SAND2017-7255PE

Photos placed in horizontal position with even amount of white space between photos and header

Photos placed in horizontal position with even amount of white space between photos and header

Sierra's SIMD library for portable intrinsics based vectorization

KNL workshop, 2017

Michael Tupek

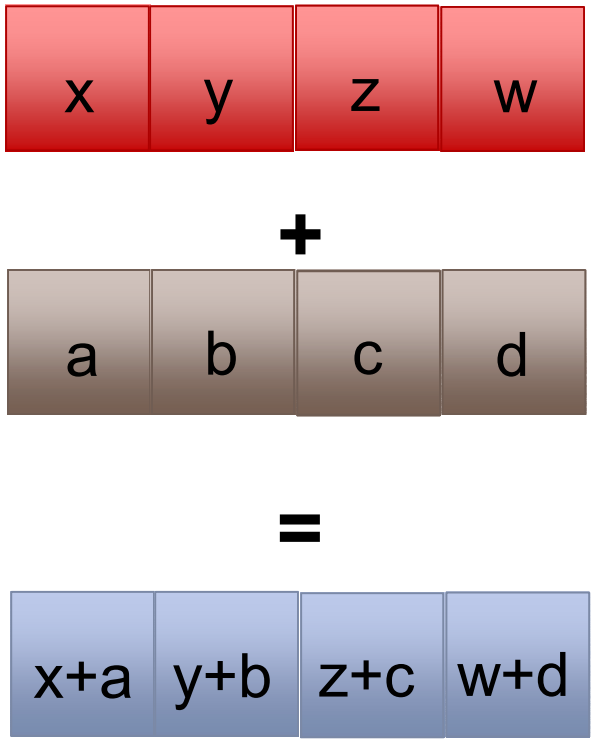
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

What is SIMD?

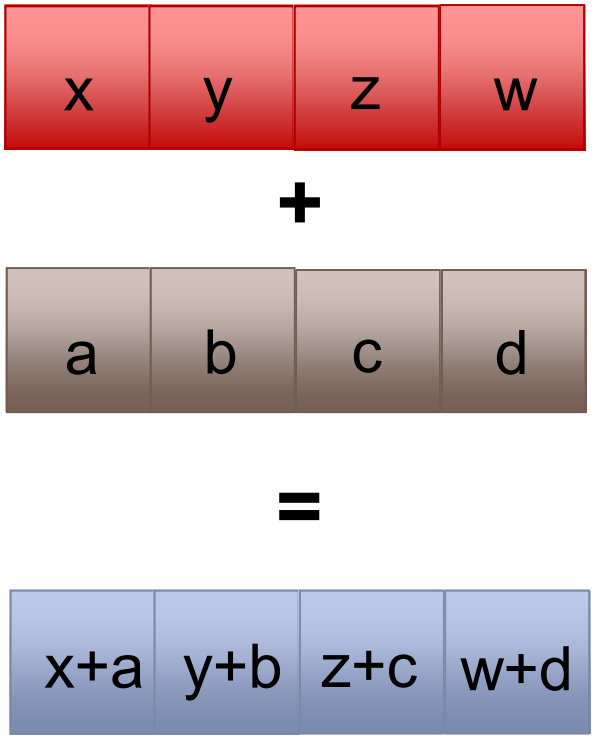
Sandia National Laboratories

Single Instruction, Multiple Data

- SSE2 instructions: (Intel, AMD ~2004)
  - 2 simd::Double, 4 floats
- AVX instructions (Intel, AMD)
  - 4 simd::Double, 8 floats
- AVX-512 instructions (Intel ~2014)
  - 8 simd::Double, 16 floats
- AltiVec (IBM)
- GPU (eg. CUDA): (Nvidia)
  - Can think of warp as 32-wide simd double



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



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:

- Nested loops: vector/tensor math, etc
- Eigenvectors
- Material models

Auto-vectorization

Sandia National Laboratories

- Simple loops can automatically use SIMD:

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

- "Complicated" loops rarely auto-vectorized efficiently:
  - Eigenvectors
  - Constitutive law evaluations
- Use SIMD vector intrinsics (low level functions):
  - Each intrinsic is equivalent to an assembly instruction

Outer loop vectorization

Sandia National Laboratories

- Very difficult for compiler to vectorize over outer loops

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

- Easy when using simd intrinsics and templating on double type

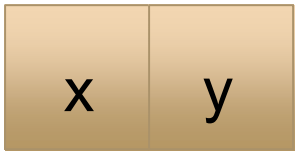
```
simd::Double **a, **b;
for (int i=0; i < N%simd_width; ++i) {
   function_with_inner_loop( a[i], b[i] );
}

template <typename T>
void function_with_inner_loop( T* array1, T* array2) {…}
```

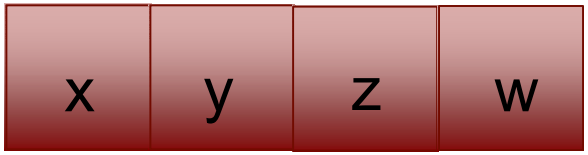
SSE2/AVX intrinsics (Intel, AMD)

Sandia National Laboratories

__m128d (2 simd::Double)

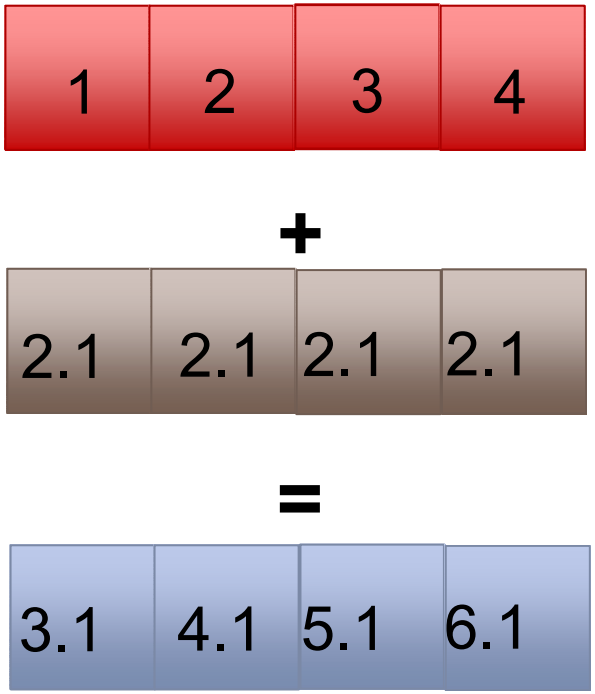


__m256d (4 simd::Double)



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 num_doubles = 4;
    class simd::Double { __m256d d };
#elif defined(SSE2)
    const int num_doubles = 2;
    class simd::Double { __m128d d };
#else
    const int num_doubles = 1;
    typedef double simd::Double;
#end
```

main.cc:

```
#include <Simd.h>

double x[nsimd::Double];

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

// operator overload:
simd::Double c = a+b;

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

Sierra SSE2/AVX interface

- Difficult to have portable code:

```
simd::Double 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)

SIMD functionality

Sandia National Laboratories

Standard math functions:

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

Simd boolean (mask) types:

<, <=, >, >=, == returns booleans

```
simd::Bool isTrue = x < 5;
```

Simd ternary:

```
simd::Double z =
if_then_else(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 simd::Double:

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

SSE2 (on blade, gcc)

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

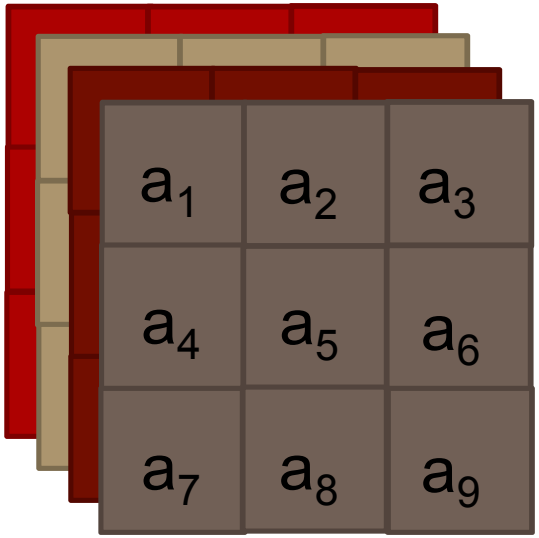
AVX (on chama, Intel compiler)

- **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.

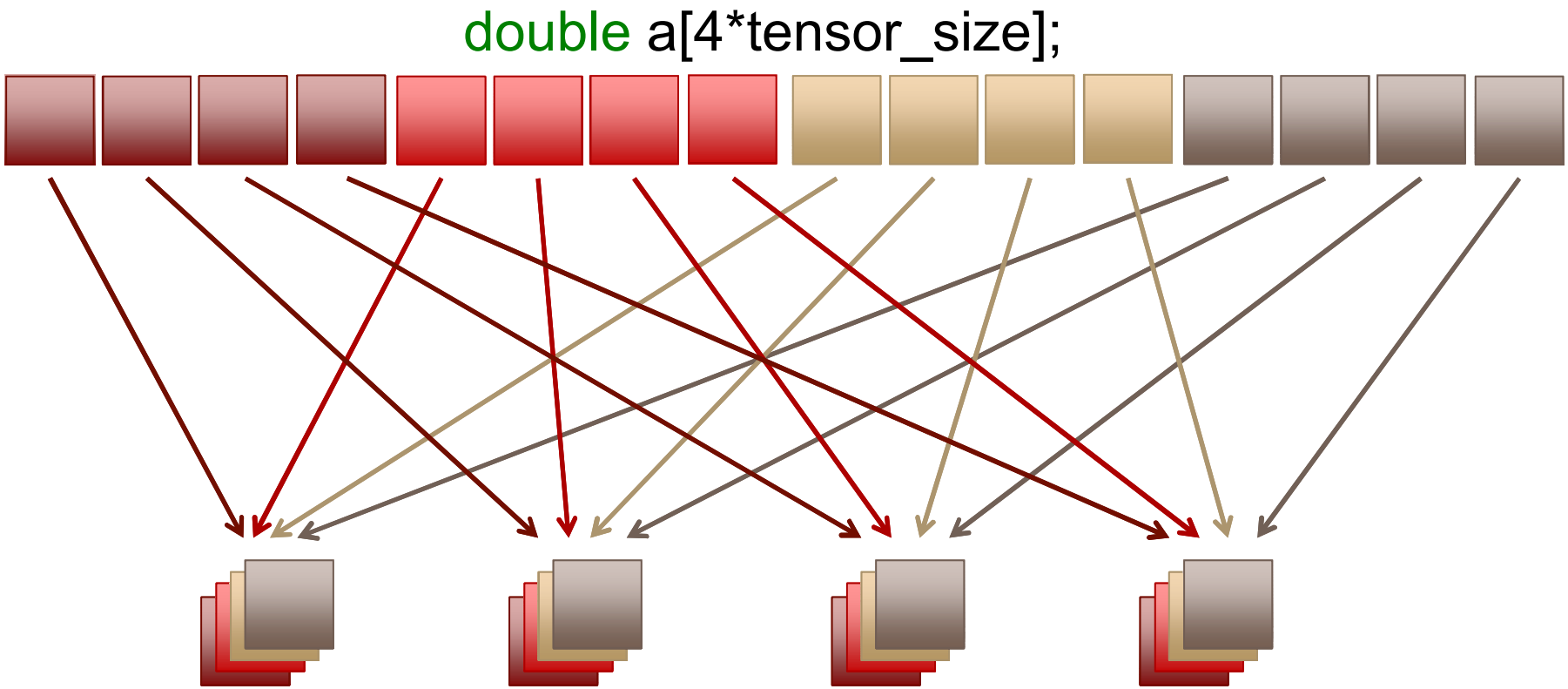
SIMD Tensor class



Process 4 tensors at a time (AVX):

```
double tensors[4*9];
// fill 4 tensor
Tensor33<simd::Double> 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




```
simd::Double 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

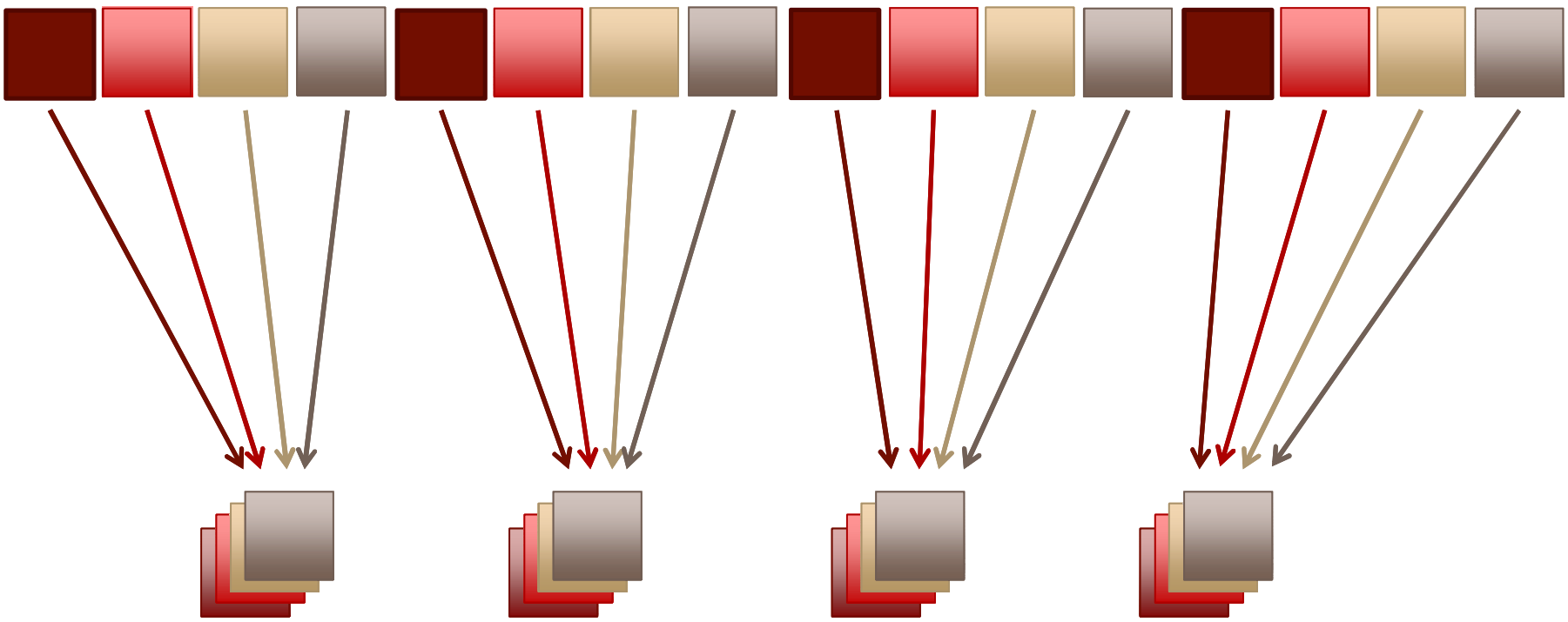
Sandia National Laboratories

- **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];
```



```
simd::Double 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

Vectorized Material Models

- Re-implemented 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) ) {
  while ( any(notConverged) ) {
    // compute plasticity model sub-iterations
  }
} else {
  // 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:

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

Branches handled carefully to avoid expense

SIMD ternary operator

Sandia National Laboratories

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

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

```
simd::Bool b = y < z;      // overloaded element-wise comparisons
simd::Double x = simd::if_then_else(b, v, w);       // fake ternary
```

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

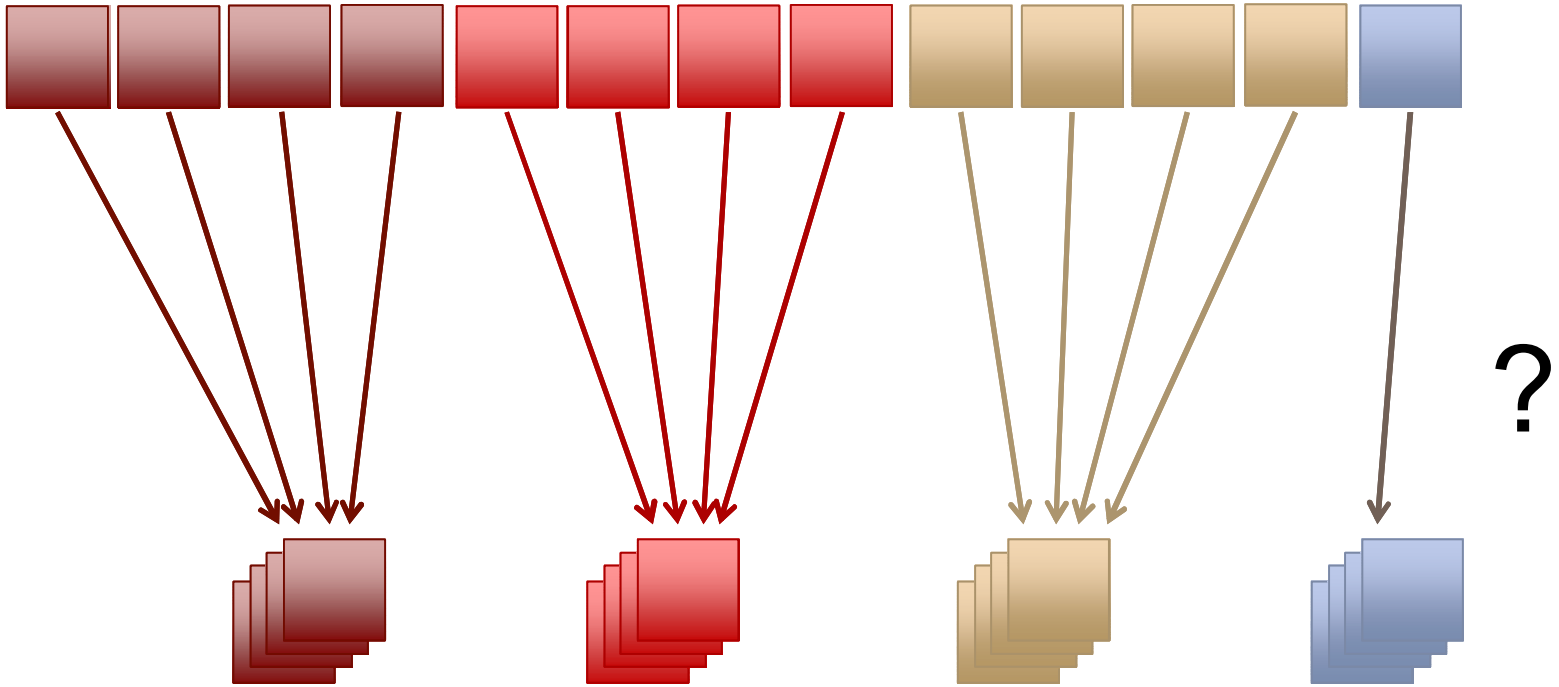
Sandia National Laboratories

- simd::Double 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.

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…



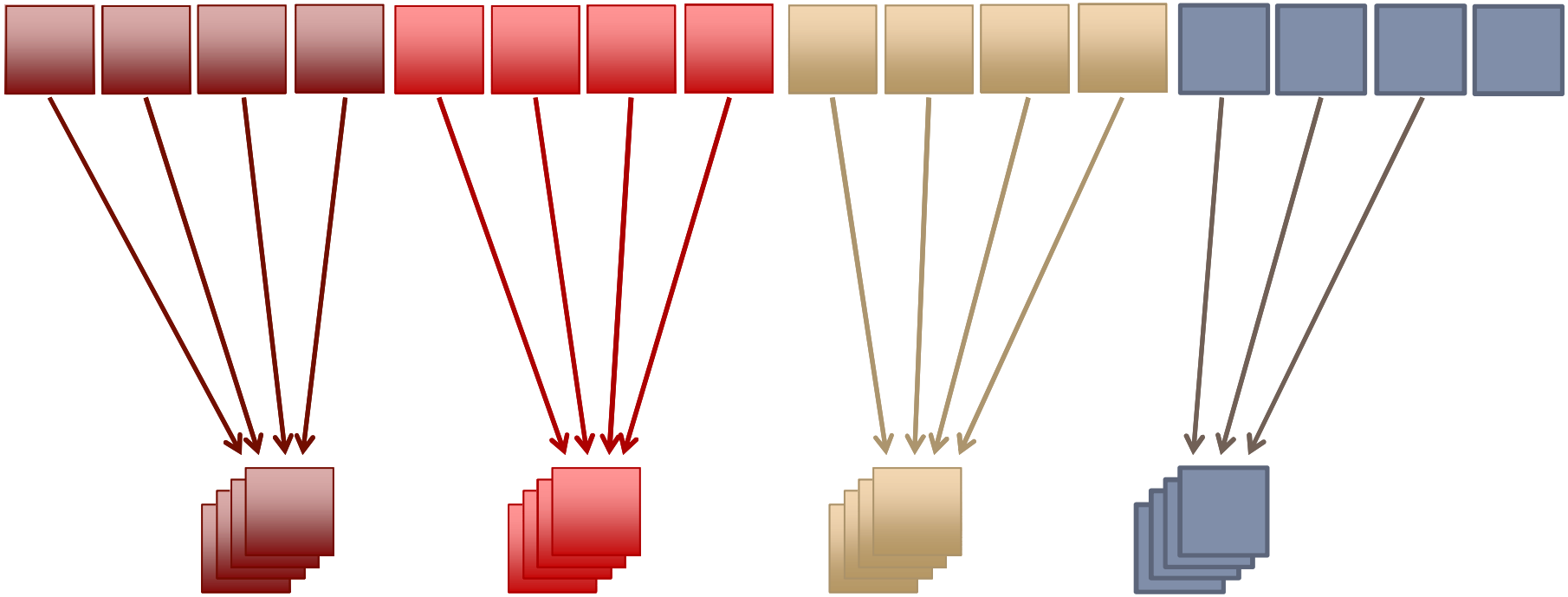
Code example: handling the remainder

Sandia National Laboratories

```
for (int e=0; e < nelems; e+=simdWidth)  {
    simd::Double xLoc;
    if ( not_at_end_of_loop(e) ) {                    // load all 4
        xLoc = simd::load(x+e);
     } else {
        xLoc = simd::load_part(x+e, nelems-e);        // load remainder
    }
}
```

Alternative: cast directly to simd::Double*

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



Code example: casting to simd::Double*

```
simd::Double* xLoc    = simd_ptr_cast(x);     // x is array of '3 double'
simd::Double* dotLoc = simd_ptr_cast(dot);  // dot is array of double

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

>1.5 X improvement over original layout/load strategy

Real applications!

Sandia National Laboratories

- Sierra-SM (since 2015)
  - Implemented for key kernels only
  - *~2X overall improvement using AVX*
  - Tensor math speedups:
    - 2.2 – 3.6 X (Haswell)
    - 2.4 – 5.3 X (KNL)
- Sierra-TF (since 2017)
  - Refactored ~80% of code to use Kokkos and enable simd types
  - Thermal matrix assembly speedups:
    - 1.6 X (Haswell)
    - 3.7 X (KNL)
- Nalu (since 2017)
  - Implementation in progress
- SPARC (2018?)

Remaining challenges

Sandia National Laboratories

- Prefetching and other common optimizations
- Effective even for bandwidth limited loops. iCache?
- Integration with Kokkos Views
- Already open source in Trilinos (stk::simd)
  - For integration with Kokkos, moving to its own github
  - Have prototyped a platform portable “SimdView”
- New C++ standard proposal might replace this functionality – a long time off