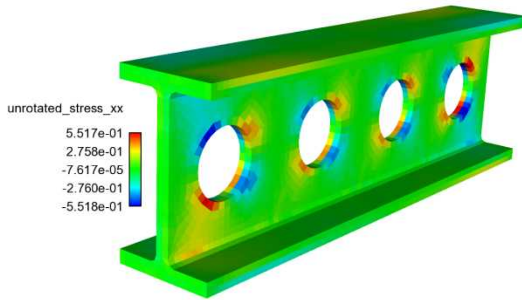
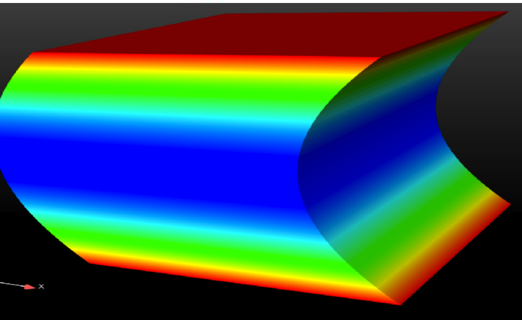
SAND2015-0492PE

Sierra's SIMD vector-math library for element, tensor and material calculations





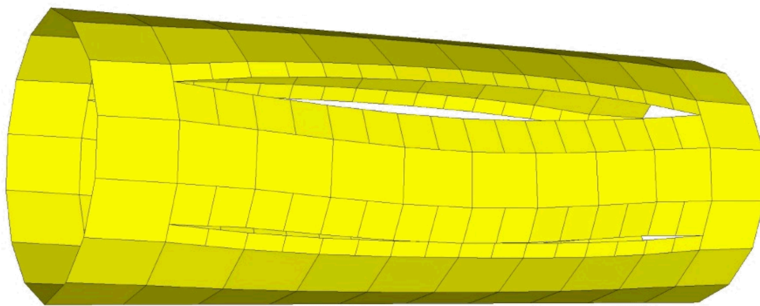
JOWOG 34 Applied Computer Science Meeting

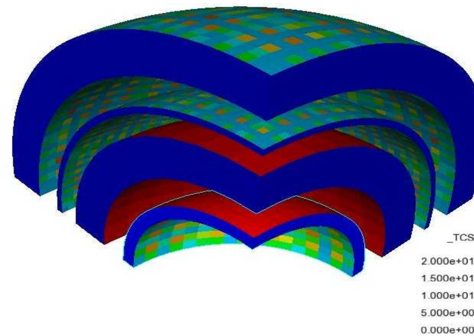
Michael Tupek

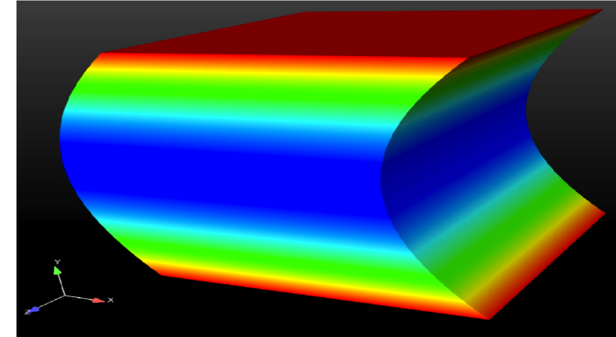
Exceptional service in the national interest

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

SIERRA/SM (Solid Mechanics)





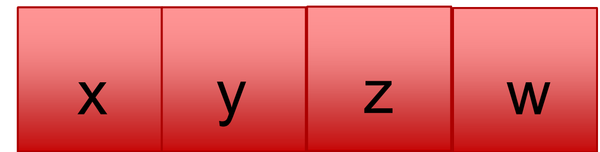


- A general purpose massively parallel nonlinear solid mechanics finite element code for explicit transient dynamics, implicit transient dynamics and quasi-statics analysis.
- Built upon extensive **material**, **element**, contact and solver libraries for analyzing challenging nonlinear mechanics problems for normal, abnormal, and hostile environments.
- Similar to LSDyna or Abaqus commercial software systems.

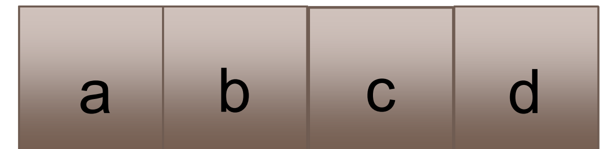
What is SIMD?

Single Instruction, Multiple Data

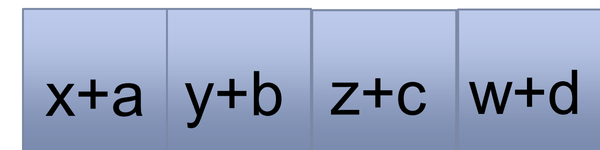
- SSE2 instructions: (Intel, AMD ~2004)
  - 2 doubles, 4 floats
- AVX instructions (Intel, AMD)
  - 4 doubles, 8 floats
- AVX-512 instructions (Intel ~2014)
  - 8 doubles, 16 floats
- AltiVec (IBM)
- GPU (eg. CUDA): (Nvidia)
  - ~16/32 doubles or floats



+

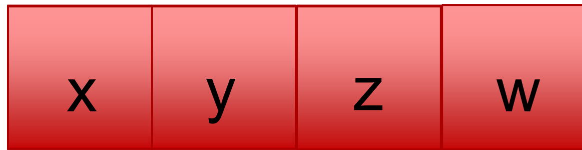


=

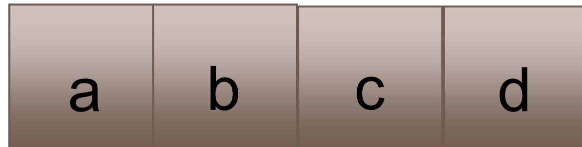


SSE2/AVX/AVX512 SIMD in Sierra-SM for nonlinear element assembly

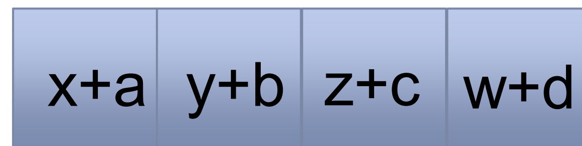
Sandia National Laboratories



+



=



For simple loops, compilers can auto-vectorize:

```
for (int i=0; i < N; ++i) {
    a[i] = b[i] + c[i] * d[i];
}
```

Complicated loops don't auto-vectorize:

- Tensor33 multiply
- Eigenvectors
- Constitutive law evaluations

Auto-vectorization

Sandia National Laboratories

- For simple loops, compilers with optimizations on automatically use SIMD:

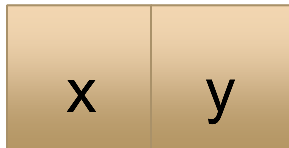
```
for (int i=0; i < N; ++i) {
  a[i] = b[i] + c[i] * d[i];
}
```

- "Complicated" loops are not yet auto-vectorized efficiently:
  - Eigenvectors
  - Constitutive law evaluations
- Use SIMD vector intrinsics (low level functions):
  - Each intrinsic is equivalent to an assembly instruction

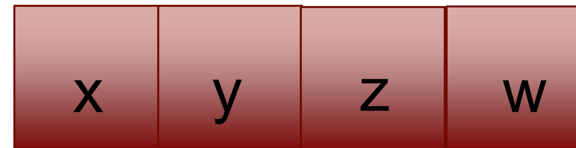
SSE2/AVX intrinsics (Intel, AMD)

Sandia National Laboratories

__m128d (2 doubles)

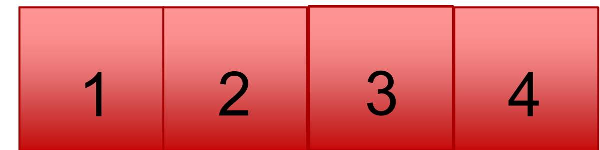


__m256d (4 doubles)

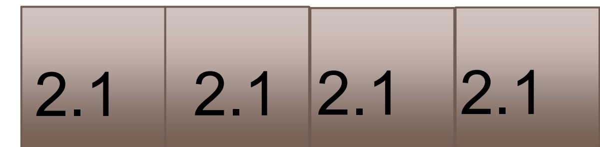


Compute {1,2,3,4} + 2.1:

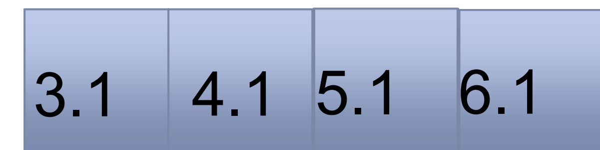
```
double x[4] = {1,2,3,4};
__m256d a = _m256_loadu_pd(x);
__m256d b = _m256_set1_pd(2.1);
__m256d c = _m256_add_pd(a,b);
double result[4];
_m256_store_pd(result,c);
```



+



=



Sierra SSE2/AVX interface

Sandia National Laboratories

- Developers can't know which instruction set is available, as it differs by processor generation:
  - Chama (with Intel Sandy Bridge) has AVX
  - Other Sandia machines only have SSE2 (or SSE4)
- Want to be able to write code which works for SSE2, AVX and even future AVX-512
- We provide an abstraction layer to simplify development

Sierra SSE2/AVX/AVX512 interface

Sandia National Laboratories

Simd.h:

```
#if defined(AVX)
    const int ndoubles = 4;
    class Doubles { __m256d d };
#elif defined(SSE2)
    const int ndoubles = 2;
    class Doubles { __m128d d };
#else
    const int ndoubles = 1;
    typedef double Doubles;
#end
```

main.cc:

```
#include <Simd.h>

double x[ndoubles];

Doubles a = simd::load(x);
Doubles b = Doubles(2.1);

// operator overload:
Doubles c = a+b;

double output[ndoubles];
simd::store(output,c);
```

Sandia National Laboratories

Sierra SSE2/AVX interface

- Difficult to have portable code:

  Doubles x = a+c/b;

  - Overloaded math operator only available with certain compilers (gcc, clang)
  - Wrapping SIMD type in a class creates some overhead
  - Expression templates slightly slower (and harder to read)
- Want to provide a library of math functions
  - sqrt, log, exp, pow, max, min, fabs, etc.
  - either not implemented or implemented only with certain compilers (intel)
- Current capabilities: **simd::sqrt(x), simd::log(x), simd::min(x,y)**.

SIMD "EDSL"

Standard math functions:

sqrt, cbrt, log, exp, pow, fabs, copysign, min, max

Simd boolean types:

<, <=, >, >=, == returns booleans, e.g.,

```
Bools isTrue = x < 5;
```

Simd ternary:

```
Doubles z = if_then(isTrue, 1.0, y);
```

Simd reduction:

```
double a = reduceSum(z);
```

Operator overloads:

+, -, *, /, +=, -=, *=, /=

Also Simd Loads and Store

Bottlenecks:

_mm256_sqrt_pd() is only ~2X faster than std::sqrt()

Same with _mm512_sqrt_pd()?

Some compilers don't implement cbrt, log, exp, etc.

Performance Improvements

Sandia National Laboratories

1,000,000 evaluations of random doubles:

c = (a+b)*(a-b)/a;

SSE2 (on blade, gcc)

- **1.83 x** speed up (memory access still a slight bottleneck)

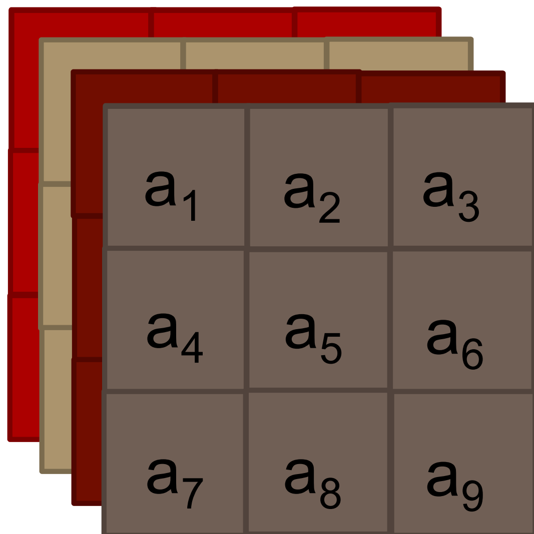
AVX (on chama, Intel compiler)

- **2.01 x** speed up (AVX often uses SSE for / and sqrt)

Auto-vectorization sometimes gives similar improvements, but…

- can't use sqrt()?, log(), exp() for all compilers (may work with Intel)
- doesn't work well for tensor operations/complicated data layouts.

SIMD Tensor class

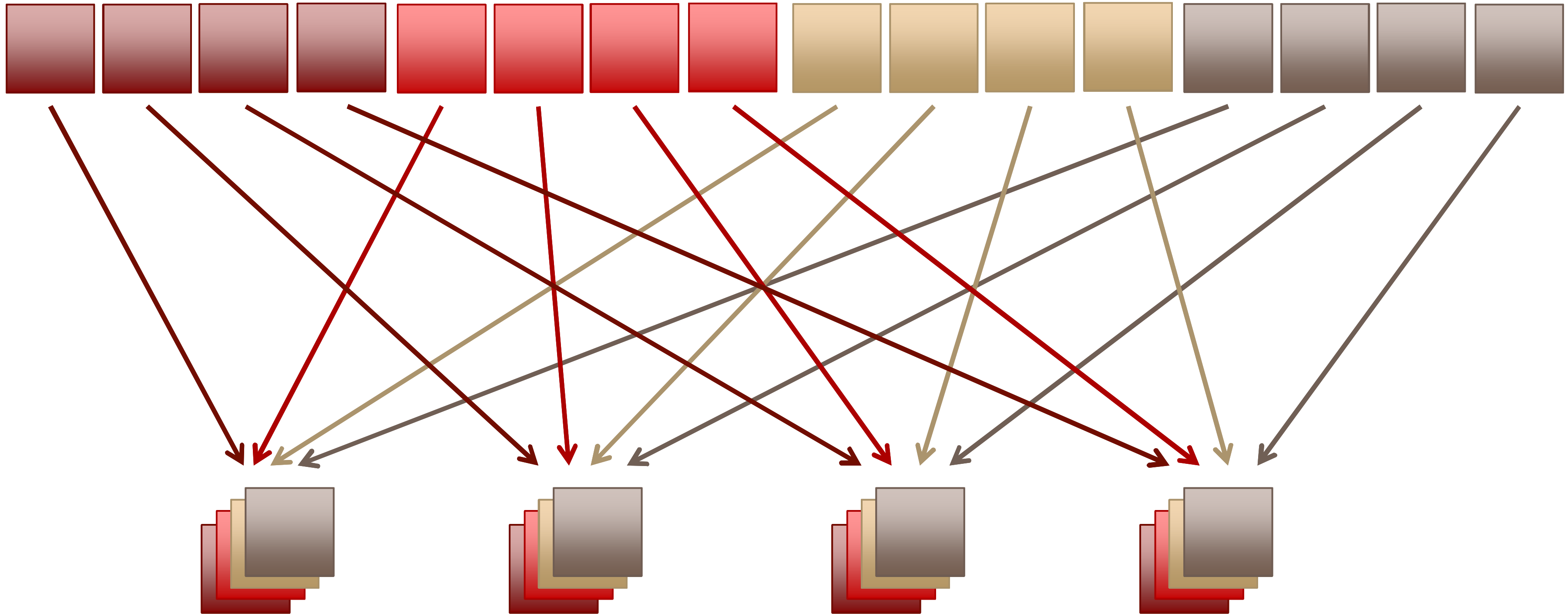


Process 4 tensors at a time (AVX):

```
double tensors[4*9];
// fill 4 tensor
Tensor33<Doubles> a(tensors)
c = mult(a,b);
Eigenvector(c,vects,vals);
c[0] = a[8] + b[4];
double output[4*9];
c.Store(output);
```

Loading 4 2x2 tensors

```
double a[4*tensor_size];
```



```
Doubles A[4];
for (int i=0; i < 4; ++i) A[i] = simd::load(a+i,tensor_size);
```

Slow memory access, but necessary unless we change memory layout of a.

Performance improvements

Sandia National Laboratories

	SSE2	AVX	AVX512(KNC)
Tensor multiply:	**1.80 x**	**3.63 x**	**2.42 x**
Eigenvalue:	**1.97 x**	**3.19 x**	**5.25 x**
Polar Decomp:	**1.7 x**	**2.28 x**	**4.89 x**

Performance Improvements: tensor operation which may not auto-vectorize

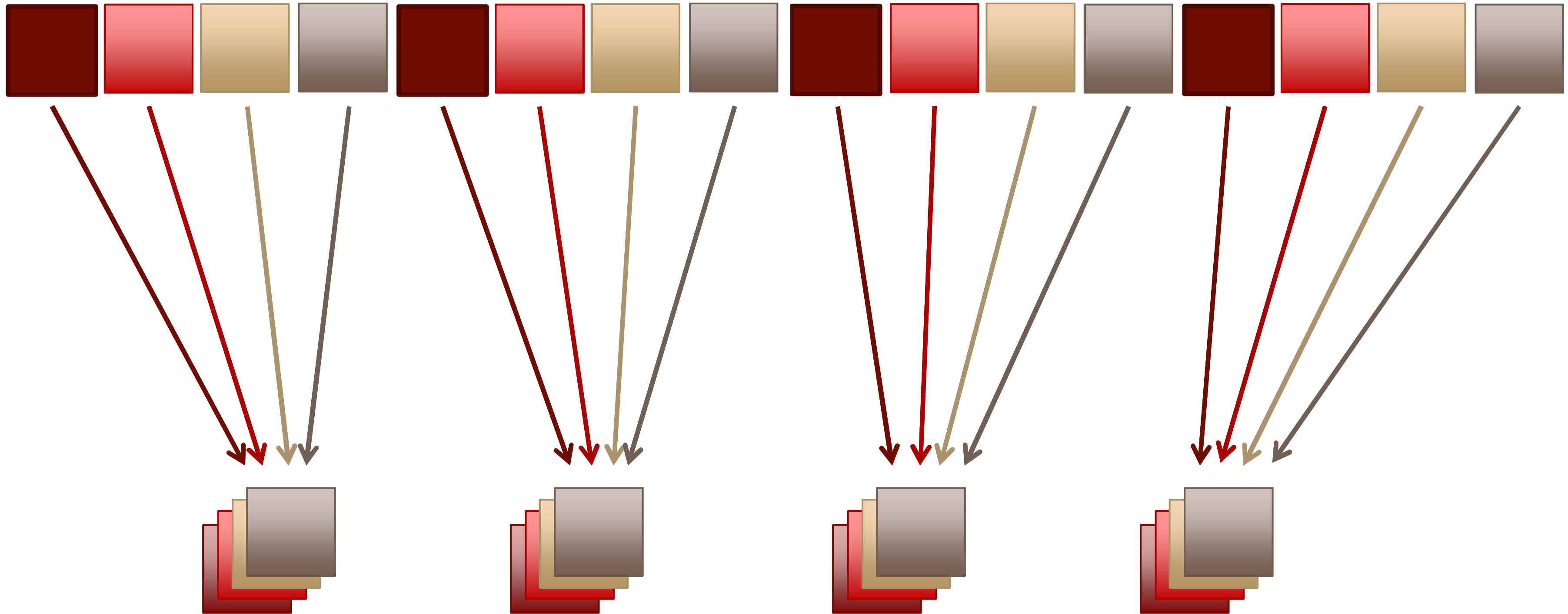
Sandia National Laboratories

- **SSE2 (on blade)**
  - Tensor multiply: **1.62 x**
  - Eigenvalue: **1.96 x**
  - Eigenvector: **1.61 x**
  - Polar Decomp: **1.56 x**
- **AVX (on chama)**
  - Tensor multiply: **3.35 x**
  - Eigenvalue: **2.90 x**
  - Eigenvector: **2.35 x**
  - Polar Decomp: **2.04 x**

Improved memory layout

Sandia National Laboratories

```
double a[4*tensor_size];
```



```
Doubles A[4];
for (int i=0; i < 4; ++i) A[i] = simd::load_better(a+i);
```

Fast memory access, but requires significant code refactoring.

Memory layout time comparison

Sandia National Laboratories

Time to load, add 1.0 and store 20,000
32 length vectors

Standard memory layout for array of vectors/tensors	**Improved layout**
2.11 ms	1.37 ms

1.54 X speedup

1.7 X if we use pointer casting instead of load/store …

Performance improvements using better memory layout

Sandia National Laboratories

- **SSE2 (on blade)**
  - Tensor multiply: **1.80 x**
    10% improvement
  - Eigenvalue: **1.97 x**
    3% improvement
  - Polar Decomp: **1.7 x**
    9% improvement

- **AVX (on chama)**
  - Tensor multiply: **3.63 x**
    7% improvement
  - Eigenvalue: **3.19 x**
    8% improvement
  - Polar Decomp: **2.28 x**
    9% improvement

Next Generation Material Models:

Vectorized hyper-elastic and J2 plasticity models

Sandia National Laboratories

- Re-implemented LAME material models
- Vector instruction require slight algorithm re-write to consider multiple material models running with the same instructions, e.g. :

```
Vec4<bool> isYielding = stress_trial > stress_yield;
if ( any (isYielding) ) {
 // compute plasticity model
 while ( any (notConverged) ) {
   // compute plasticity model sub-iterations
 }
}
// compute elasticity model
```

Speedups:

Neo-hookean: > **2 X faster with 4-wide SIMD**

J2 plasticity: **1.5 – 2 X faster**

Case Study: Branchless 3x3 Eigenvalue

Requires a variety of special functions:

- Doubles x = sqrt(y);
- Doubles x = min(y , z);
- Doubles x = cos(arccos(y)/3);
- copysign(): Doubles x = abs(a)*sign(b);
- ternary operator: Doubles x = (istrue ? a : b);

Branches handled carefully to avoid excessive expense

SIMD ternary operator

Sandia National Laboratories

- Really don't want to branch, but need:

```
x = ( y<z ) ? v : w;
```

```
Bools b = y < z;      // overloaded element-wise comparisons
Doubles x = simd::if_then(b,v,w);                // fake ternary operator
```

Implemented as:

```
__m128d istrue = _mm_cmplt_pd(y,z);       // returns (2) 0s or NaNs
__m128d t1 = _mm_and_pd(istrue,v);        // bitwise  istrue & v
__m128d t2 = _mm_andnot_pd(istrue,w); // bitwise !istrue & w
x = _mm_add_pd(t1,t2);
```

Done using bitwise operations, very fast, no branch!

SIMD copysign

Sandia National Laboratories

- Doubles x = copysign(y,z); // fabs(y)*sign(z)
- sign_mask = _mm_set1_pd(-0.0);
- sign_z = _mm_and_pd(sign_mask , z);
- fabs_y = _mm_andnot_pd(sign_mask,y);
- copysign(y,z) = _mm_xor_pd(sign_z,fabs_y);

This is all bitwise magic, but encapsulated so users don't have to know implementation details.

Sandia National Laboratories

Fast approximation for 3x3 eigenvalues

- Analytic eigenvalue calculations require evaluating:

$$\cos(\arccos(x)/3)$$

- A Padé approximation can be derived (in Mathematica):

$$\cos(\arccos(x)/3) \approx \frac{0.866 + 2.13x + 1.89x^2 + 0.74x^3 + 0.12x^4 + 0.0066x^5}{1 + 2.26x + 1.8x^2 + 0.6x^3 + 0.078x^4 + 0.0027x^5}$$

- Error < $5.6e^{-16}$ over entire range
- >7 X speed up over native C++ trig functions
- With AVX: speed up > 14 X

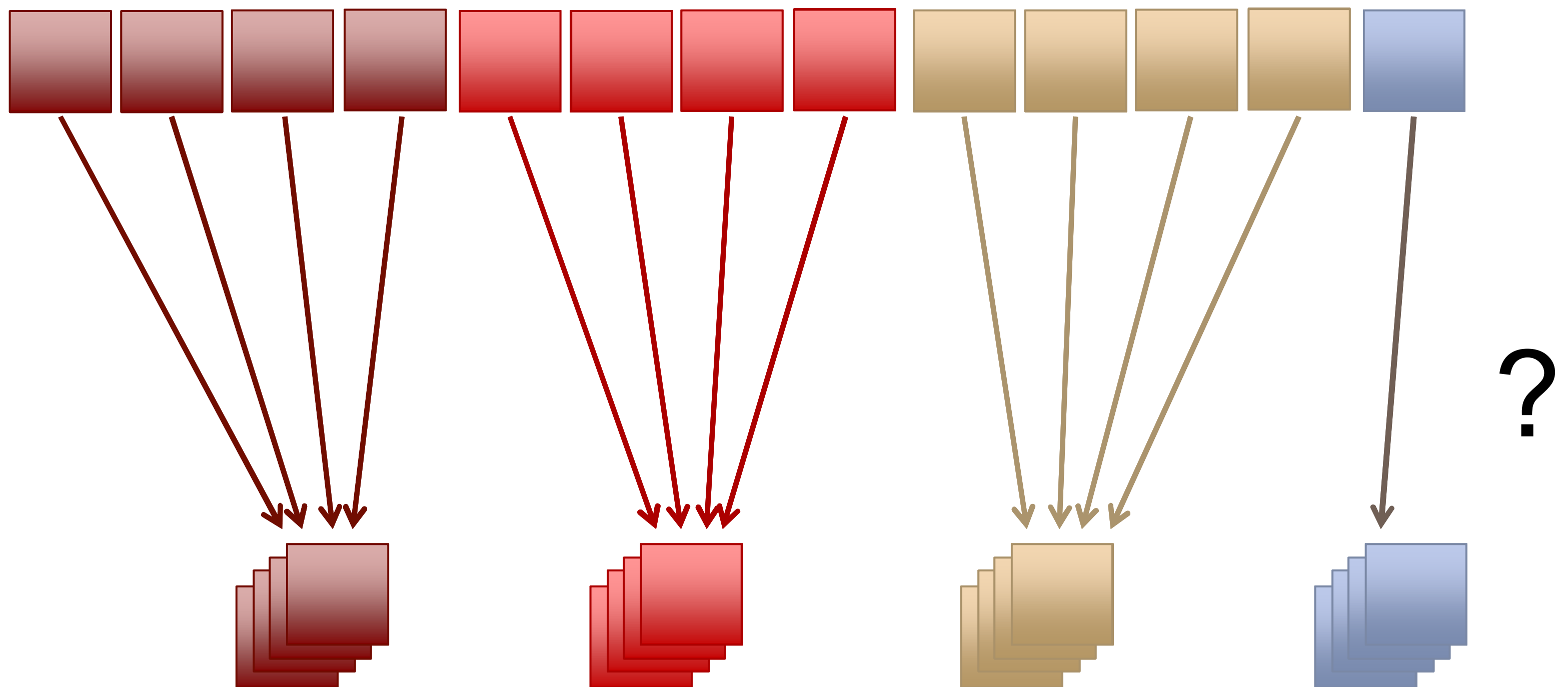
Additional functionality

```
Doubles x = 2.0;                // load 4 2.0s
Doubles y = 2.0*x + 3.0;        // may eventually use fused mult-adds
Doubles z = -y;
Doubles z /= x;
Bools d = a && (!b) || c;       // logical comparisons and negations
```

aligned load/store and **partial load/store** …

Code example: handling the remainder

- Suppose input is a flat array: x[nelems];
- What if nelems is not divisible by 4? Need to handle the remainder elements…



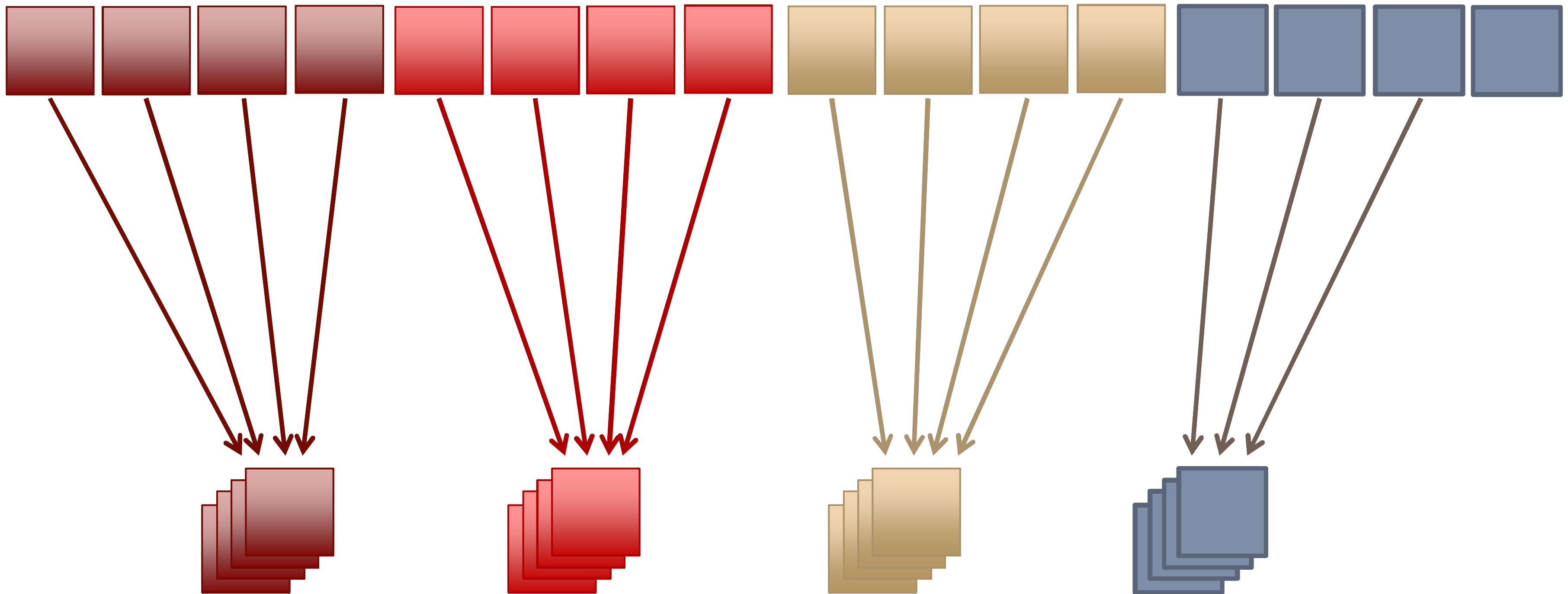
Code example: handling the remainder

Sandia National Laboratories

```
for (int e=0; e < nelems; e+=simd::nelements)  {
    Doubles x_loc;
    if (e + simd::ndoubles  <= nelems) {                  // load all 4 doubles
        x_loc = simd::load(x+e);
     } else {
            x_loc = simd:: loadPart(x+e, nelems-e);       // load remainder
    }
     Doubles x_squared = x_loc*x_loc;
     if (e + simd::ndoubles  <= nelems) {                 // store all 4 doubles
        simd::store(x+e,x_squared);
     } else {
            simd::storePart(x+e,x_squared, nelems-e);     // store remainder
     }
}
```

New approach: cast directly to Doubles*

- Over allocate arrays to multiple of ndoubles
- 32 bit aligned memory allocator for vector<>
- Doubles* X = reinterpret_cast(double* x);



Code example: casting to Doubles*

Sandia National Laboratories

```
Doubles* x_loc    = SimdPtrCast(x);        // x is array of '3 double'
Doubles* dot_loc = SimdPtrCast(dot);      // dot is array of double

for ( int e=0; e < nelems; e+=simd::nelements )  {
        dot_loc[0] = 0.0;
        for ( int i=0; i < 3; ++i )  {
       dot_loc[0] += x_loc[i] * x_loc[i];
    }
        x_loc += 3;
        dot_loc += 1;
}
```

1.5 X improvement over original layout/load strategy

Remaining challenges/ideas

Sandia National Laboratories

- Getting full AVX performance (4X speed up)
- Optimal memory layout, is there a better one?
- Efficient implements for math functions:
  - exp(), pow(), cbrt(), cos(), etc.
- Propagate though time-critical parts of Sierra
  - Can it be used effectively in search / contact?
  - Evaluating shape-functions?
  - Matrix assembly?
  - Constitutive law evaluations?

Remaining challenges/ideas

Sandia National Laboratories

- Can we use expression templates (or C++11 with move semantics) at the tensor level to minimize temporaries and aid cache performance
- Prefetching and other cache tricks?
- AVX-SIMD provides a very natural transition to eventual GPU implementations (minimal branching, related data layout requirements)
- But is available to us now!

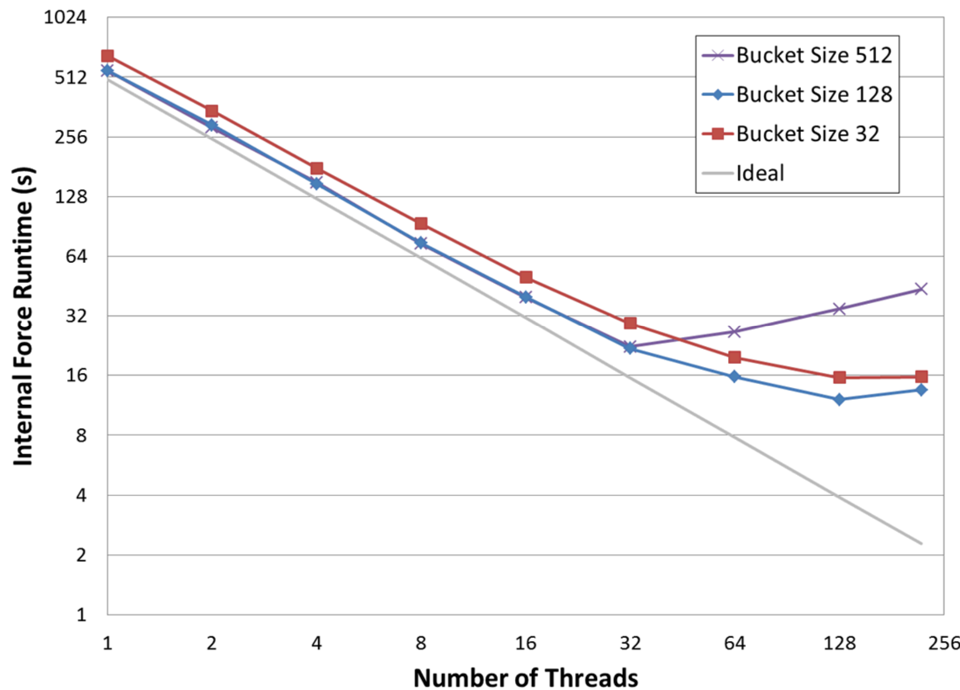
Real applications!

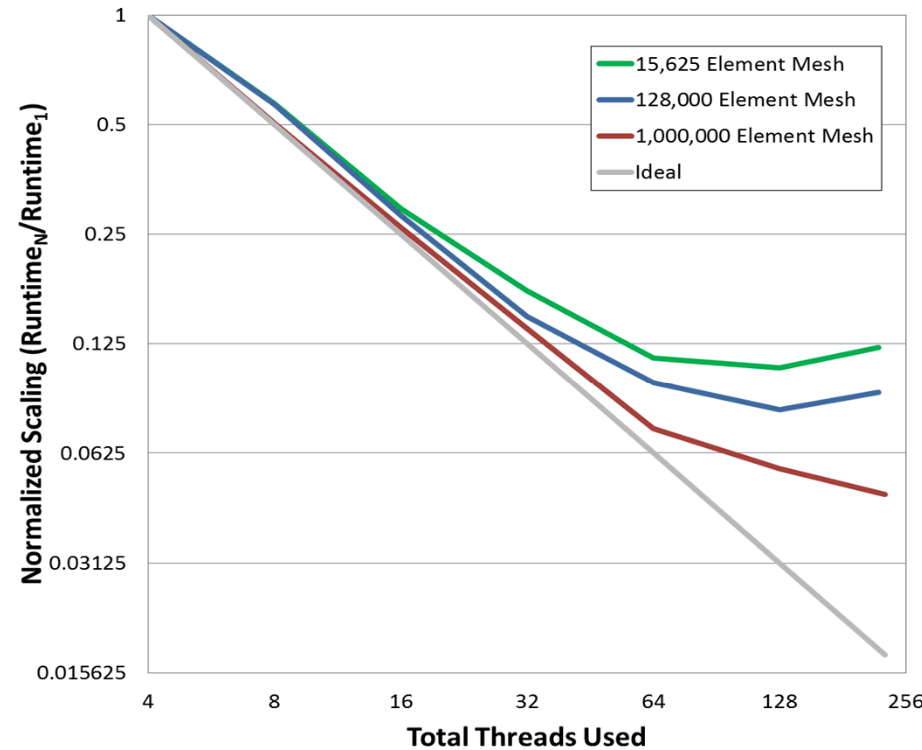
Sandia National Laboratories

- Goodyear milestone: get run time of < 1.5 x Abaqus
- Previously at ~1.8 x
- Initial SIMD implementation
  - ***~ 40% overall improvement***
  - now at **~1.1-1.2x** !
- High velocity impact simulation:
  - Originally, ~70% calculating 3x3 eigenvectors
  - Now ~10%

Early KNC results

- Sierra/SM compiles and runs on our test-bed KNC
- Use coloring algorithm to thread "force assembly"





Data provided by Nate Crane

Take aways

Sandia National Laboratories

- Solvers
  - Improve/thread sparse direct solvers
  - Bottleneck in both factorization and forward/backward solves
- Contact/Search
  - Serial cost dominated by random memory access
  - Parallel dynamic load balancing required
- Element assembly
  - Requires multi-threading (OpenMP?)
  - Better auto-vectorization would be nice
  - For now we are “hand” vectorizing