# INVESTIGATIONS OF VECTOR AND LINEAR ALGEBRA LIBRARIES

Joe Boudreau (University of Pittsburgh)
Roberto A. Vitillo (LBNL)

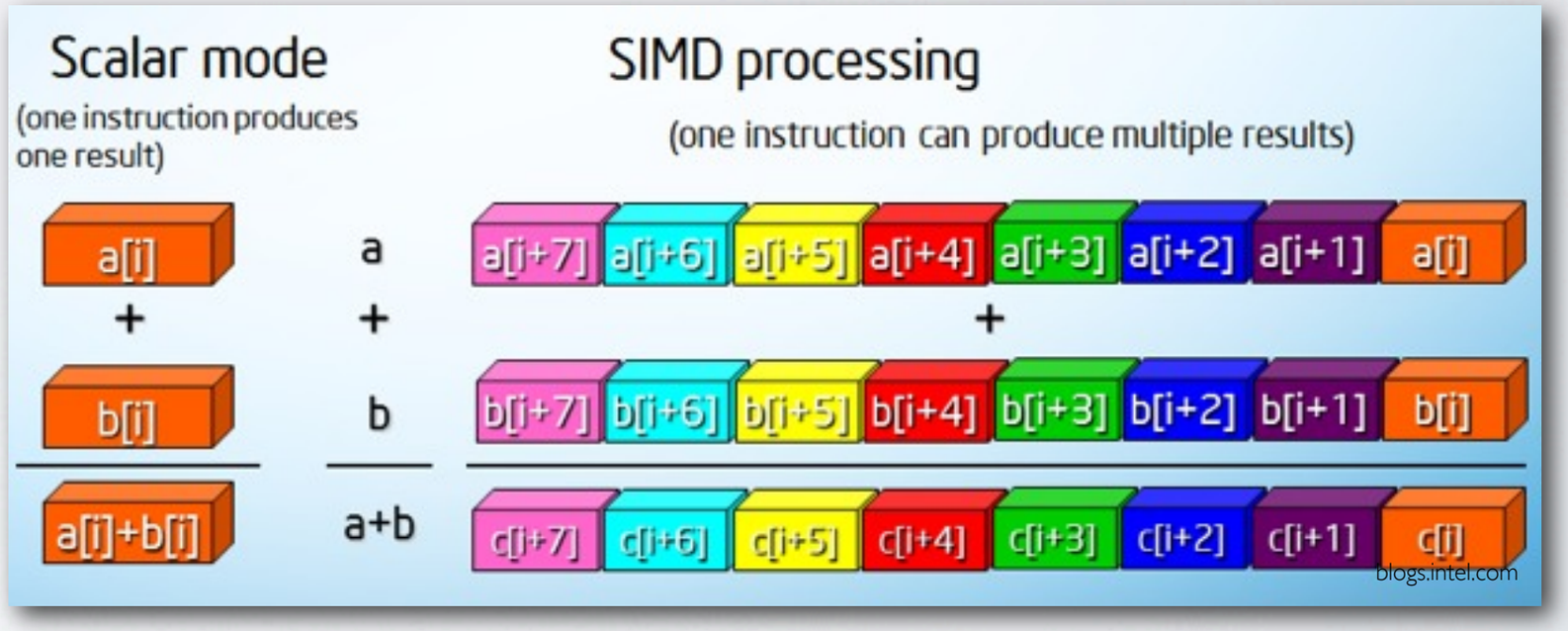
February 2013, Concurrency WS, Chicago

# ATLAS REQUIREMENTS

- Linear algebra operations
  - symmetric matrices (3x3, 4x4, 5x5) for transforms
  - NxM matrices (e.g. 2x5, 3x5, 5x2) for error propagation
  - separate use of 25x25 matrix in a few places
- Transcendental functions (sin, cos, ...)
- Abstractions for hardware vectors (SSE, AVX, IMCI)

# EXPLOITING SIMD

- Concept:
  - **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
- How?
  - intrinsic functions / assembly
  - vector libraries
  - linear algebra libraries
  - autovectorization (see Backup)
  - language extensions (see Backup)
- Where should we use it?
  - linear algebra libraries
  - transcendental libraries
  - numerical hotspots identified by a profiler, e.g. GOoDA



# WHAT ABOUT THE XEON PHI?

- Xeon PHI is based on a modified Pentium processor
  - supports x87 instruction set
  - doesn't support MMX, SSE* nor AVX
  - provides its own vector instruction set
  - 512 bit registers for SIMD operations

- Theoretical throughput can **only** be achieved if the wide vector units are exploited!
  - but what is going to happen then with our branchy code?
  - subject of another talk

APPROACH 1

# HAND TUNING NUMERICAL HOTSPOTS

# HANDTUNING

- GOoDA report of an high pile-up job
  http://annwm.lbl.gov/~vitillo/visualizer/#report=reports/pileup

- Most cycles spent in *RungeKuttaPropagator::rungeKuttaStep*

- Most nested loop accounts for ~50% of its cycles

  - contains lots of floating point operations

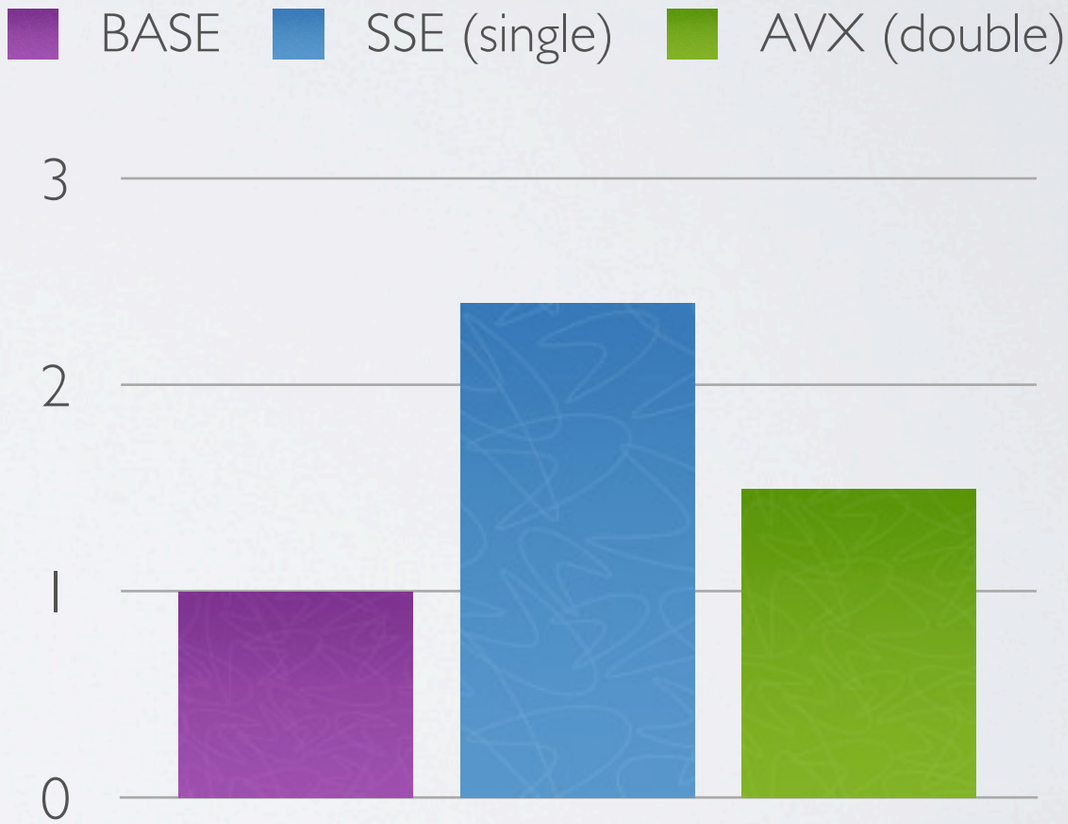
- Good candidate for vectorization

  - autovectorization fails

```
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

- Tested on a Sandy Bridge-EP CPU
- SSE (single) version: 2.4x faster
- AVX (double) version: 1.5x faster
  - slower than SSE because of costly cross lane permutations
  - not as mature as SSE
  - AVX2 (Haswell) will change that



# HANDTUNING

- Hand-vectorizing may be suitable when maximum speed-up is required and/or other approaches fail
- Options:
  - Inline Assembly
    http://www.ibm.com/developerworks/library/l-ia/index.html
  - Compiler Intrinsics
    http://en.wikipedia.org/wiki/Intrinsic_function
  - C++ Vector Class Library
    http://www.agner.org/optimize/vectorclass.zip
  - VC Library
    http://code.compeng.uni-frankfurt.de/projects/vc

# C++ VECTOR CLASS LIBRARY

- Exposes fixed-sizes vectors and overloaded operations
- Types for vectors of single and double precision floating point numbers
  - total vector size: 128 or 256 bits

```
float a[8], b[8], c[8];
Vec8f avec, bvec, cvec;

avec.load(a);
bvec.load(b);

avec = bvec + cvec * 1.5f;

cvec.store(c);
```

- Implements vectors with SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, XOP, FMA3, FMA4.
  - implementation is chosen at compile-time
  - header only library
- Can use AMD's libm or Intel's SVML to implement transcendental functions
- Well suited for vertical vectorization

# VC LIBRARY

- Exposes vectors classes that abstract the SIMD registers
  - memory abstraction allows to handle uniformly arrays of any size
  - masked ops syntax

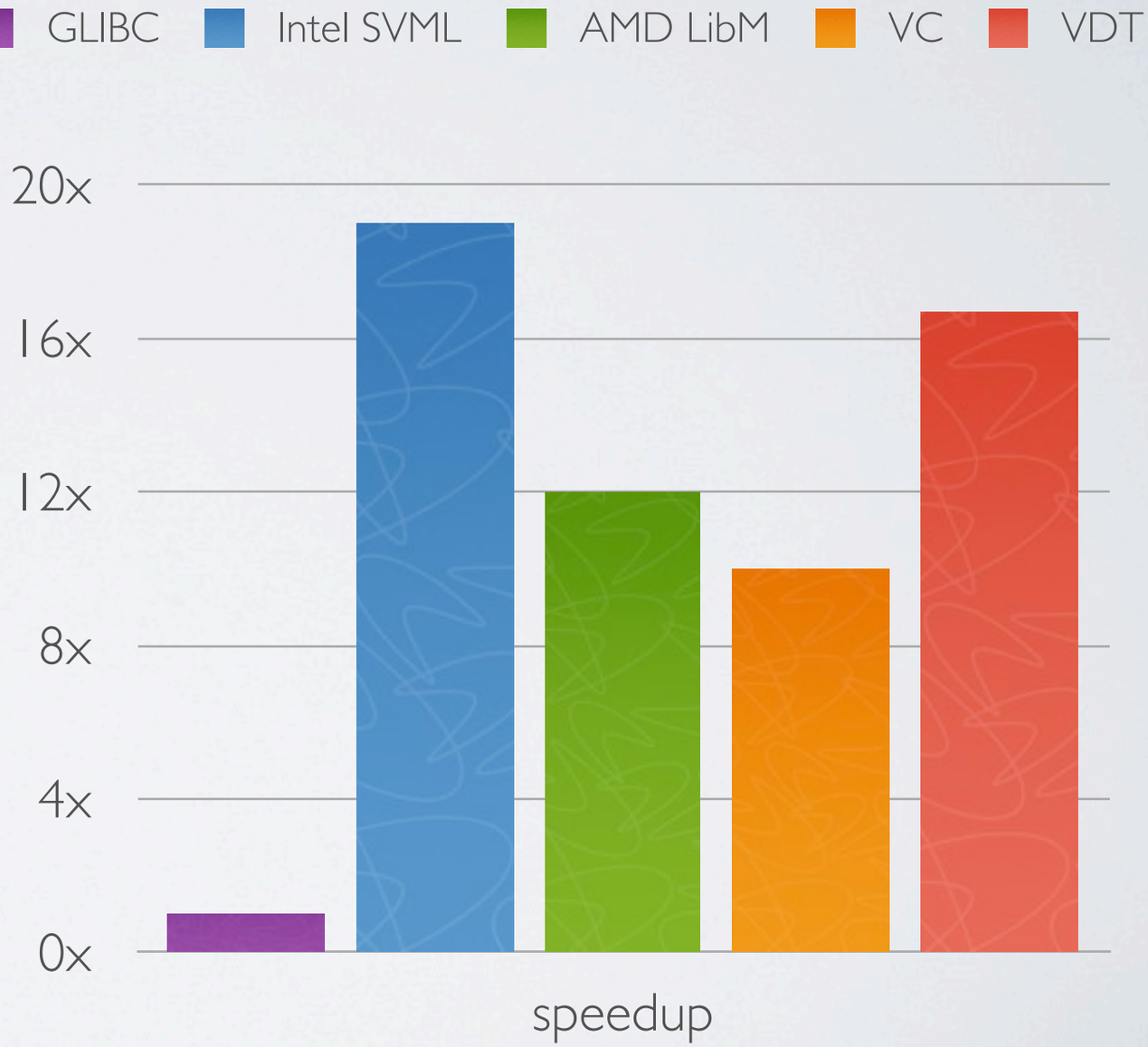
```
void testVc(){
  Vc::Memory<double_v, SIZE> x;
  Vc::Memory<double_v, SIZE> y;

  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

| 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 1000 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



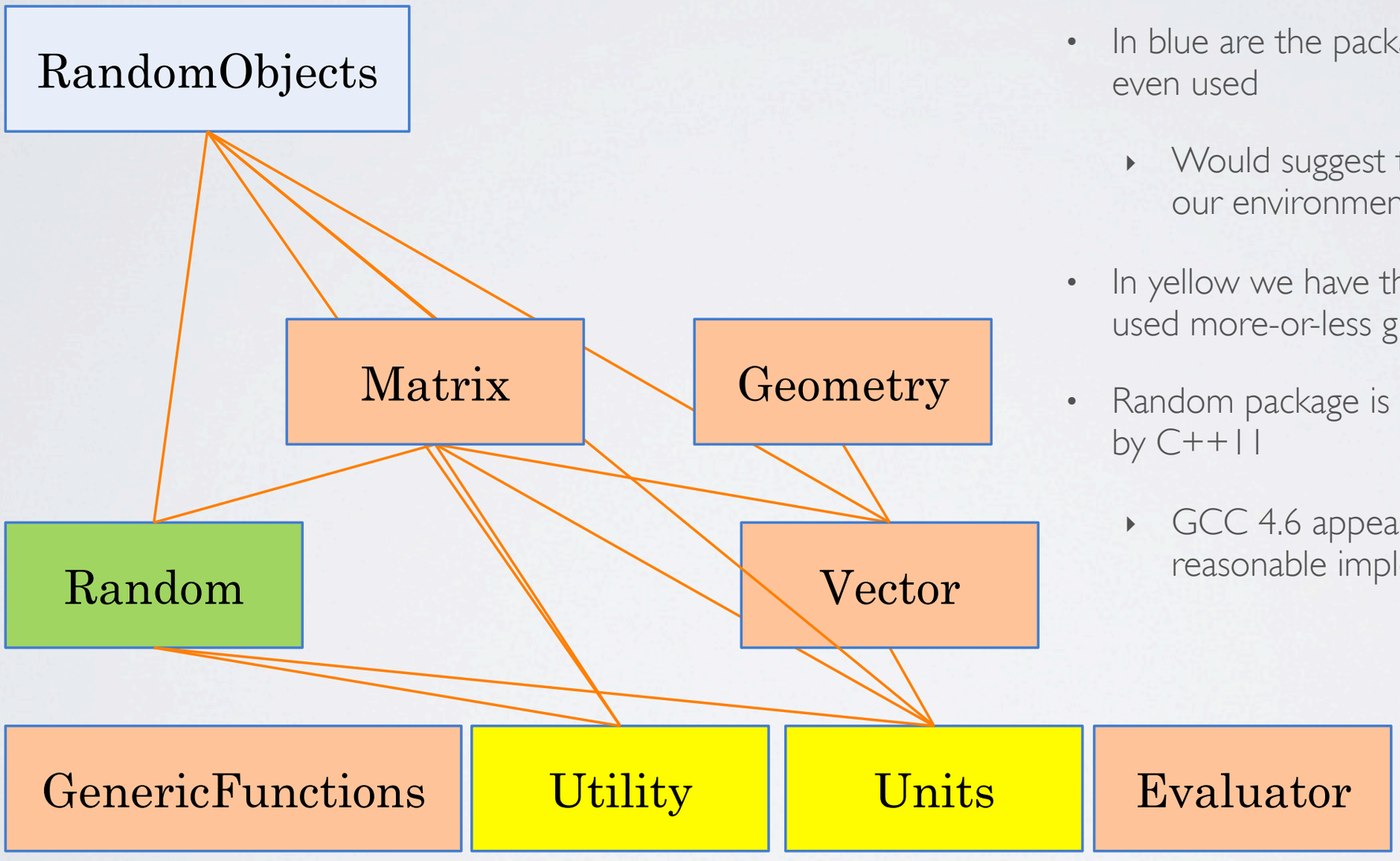

similar results with GCC 4.7.2 and ICC 13.0.1 on an Ivy Bridge

# APPROACH 2

# LINEAR ALGEBRA LIBRARIES

# CLHEP

## CRITIQUE & ANALYSIS

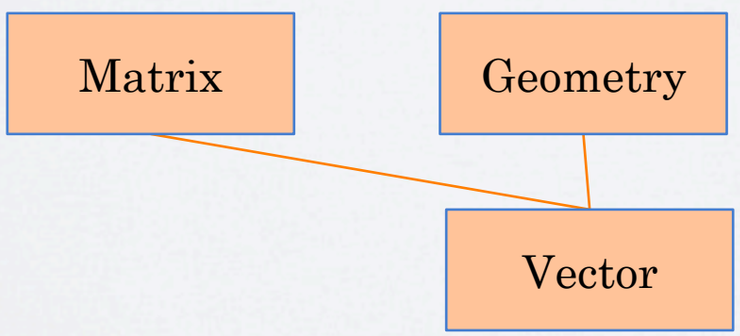



- In blue are the packages which are not even used
  - Would suggest to drop them from our environment
- In yellow we have the packages that are used more-or-less globally
- Random package is rendered obsolete by C++11
  - GCC 4.6 appears to have a reasonable implementation

# CLHEP

## HERE IS WHAT WE NEED TO OPTIMIZE

- Matrix is the class where a large amount of CPU is going and which needs to be optimized and vectorized
- Note: Vector and Geometry are exposed in very many places to end-users. Matrix which implements linear algebra, is used to perform specialized calculations and eventual replacements can be considered even if they do not "drop-in"...



# CLHEP

## STILL SOME EASY CHANGES TO MAKE

```
double Transform3D::operator () (int i, int j)
const {
    if (i == 0) {
        if (j == 0) { return xx_; }
        if (j == 1) { return xy_; }
        if (j == 2) { return xz_; }
        if (j == 3) { return dx_; }
    } else if (i == 1) {
        if (j == 0) { return yx_; }
        if (j == 1) { return yy_; }
        if (j == 2) { return yz_; }
        if (j == 3) { return dy_; }
    } else if (i == 2) {
        if (j == 0) { return zx_; }
        if (j == 1) { return zy_; }
        if (j == 2) { return zz_; }
        if (j == 3) { return dz_; }
    } else if (i == 3) {
        if (j == 0) { return 0.0; }
        if (j == 1) { return 0.0; }
        if (j == 2) { return 0.0; }
        if (j == 3) { return 1.0; }
    }
    std::cerr << "Transform3D subscripting: bad
indexes "
    << "(" << i << "," << j << ")" <<
std::endl;
    return 0.0;
}
```

- The worse offenders are accessors and they are not inlined!
- The accessor for HepTransform3D has a lot of tests
- We are starting to test the gain from inlining these functions...

# CLHEP

REVIEW

- A close look at CLHEP reveals that the focus falls on very few sub-packages: Matrix, Geometry, Vector
- While stability is a good thing, we can retool before Pase 0 and we should take the opportunity to rationalize several aspects of CLHEP
- Simple changes (inlining) to Vector and Geometry packages will improve the CPU
- Further changes (homogenous transformations) to the implementation and not the the interface will bring additional CPU performance and vectorizability
- Need to bring the editors and maintainers together again

# EIGEN 3

- C++ template library for linear algebra
- Matrices, vectors, numerical solvers and related algorithms
- Different codepaths for big matrices (dim > 8) and small matrices
  - only dim4 and dim8 are vectorized
- Supports SIMD
- Supports expression templates which allow to remove temporaries and enable lazy evaluation

```
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;
using namespace std;

int main()
{
    Matrix3d m = Matrix3d::Random();
    m = (m + Matrix3d::Constant(1.2)) * 50;
    cout << "m =" << endl << m << endl;
    Vector3d v(1,2,3);
    cout << "m * v =" << endl << m * v << endl;
}
```

# INTEL MKL

- BLAS: Basic Linear Algebra Subroutine
  - is a de facto API standard for basic linear algebra operations such as vector and matrix multiplication, e.g. for general matrix multiply:
    - DGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)
    - $C = \alpha AB + \beta C$
- LAPACK:  Linear Algebra PACKage
- MKL provides a vectorized BLAS and LAPACK implementation
- MKL supports C and Fortran

# SMATRIX

- Is a ROOT C++ package for high performance vector and matrix computations
- Provides generic Matrix and Vector classes of arbitrary dimensions and type
- Classes templated on the dimension and on the scalar type
  - header only library
  - exploits expression templates
- It's not a complete linear algebra package unlike Intel MKL or Eigen3
- Not vectorized

# MATRIX MULTIPLICATION

- 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];
   }
}
```

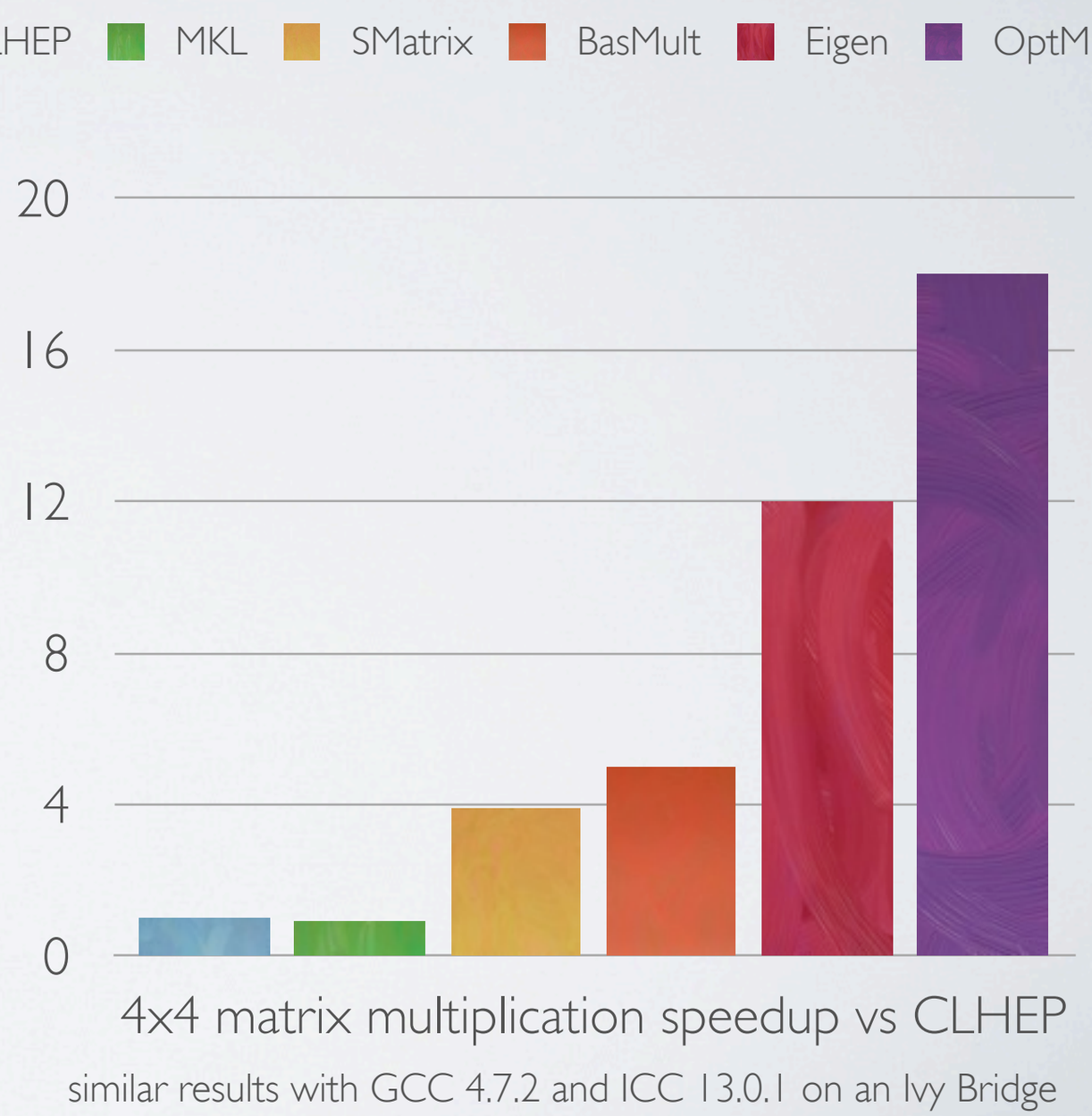
OptimizedMult

```
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

- Eigen 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 for the specific problem

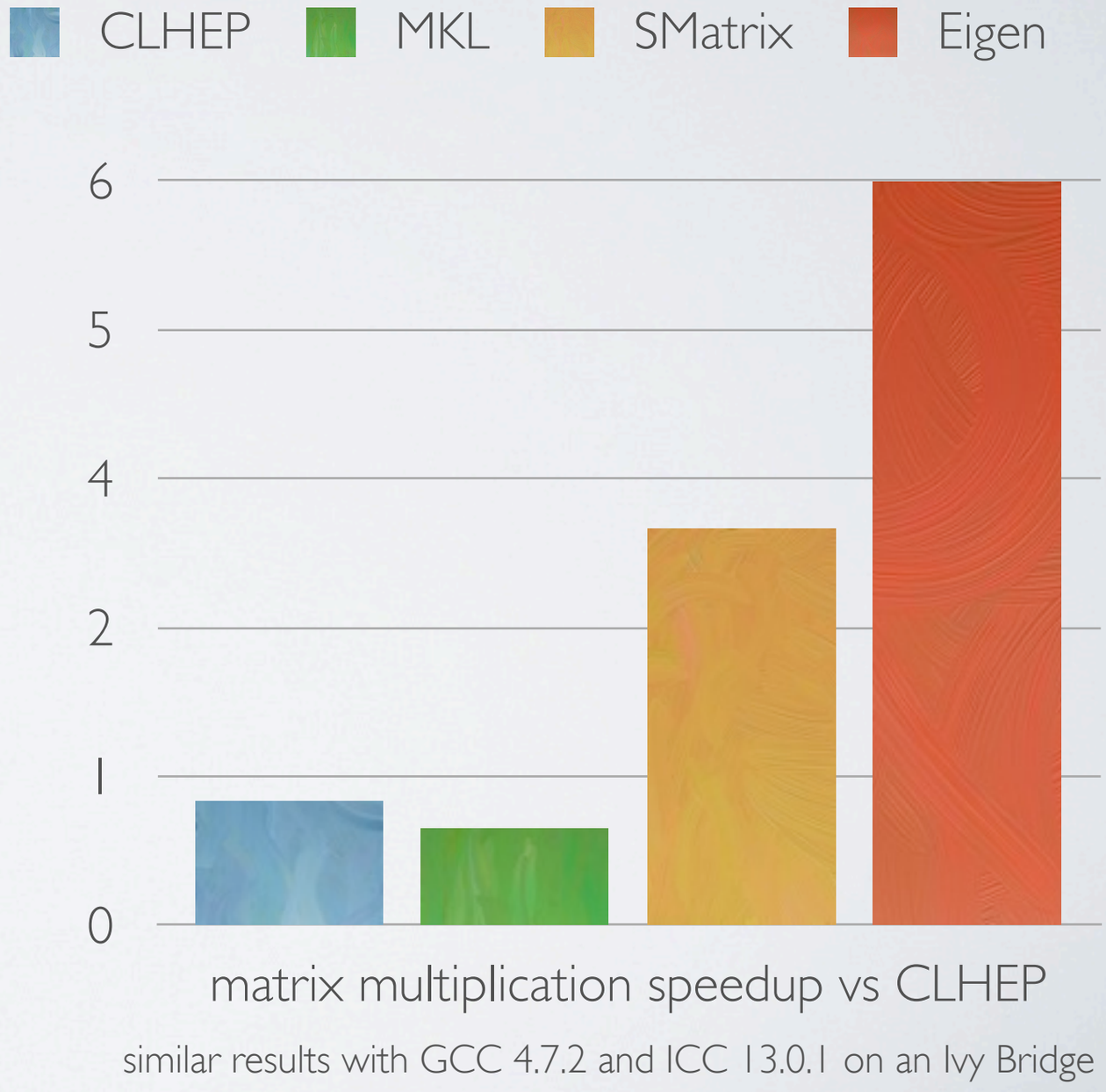



similar results with GCC 4.7.2 and ICC 13.0.1 on an Ivy Bridge

# MATRIX MULTIPLICATION

## RECTANGULAR MATRICES

- Evaluating $A_{5\times3} \times B_{3\times5}$
- None of the libraries is using vectorized code!
  - vectorization is not possible in this case without wasting memory

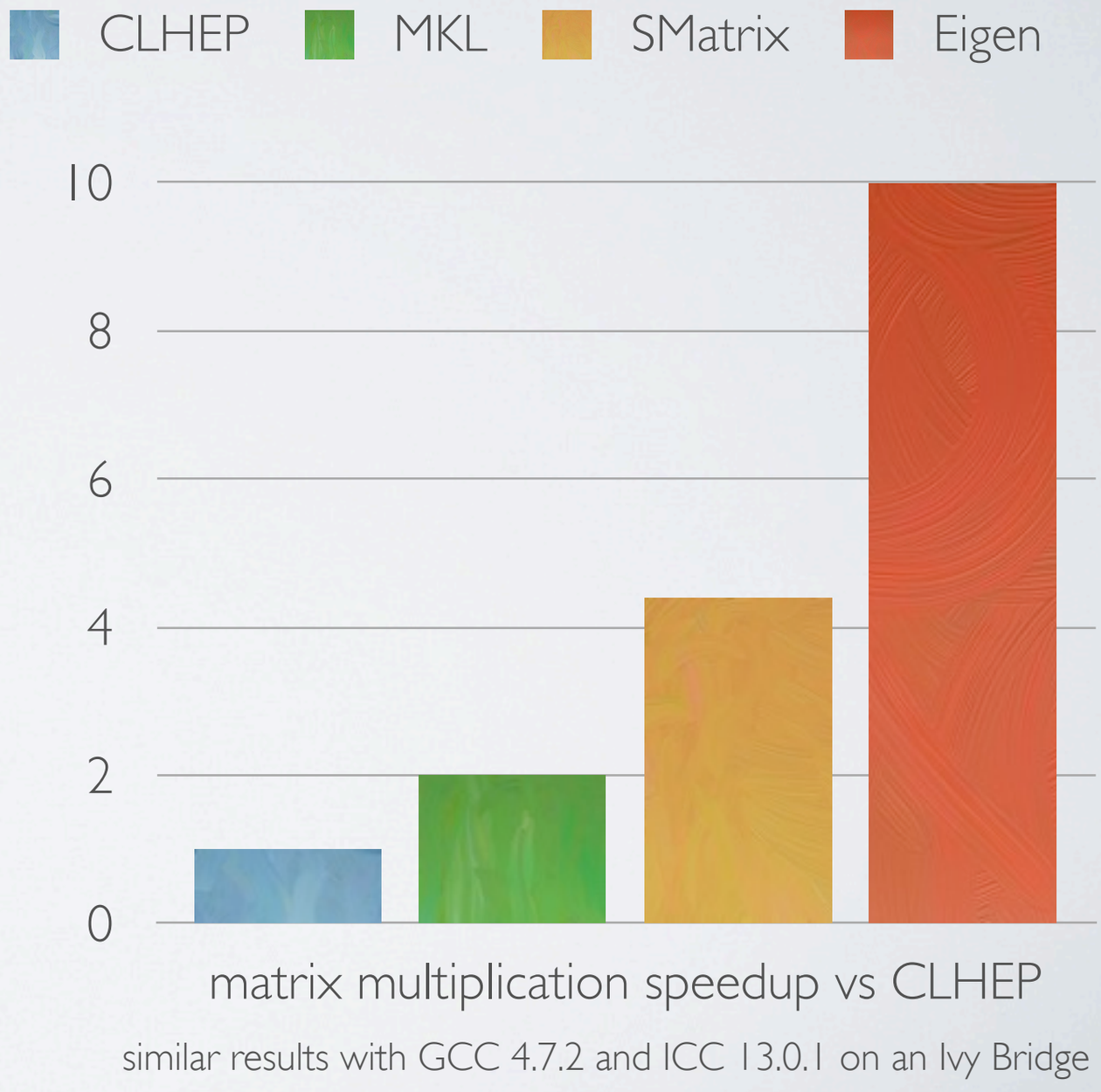



similar results with GCC 4.7.2 and ICC 13.0.1 on an Ivy Bridge

# MATRIX MULTIPLICATION

## EXPRESSION TEMPLATES

- Evaluating $C_{5\times5} = \alpha A_{5\times3} B_{3\times5} + \beta C_{5\times5}$
- MKL is still evaluating the same expression as before



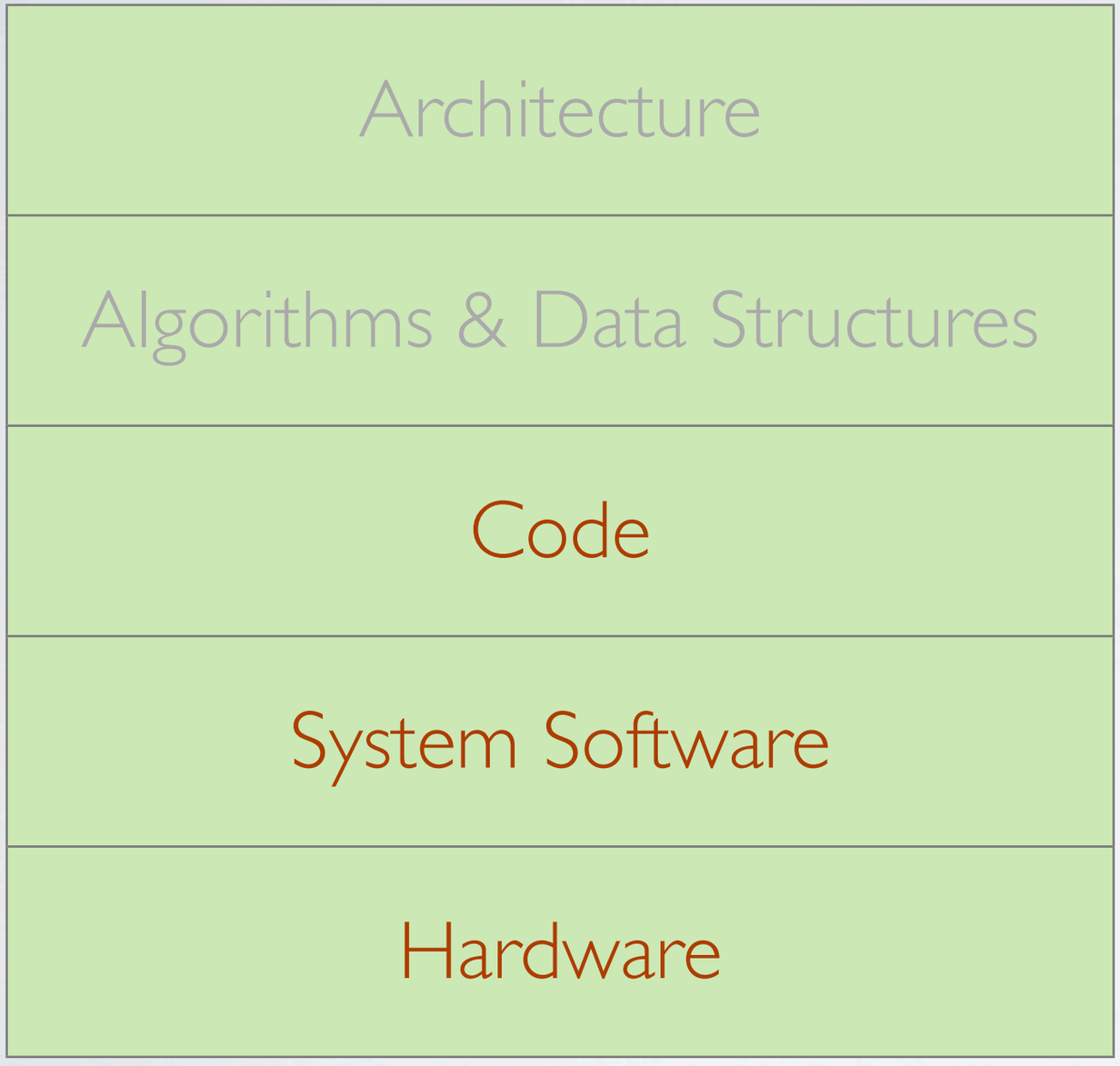

similar results with GCC 4.7.2 and ICC 13.0.1 on an Ivy Bridge

# CONCLUSIONS

- Use autovectorization when applicable
  - generally you need to know what you are doing to get some speedup (see Backup)
- Use Vc to handtune numerical hotspots but...
  - use SVML or VDT instead of Vc's native transcendental library
  - OpenMP4 (see Backup) may render this approach obsolete in a few years...
- Use Eigen3 for Matrix and Geometry
  - is a complete linear algebra library
  - significant speedups can be achieved through vectorization and template expressions in some cases
  - AVX support will come soon enough

# BACKUP

# DESIGN LEVELS




- Clock speed free lunch is over
  - hardware parallelism is increasing
- Independent changes on different levels causes speedups to multiply
- Rules of thumb:
  - If you need a small speedup, work at the best level
  - if you need a big speedup, work at all levels
- How can we exploit parallelism at the bottom layers?

# APPROACH 3

# AUTOVECTORIZATION

# LAR CALIBRATION CODE

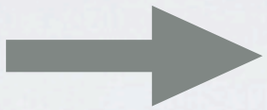
## WALTER LAMPL

- 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)
- Autovecorizing 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 (good enough for our purpose)

# AUTOVECTORIZATION CAVEATS

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

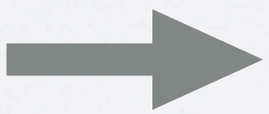
what we would like:



```
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
- If they do, scalar addition is performed
  - otherwise loop is only partially vectorized

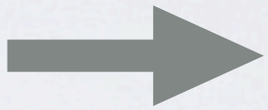
what we actually get:



```
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];
    }
}
```

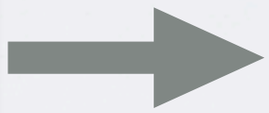


```
80:    push   rbp
81:    mov    r9,rdi
84:    and    r9d,0x1f
88:    mov    rbp,rsp
8b:    push   r12
8d:    shr    r9,0x3
91:    neg    r9
94:    push   rbx
95:    mov    edx,r9d
98:    and    rsp,0xfffffffffffffe0
9c:    add    rsp,0x20
a0:    and    edx,0x3
a3:    je     185 <foo1+0x105>
a9:    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
  - 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 in general the most efficient vector code
- Fore more information: http://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

# APPROACH 4

# LANGUAGE EXTENSIONS

# CILK PLUS

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

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

- 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*
- Is being ported to GCC but not yet included in the main branch
- Offers other features not related to vectorization, see: http://software.intel.com/en-us/intel-cilk-plus

# OPENMP 4

```
#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]));
    }
}
```

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