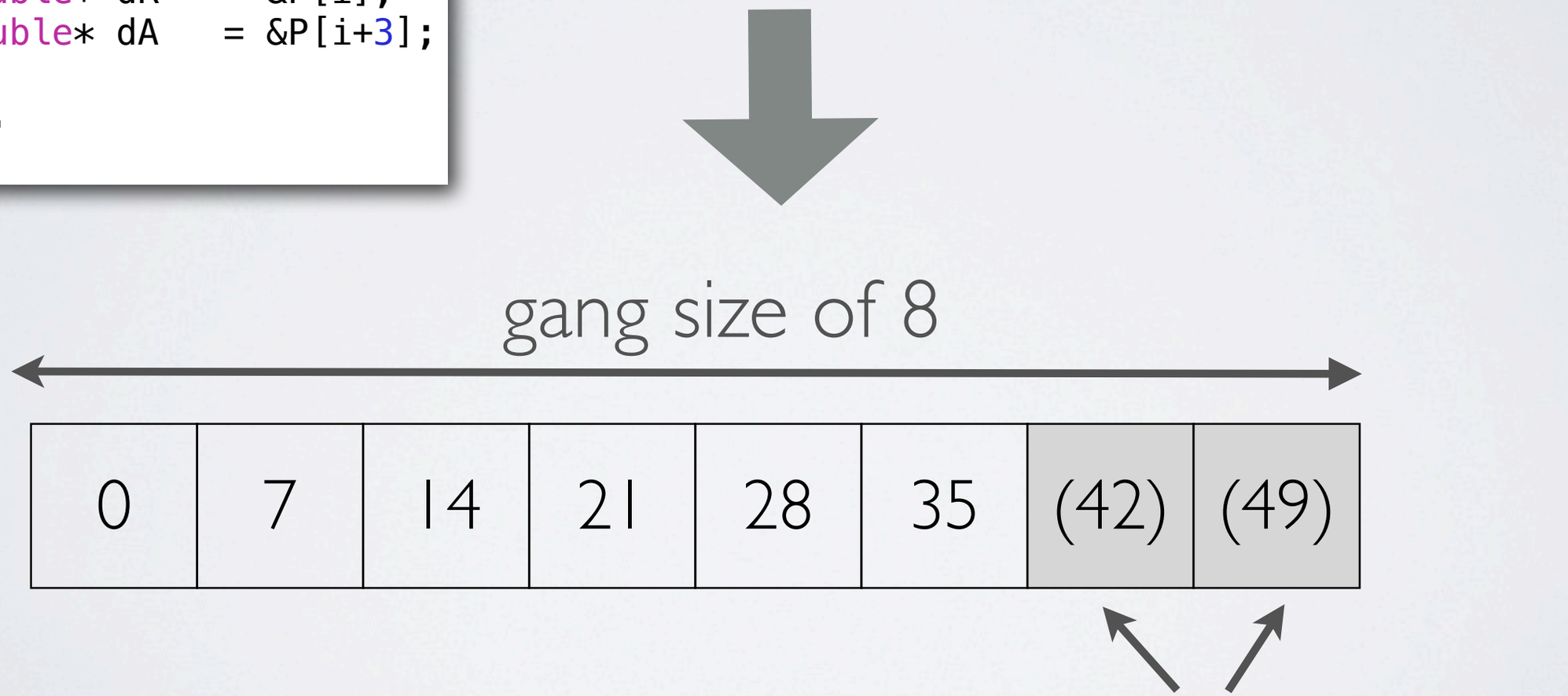
Execution of a SPMD program with a gang size of 4

DEBUGGING SUPPORT

BEYOND GDB

```
foreach(k = 0 ... 6){
    int i = k * 7;

    print("%\n", i);

    double* dR   = &P[i];
    double* dA   = &P[i+3];

    ...
}
```

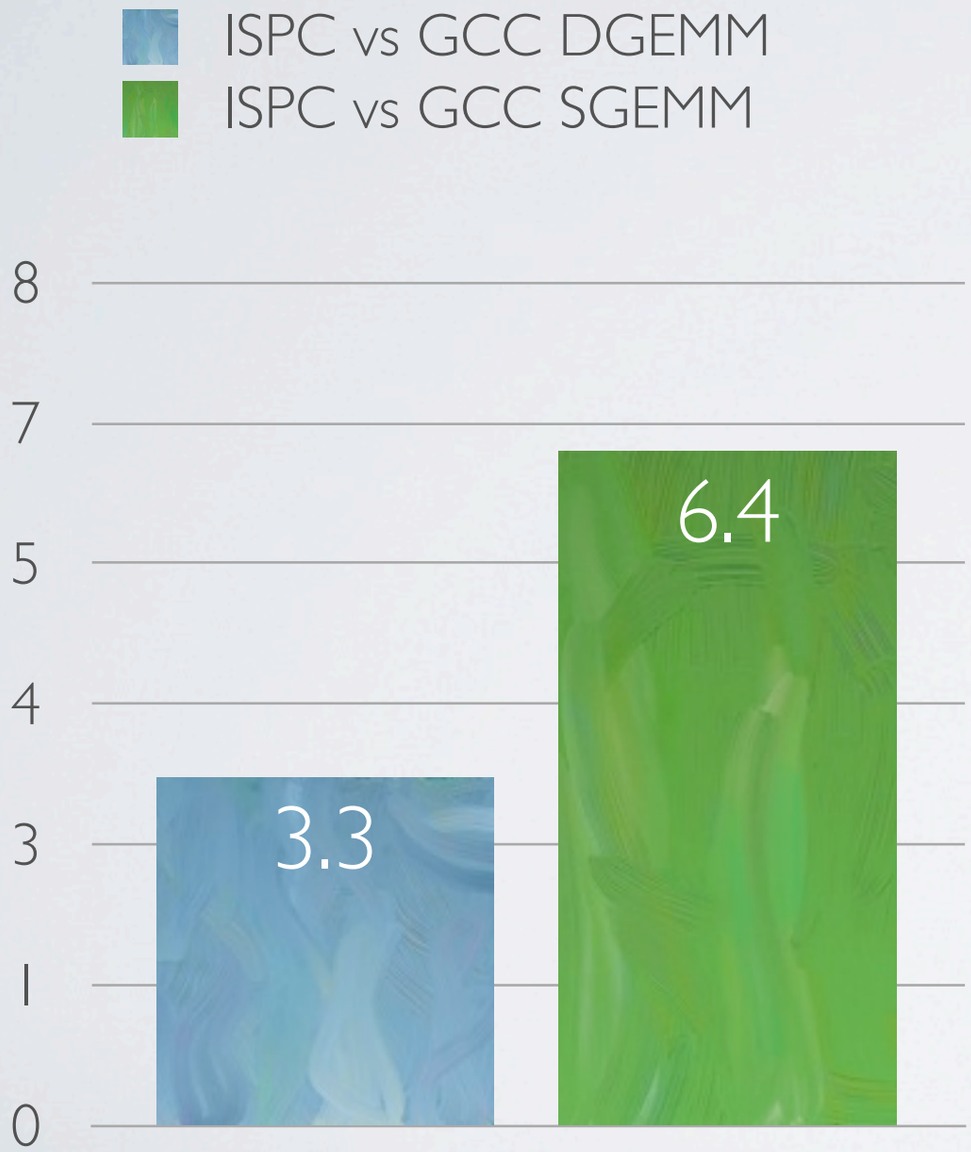
Prints [0, 7, 14, 21, 28, 35, ((42)), ((49))]



MATRIX MULTIPLICATION

EXPLOITING HORIZONTAL VECTORIZATION WITH SMALL MATRICES

- ISPC vs GCC DGEMM
- ISPC vs GCC SGEMM




```
inline void mxm(uniform scalar_t * uniform A,
                uniform scalar_t * uniform B,
                uniform scalar_t * uniform C,
                uniform int M,
                uniform int N,
                uniform int K,
                uniform int nmat,
                int idx)
{
    for(uniform int i = 0; i < M; i++){
        for(uniform int j = 0; j < N; j++){
            scalar_t sum = 0;

            for(uniform int k = 0; k < K; k++){
                sum += A[i*K*nmat + k*nmat + idx] * B[k*N*nmat + j*nmat + idx];
            }

            C[i*N*nmat + j*nmat + idx] = sum;
        }
    }
}

export void gemm(uniform scalar_t * uniform A,
                 uniform scalar_t * uniform B,
                 uniform scalar_t * uniform C,
                 uniform int M,
                 uniform int N,
                 uniform int K,
                 uniform int nmat)
{
    foreach(i = 0 ... nmat){
        mxm(A, B, C, M, N, K, nmat, i);
    }
}
```

xGEMM 5x5 speedup over 1000 matrices (GCC 4.8 -O3)

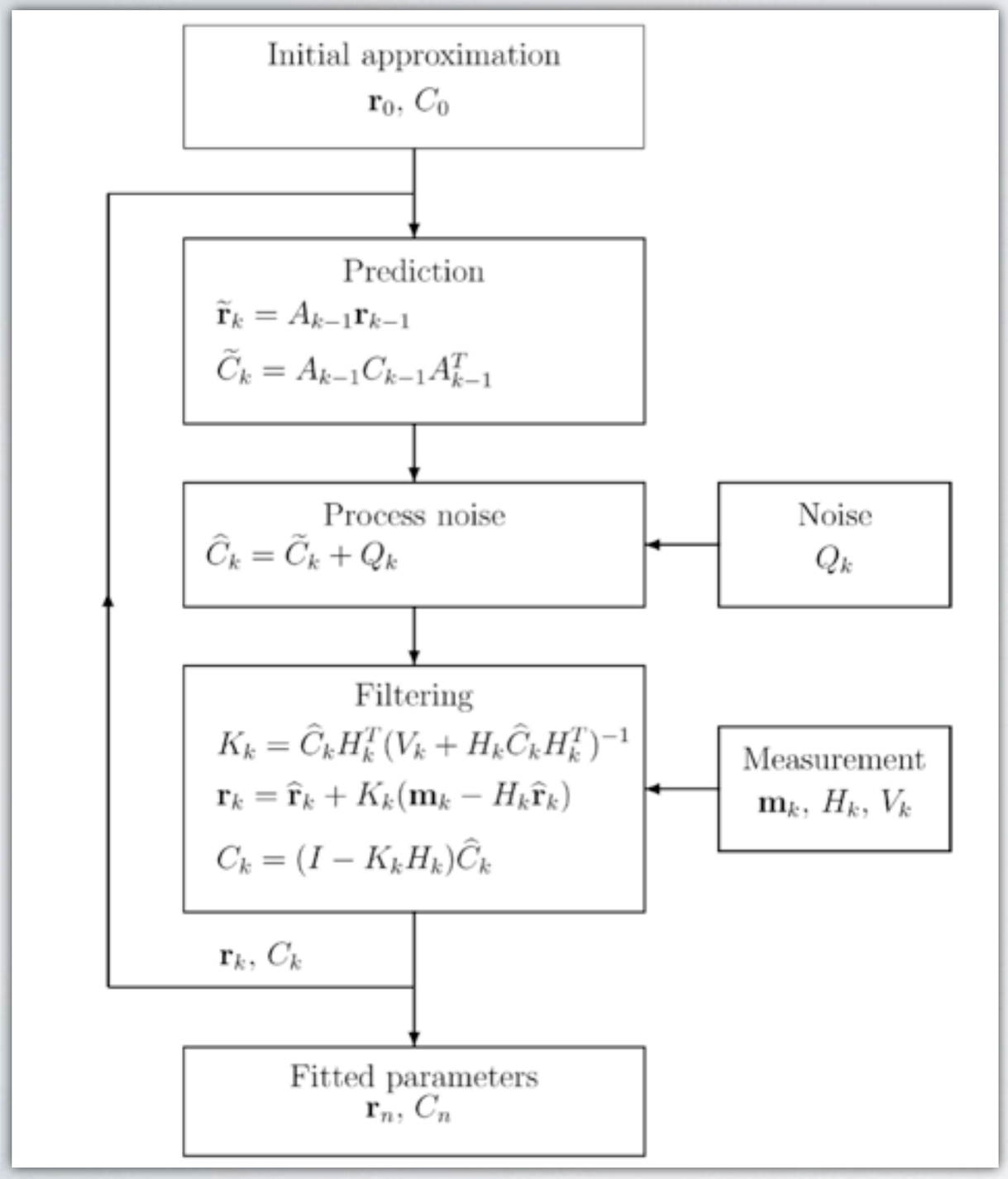
MATRIX MULTIPLICATION

EXPLOITING HORIZONTAL VECTORIZATION WITH SMALL MATRICES


```
inline void mxm(uniform scalar_t * uniform A,
                uniform scalar_t * uniform B,
                uniform scalar_t * uniform C,
                uniform int M,
                uniform int N,
                uniform int K,
                uniform int nmat,
                int idx)
{
    for(uniform int i = 0; i < M; i++){
        for(uniform int j = 0; j < N; j++){
            scalar_t sum = 0;

            for(uniform int k = 0; k < K; k++){
                sum += A[i*K*nmat + k*nmat + idx] * B[k*N*nmat + j*nmat + idx];
            }

            C[i*N*nmat + j*nmat + idx] = sum;
        }
    }
}

export void gemm(uniform scalar_t * uniform A,
                 uniform scalar_t * uniform B,
                 uniform scalar_t * uniform C,
                 uniform int M,
                 uniform int N,
                 uniform int K,
                 uniform int nmat)
{
    foreach(i = 0 ... nmat){
        mxm(A, B, C, M, N, K, nmat, i);
    }
}
```

xGEMM 5x5 speedup over 1000 matrices (GCC 4.8 -O3)

KALMAN FILTER



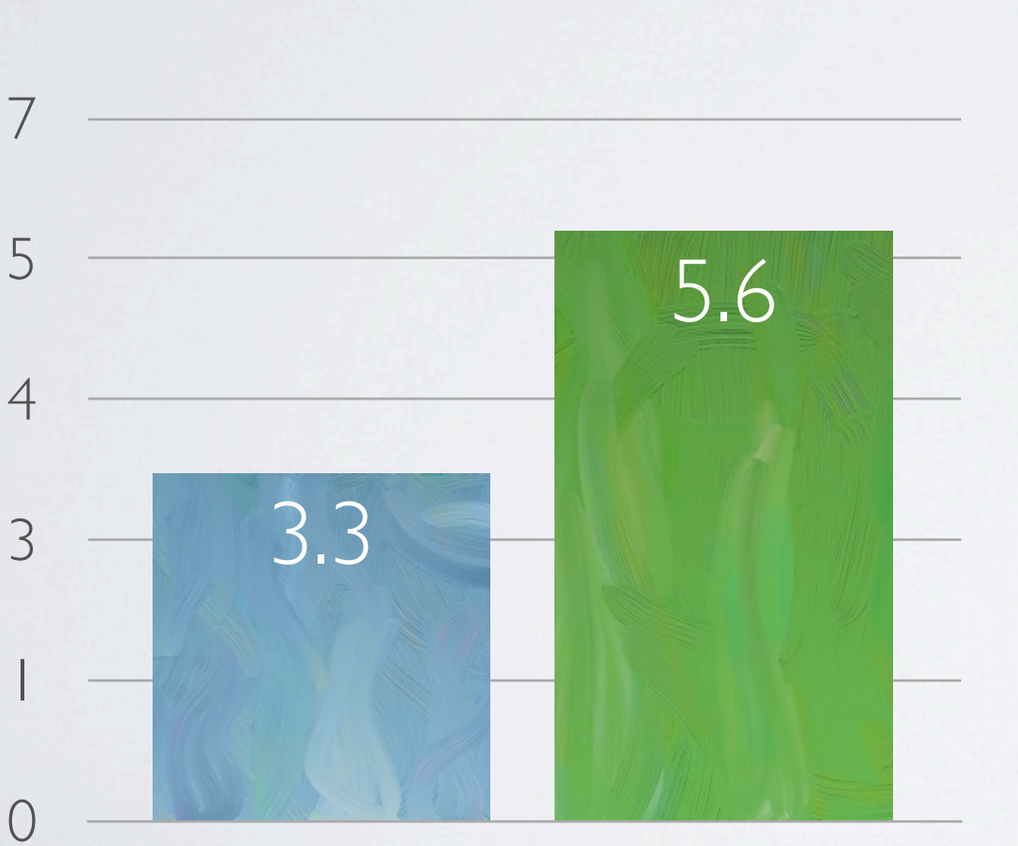
- The Kalman filter method is intended for finding the optimum estimation r of an unknown vector $\mathbf{r}^t$ according to the measurements m_k, $k=1...n$, of the vector $\mathbf{r}^t$.

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

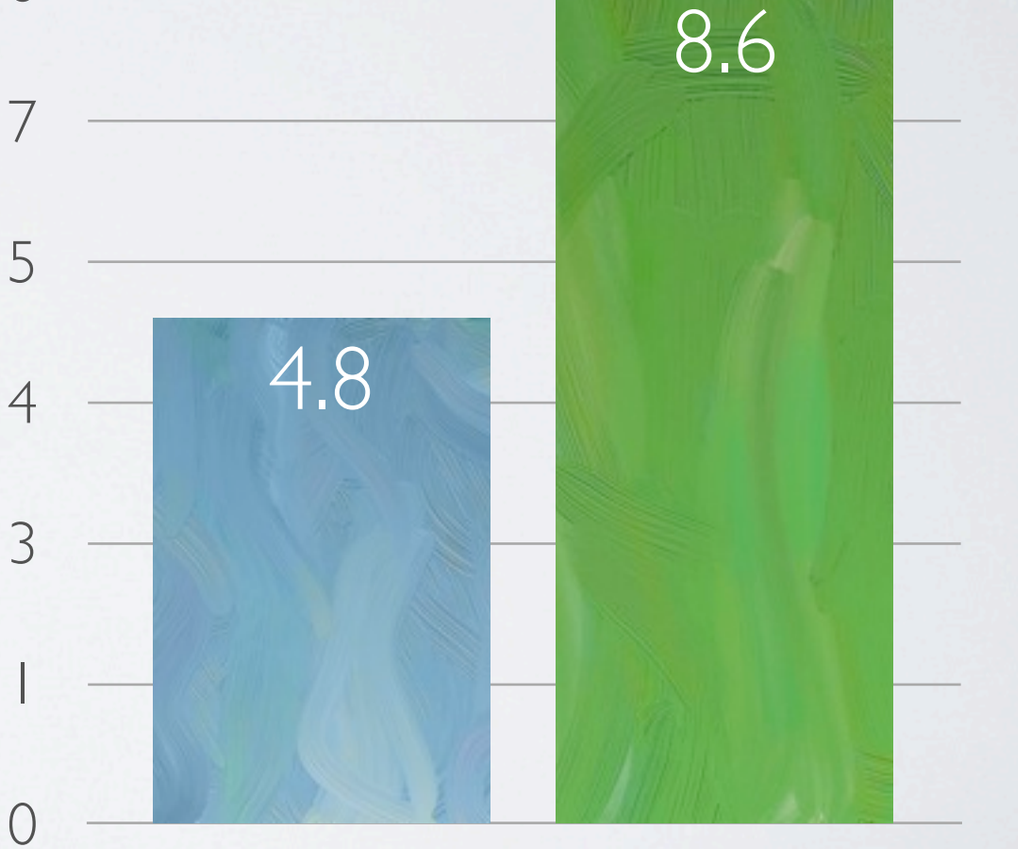
KALMAN FILTER

100 EVENTS, ~100 TRACKS WITH ~10 HITS EACH

ISPC vs GCC with AoS to SoA conversion
ISPC vs GCC assuming data is preconverted



KalmanFilter DP speedup (GCC 4.8 -O3)



KalmanFilter SP speedup (GCC 4.8 -O3)

ISPC: FINAL THOUGHTS

THE VECTORIZATION FINAL SOLUTION?

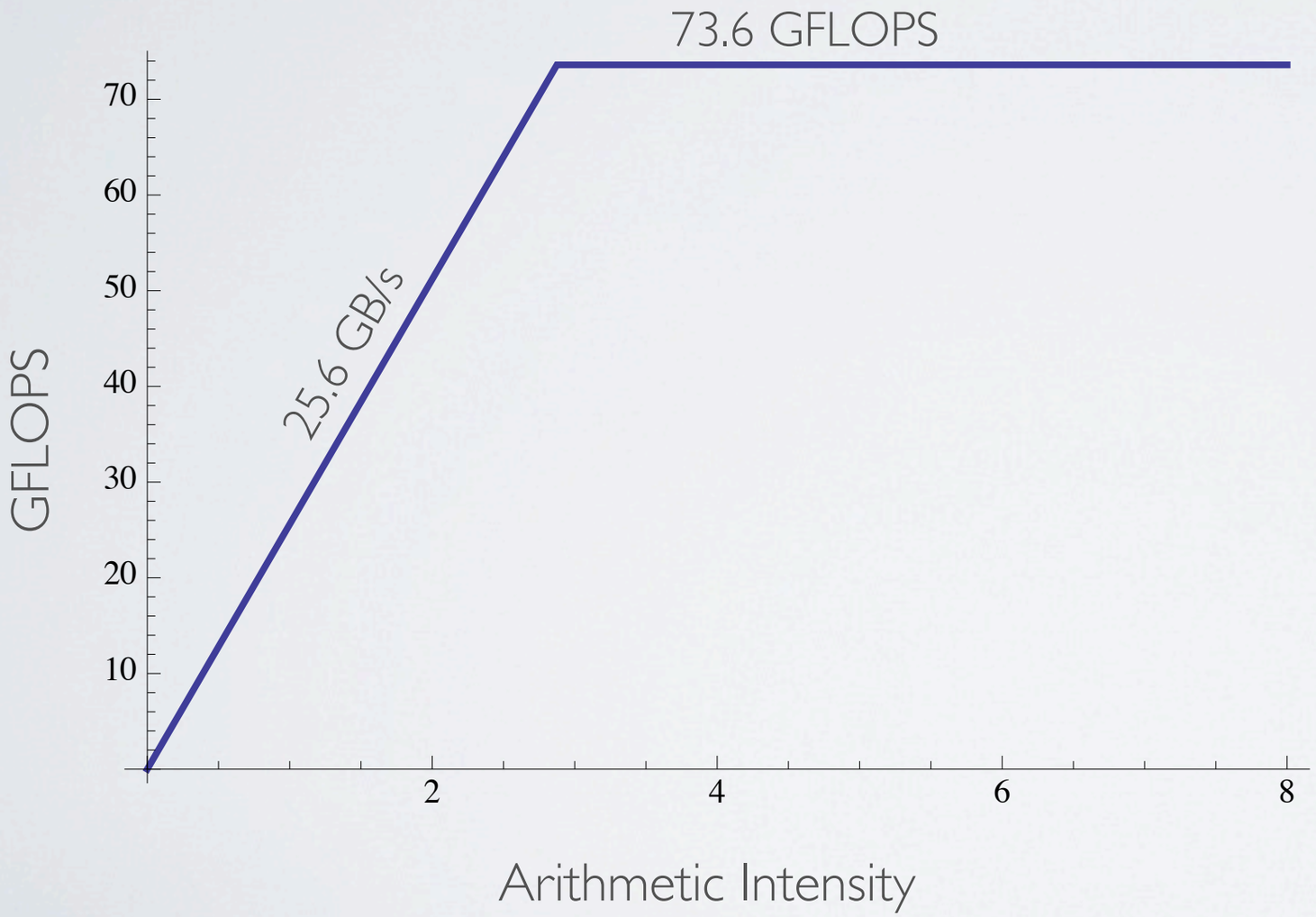
- ISPC is by far the best option I have seen to exploit vectorization
  - it gives the programmer an abstract machine model to reason about
  - it warns about inefficient memory accesses (aka gathers and scatters)
- In other words, it's not going to get much easier than that
  - full C++ support would be a welcome addition
- ISPC is stable, extremely well documented, open source and has a beta support for the Xeon PHI
- I don't have yet a comparison with OpenCL

LESSONS LEARNED

LESSONS LEARNED

WHICH PROBLEMS LEND THEMSELVES WELL TO VECTORIZATION?

Roofline model for my machine: **i7-3610QM**



- The roofline model sets an upper bound on the performance of a numerical kernel
- Performance is either bandwidth or computationally limited
- Only Kernels with an arithmetic intensity greater or equal ~3 can achieve the maximum performance

LESSONS LEARNED

MEMORY MATTERS

- 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 penalty is paid
  - Convert your arrays of structures to structures of arrays
- 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

LESSONS LEARNED

CONCLUSIONS

- Use ISPC or an equivalent tool whenever possible
  - It provides the right abstraction to exploit horizontal vectorization
- Use Eigen3 for small Matrix and Geometry calculations
  - is a complete linear algebra library
  - significant speedups can be achieved through vectorization and template expressions
- Use a SIMD vector wrapper library to hand tune numerical hotspots which lend themselves only to vertical vectorization
- What about autovectorization?
  - not trivial to exploit: generally you need to know what you are doing to get some speedup
  - small changes in the code might silently break carefully optimized code
  - avoid whenever you want to achieve maximum performances