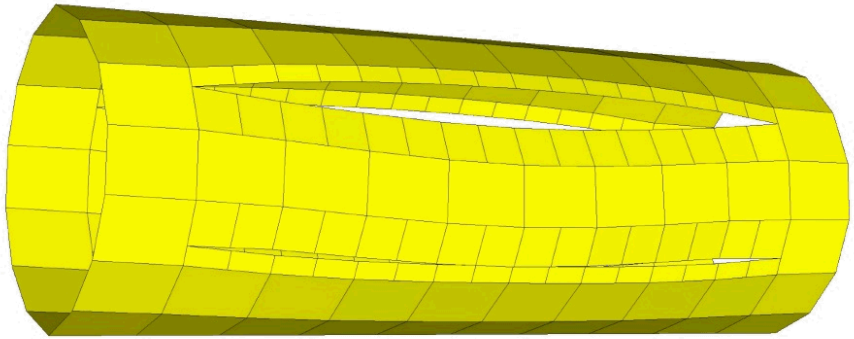
SAND2017-1067PE

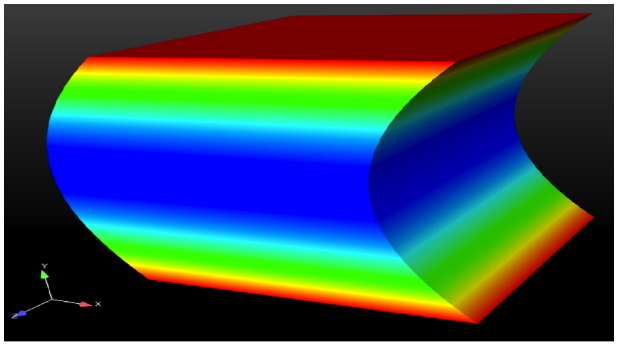
Photos placed in horizontal position with even amount of white space between photos and header

Sierra-SM's NGP experience

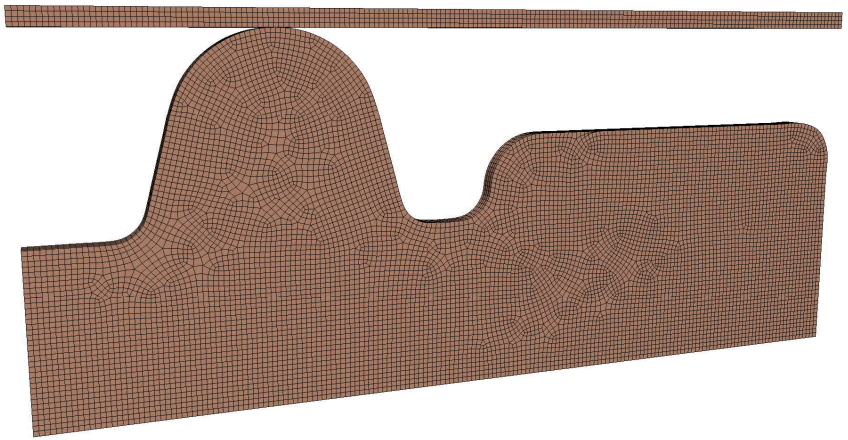
JOWOG-34, 2017

SIERRA/SM (Solid Mechanics)





- General purpose massively parallel nonlinear finite element code
  - explicit transient dynamics
  - implicit transient dynamics
  - quasi-statics analysis.
- Built upon libraries for
  - material models
  - solid and structural
  - contact
  - linear and nonlinear solvers



- Used for analyzing challenging nonlinear mechanics problems for normal, abnormal, and hostile environments

Sierra/SM GPU strategy

Sandia National Laboratories

Plan: GPU enable a subset of SM capabilities to run on ATS-2 using ***Kokkos***

- **Currently enabled (prototype)**
  - Explicit dynamics
  - Hex & linear beam elements
  - Elastic material model
  - Displacement BCs
  - Gravity
  - Basic output

- **Additional capabilities needed for primary use cases**
  - Implicit
  - Contact
  - Element death
  - Tetrahedral, shell, and beam elements
  - *All* production material models
  - Additional BCs
  - Additional Output
  - "MPI+X" parallelism (MPI + OpenMP + GPU)

Target ~**75** unique code components in Sierra/SM

(out of ~**500** total)

Adagio on the GPU: First light

- Initial implementation based on prototype Kokkos “FlatMesh”
  - uniform element topology and materials
  - single GPU only, no MPI
- Kokkos Linear Beam

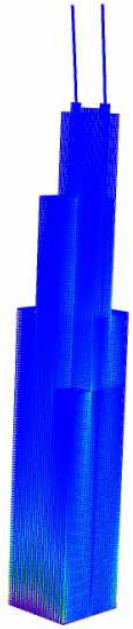
	Total runtime (min)
Serial CPU	366
CPU/GPU	17
Speedup	**22x**

- Kokkos UG Hex

	Total runtime (s)			
Timing breakdown	Kokkos hex (GPU)	Original hex (CPU)	Kokkos hex (CPU)	**Speedup**
Total	335	5820	11951	**17x**
Internal force only	186	5007	11027	**27x**

Baseline version vectorizes

Lost CPU performance due to SIMD/vectorization

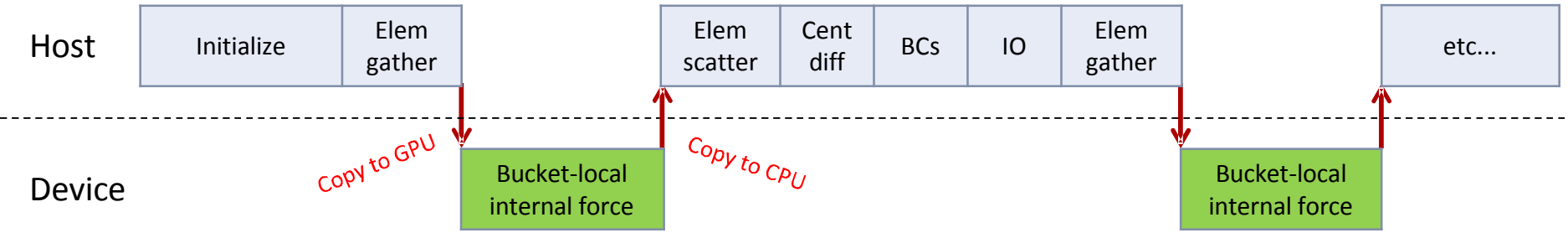


Sears tower model

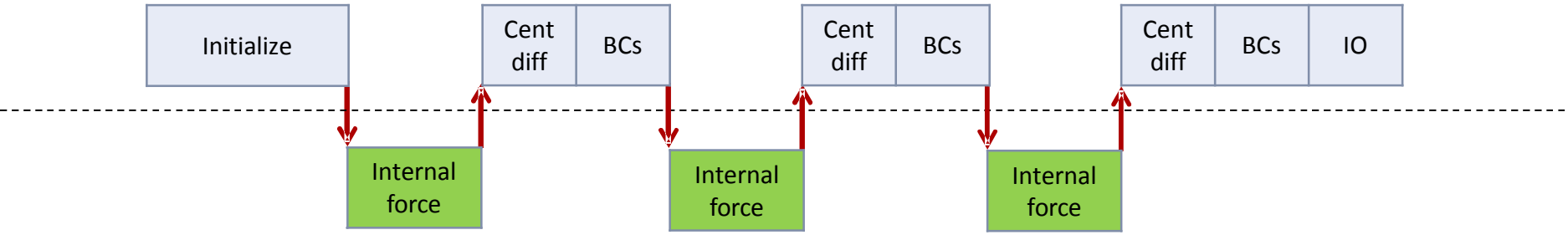
Lesson learned: data must stay on GPU

Sandia National Laboratories

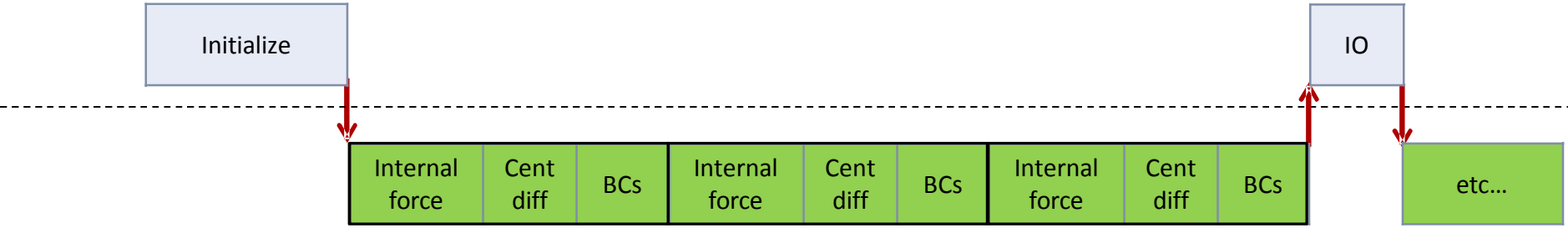
Strategy 1: Bucket-level GPU copy: ~20X slowdown vs. serial CPU



Strategy 2: Algorithm-level GPU copy: ~8X slowdown

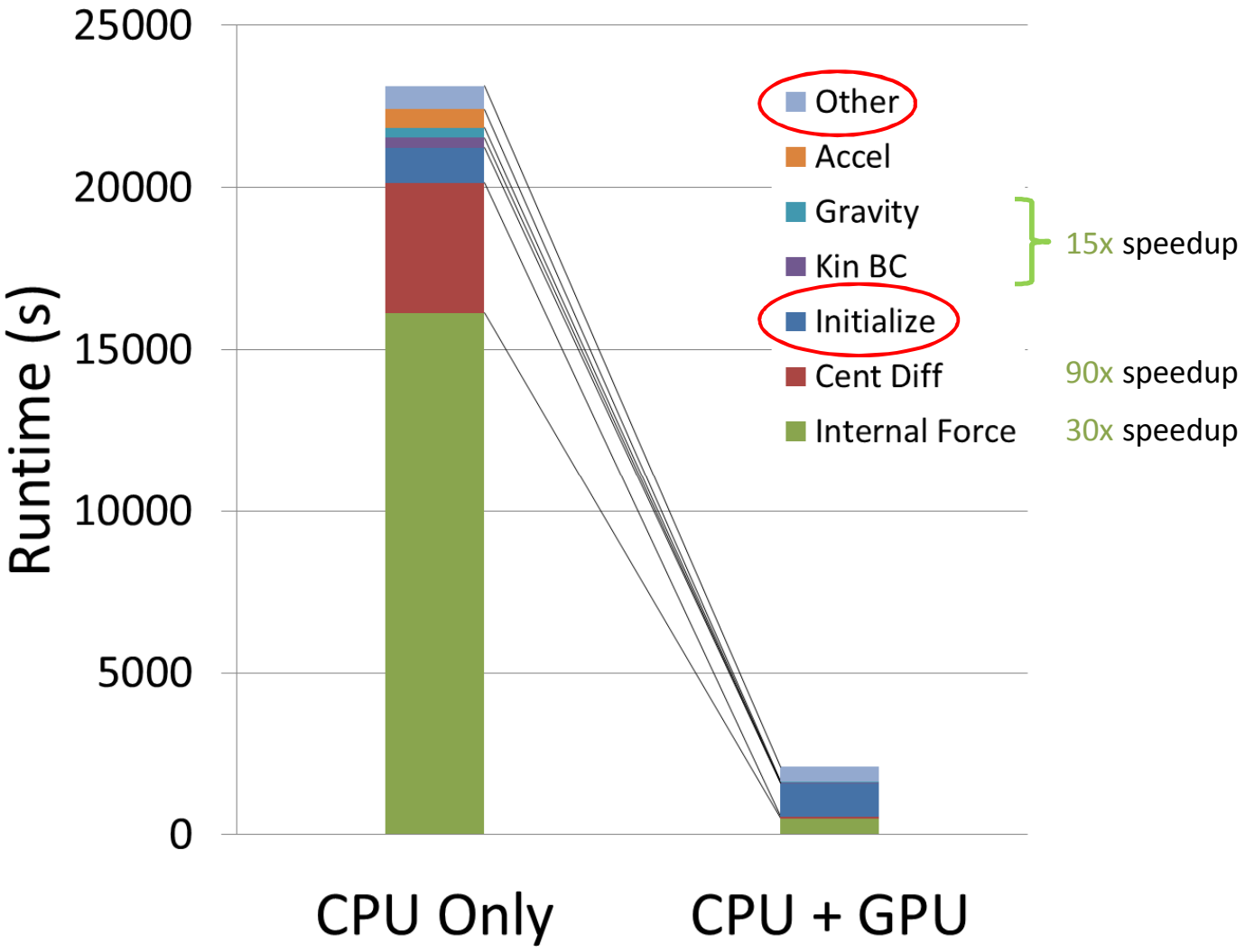


Strategy 3: Run-level GPU copy: ~22X speedup!



May change with NVLink2, but will it change enough to invalidate the conclusion?

Lesson learned: serial algorithms dominate



STK ngp::Mesh

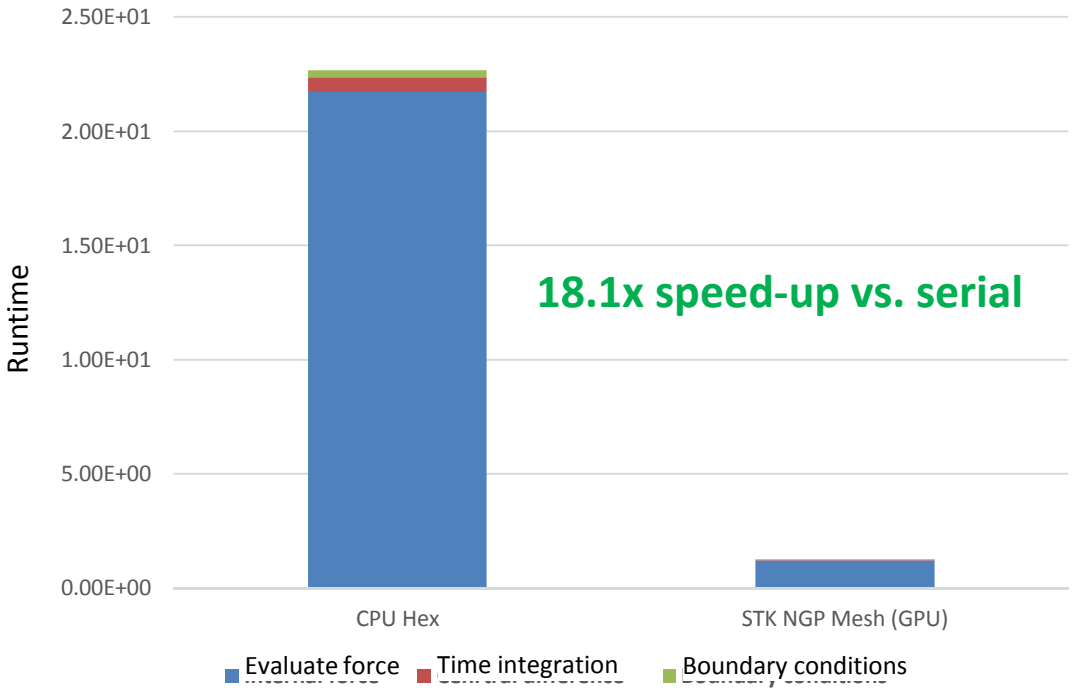
Sandia National Laboratories

- GPU-friendly API for mesh/field access
  - Stores data in Kokkos Views on GPU
  - Thin wrapper around STK Mesh when running on CPU
- Allow STK apps to start running on GPUs with subset of features
- Easy to prototype different implementations and data layouts
- Hierarchical parallelism over buckets and entities (e.g. nodes and elements)

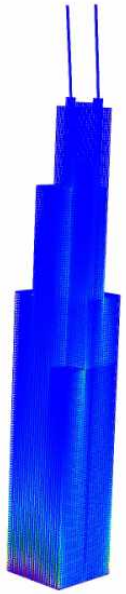
```
ngp::for_each_entity(ngpMesh, ELEM_RANK, selector,
    KOKKOS_LAMBDA(ngp::Mesh::MeshIndex elem)
    {
        kernel(ngpMesh, elem, coords);
    });
```

Using STK ngp::Mesh

- Fundamentally different from prototype “FlatMesh”:
  - More indirection, but more generality (different element formulations, parts, materials…)
  - Multilevel parallelism (buckets, elements/nodes)
  - Different data layout (component-fastest vs. element-fastest)
- Some algorithm refactoring to improve performance:
  - Break up large kernels into smaller subsets
    - reduces register pressure, more data storage and access

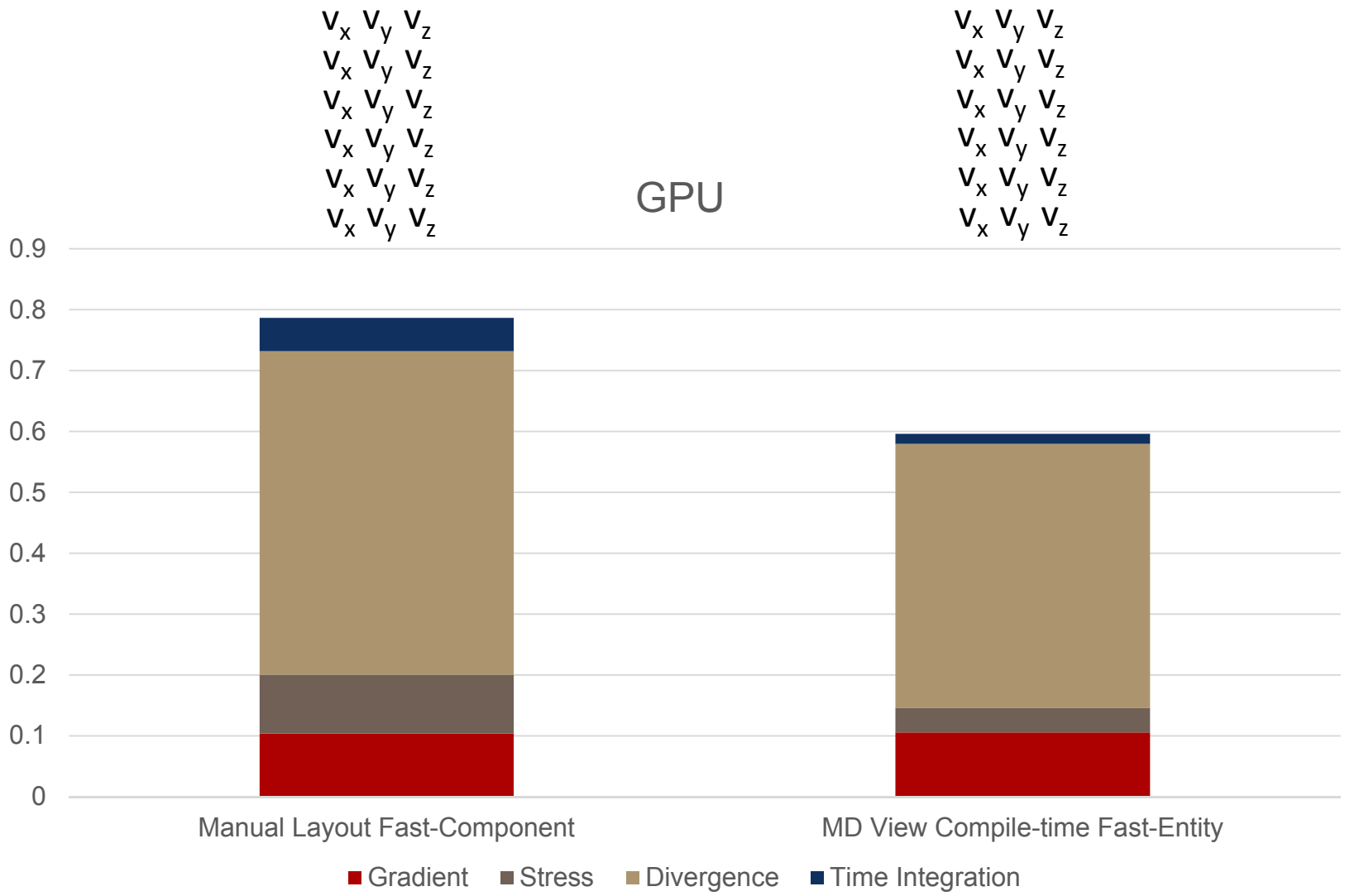


Sears Tower hex model (835,000 elements)

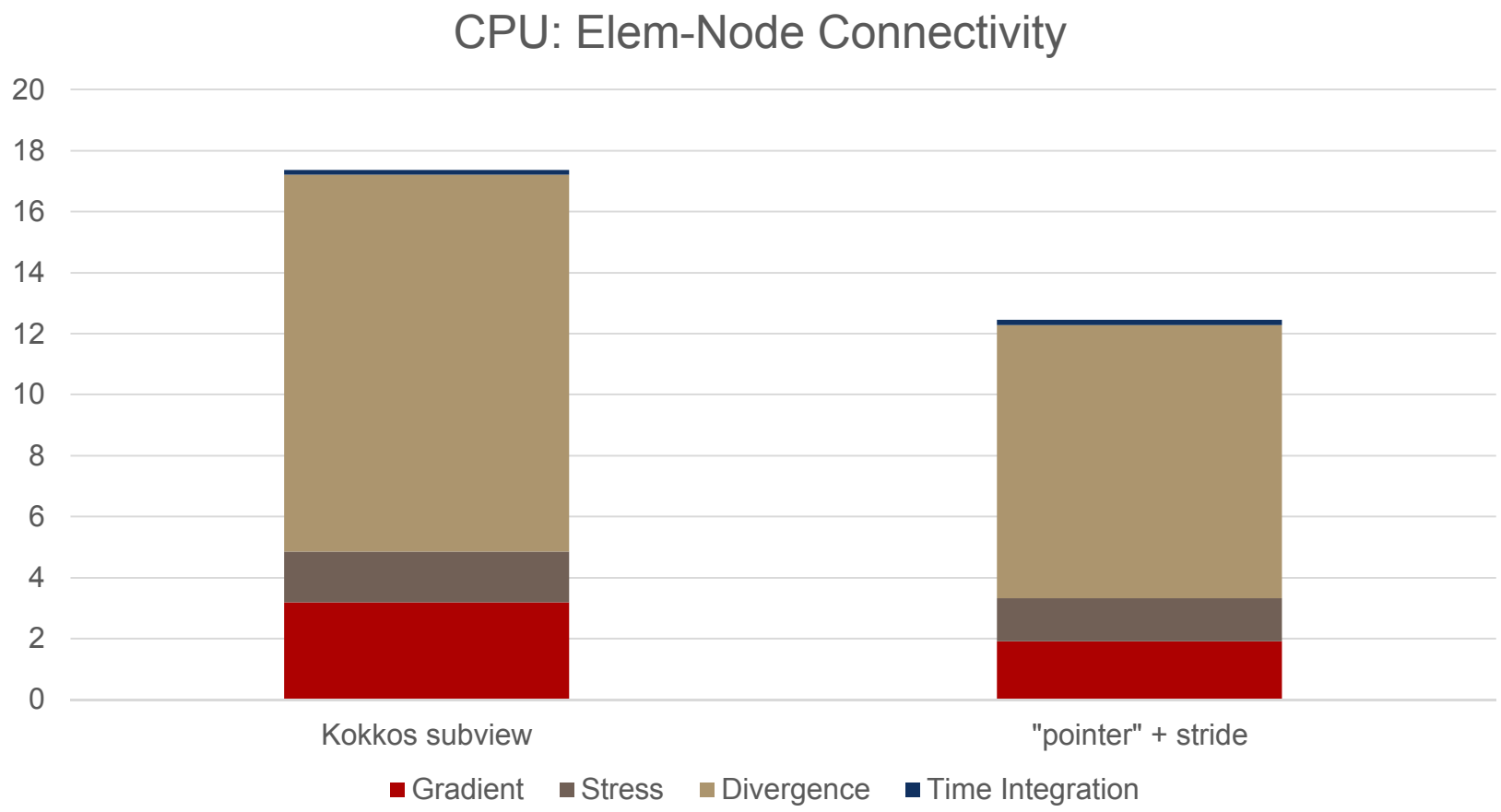


“Flat mesh” was 22x vs. serial
Is this the cost of generality?

Importance of optimal layout



Kokkos “Subview” expensive at low levels



Profiling Data Game (Max Values)

so, which run is faster?

Sandia National Laboratories

Metrics	Run A	Run B
stall_not_selected ↑	13.43%	3.21%
stall_memory_dependency ↕	38.47%	65.22%
stall_memory_throttle ↓	15.58%	6.92%
stall_exec_dependency ↓	10.44%	12.64%
stall_pipe_busy ↓	12.09%	2.79%
stall_sync ↓	0.00%	0.00%
stall_inst_fetch ↓	2.61%	3.64%
achieved_occupancy ↑	0.929071	0.951608
gld_requested_throughput ↑	77.777GB/s	44.186GB/s
gst_requested_throughput ↑	0.00000B/s	0.00000B/s
gld_efficiency ↑	47.54%	132.20%
gst_efficiency ↑	0.00%	0.00%

continued…

Profiling Data Game (Max Values)

so, which run is faster?

Sandia National Laboratories

Metrics	Run A	Run B
IPC ↑	0.843783	1.197995
Flops Efficiency ↑	0.78%	1.08%

Profile data for Nodal Volume Calculation using

- **Run A**: Field
- **Run B**: ConstField (with random access/texture memory)
  - faster by 30%

Profiling SM Gradient

incremental performance improvements using profiling data

Sandia National Laboratories

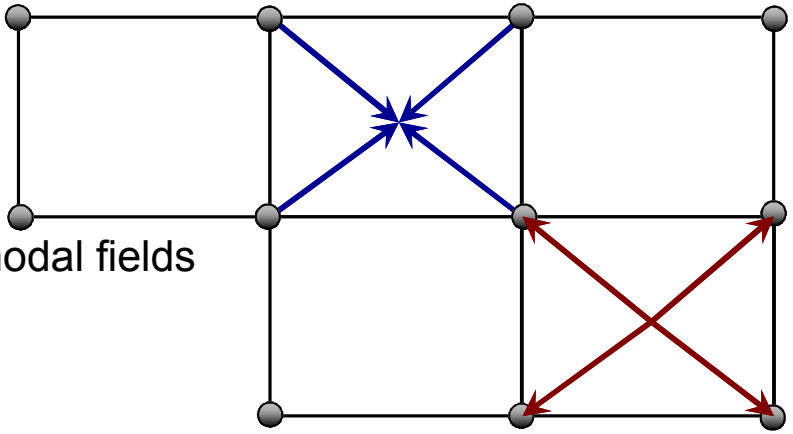
Metric	Baseline	Remove gathers	Random access memory	Coalesced memory writes
time	0.14	0.11	0.10	0.10
stall_not_selected	9.39%	4.87%	1.47%	1.28%
stall_memory_dependency	36.97%	47.85%	66.83%	72.33%
stall_memory_throttle	32.75%	26.73%	11.74%	8.40%
stall_exec_dependency	10.13%	13.95%	13.43%	12.11%
stall_pipe_busy	8.72%	4.71%	1.83%	1.49%
achieved_occupancy	0.432681	0.308475	0.307971	0.308161
gld_efficiency	35.32%	35.16%	54.86%	64.97%
gst_efficiency	27.03%	27.03%	27.03%	100.00%
gld_requested_throughput	45.53 GB/s	36.91 GB/s	6.074 GB/s	6.075 GB/s
gst_requested_throughput	5.316 GB/s	6.450 GB/s	6.586 GB/s	6.903 GB/s
ipc	0.461778	0.406153	0.410155	0.397081

Optimization Lessons: Memory access dominates runtime

Sandia National Laboratories

- Runtime cost is dominated by random memory access
  - Gathers cost ~16% percent
  - Scatters cost ~65% of total runtime (with atomic adds)
  - Coalesced memory access > 3 times faster than gathers
  - If we only do math (no memory access), 45 times faster than baseline

Elements gather from nodal fields (e.g. velocities)



Elements scatter to nodal fields (e.g. forces)

- Most other attempts at intuitive optimizations produced a slowdown

Big questions and risks

Sandia National Laboratories

- Current ~20x GPU speedup vs. single core CPU is underwhelming…
  - What can we do with additional code restructure? (*in progress…*)
  - How will future hardware perform?
- How do we amortize non-GPU algorithm costs (like results output)?
- How do we avoid algorithm duplication?
- MPI communications will need to reach into GPU data each time step. Will this be a major performance bottleneck?
- Is the GPU really worth it for unstructured finite element codes?
  - May need to fit entire problem on GPU to avoid data copy costs
  - Can we develop novel algorithms which efficiently use GPUs relative to CPU?
    - Structured grids
    - Multi-scale
    - FFTs
    - Math heavy material models

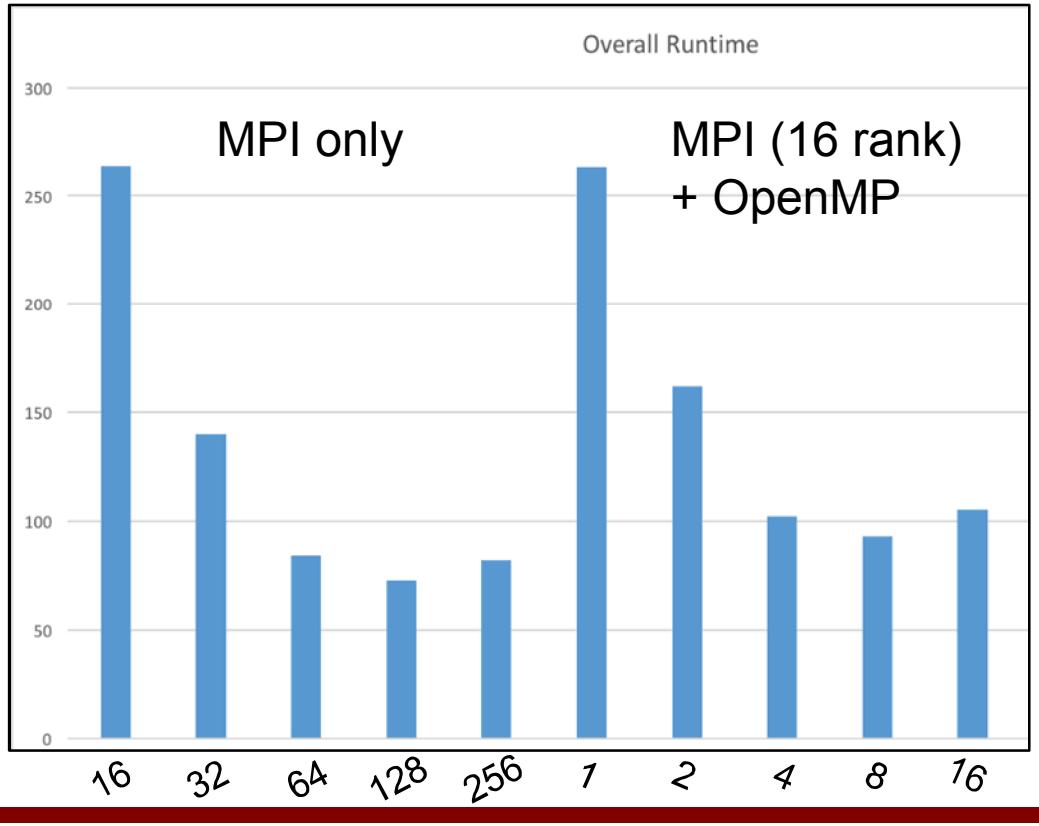
Additional efforts

Sandia National Laboratories

- ATS-1: KNL
  - KNL threading performance still below MPI everywhere
  - Intrinsic based vectorization using stk::simd, a big win!
  - Developing Kokkos::SimdView<double*>
- Thread scalable search
  - For GPU: Morton Linearized Bounding Volume Hierarchy (LBVH)
  - For CPU: either OpenMP parallel KD-tree or LBVH (already Kokkos ready)
- Contact enforcement on the GPU (very early stages)
- Exploiting structured grids (for coalesced memory access)

OpenMP Threading for KNL

- MPI everywhere still gives the best performance
  - Oversubscribing MPI seems to help (128 ranks / KNL)
  - Best threading performance is close (some routines still scale poorly)
  - Node-to-node: Haswell performance slightly better than full KNL
- SIMD/vectorization seems more important
  - Use in-house stk::simd to ensure vectorization and get platform portability



ATDM contact: thread-scalable search

Sandia National Laboratories

Background:

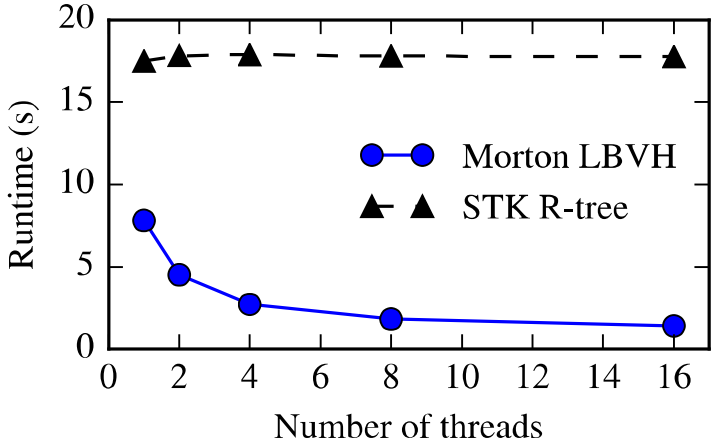
- Team has implemented thread-scalable proximity search algorithms (Morton LBVH) for contact
  - Prototyped in ATDM minicontact
  - Incorporated into Geometry Toolkit in previous sprint work
- In support of ATDM Q4 contact milestone, threaded algorithms incorporated into suite of STK search algorithms

This sprint:

- Wrote a news note on threaded performance of Morton algorithm

Future work:

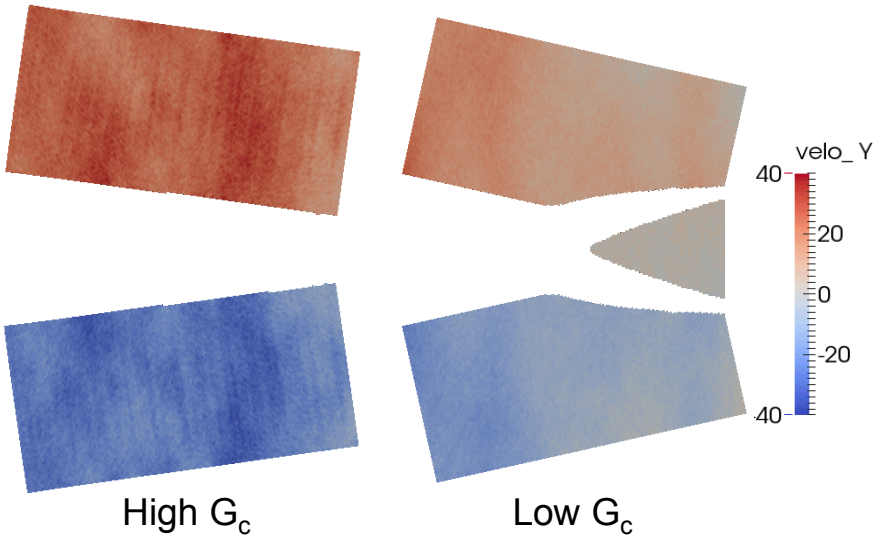
- Continue maintenance/support of search algorithms in collaboration with STK team
- Process copyright info necessary to transition code to open domain

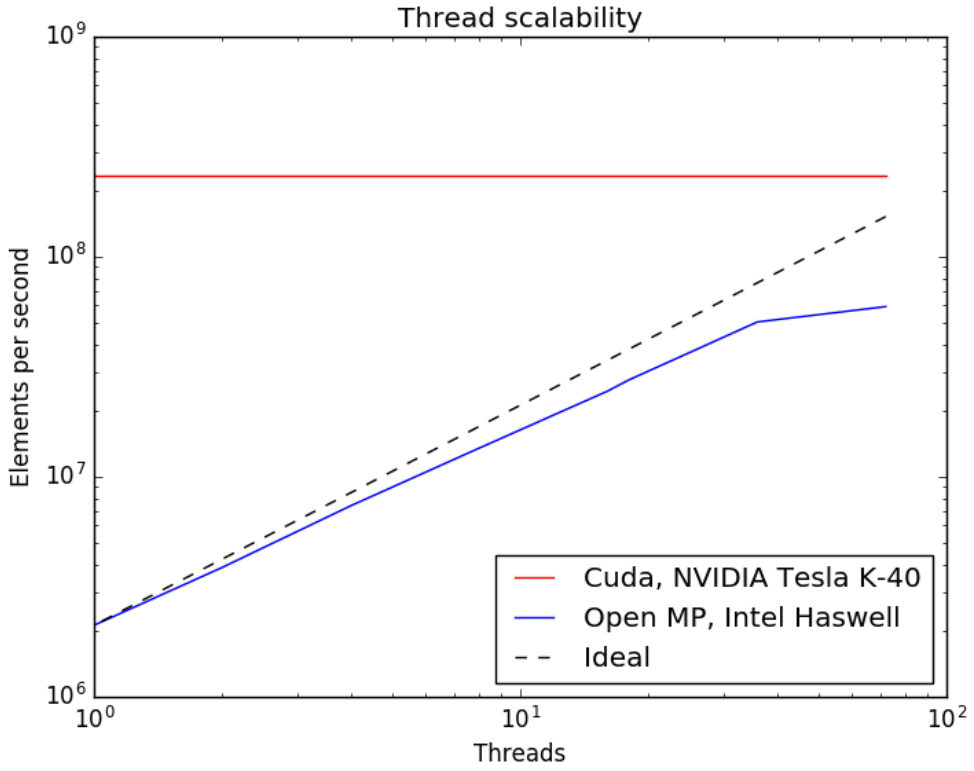


Structured grid FE simulations with Kokkos

- GPU explicit dynamics > 100x faster than serial CPU
- 1 GPU processes ~250 million element calculations / second
- Up to ~20 million structured grid elements fit on single Kepler-K80
- OpenMP also scales well

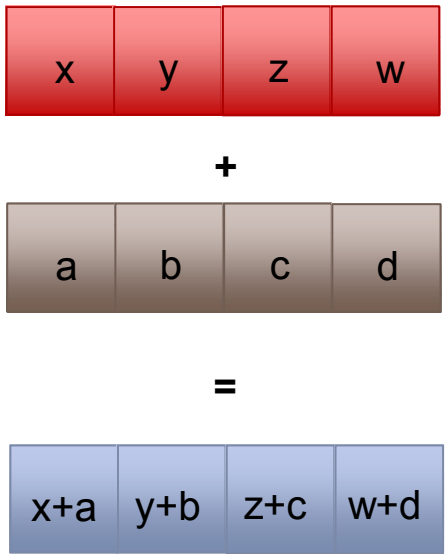
Example problem:
Mode-I crack fracture, transition to branching





stk::simd

- Directly uses vector intrinsics
- Write algorithms to use a simd::double type (via templating)
- Interface designed to be independent of simd-width
- Obtains ~95% the performance of auto-vectorization, but
  - always works
  - more portable
  - easy to outer loop vectorize
  - support conditionals
  - easier to develop and debug
    (no #pragma, vector-reports)

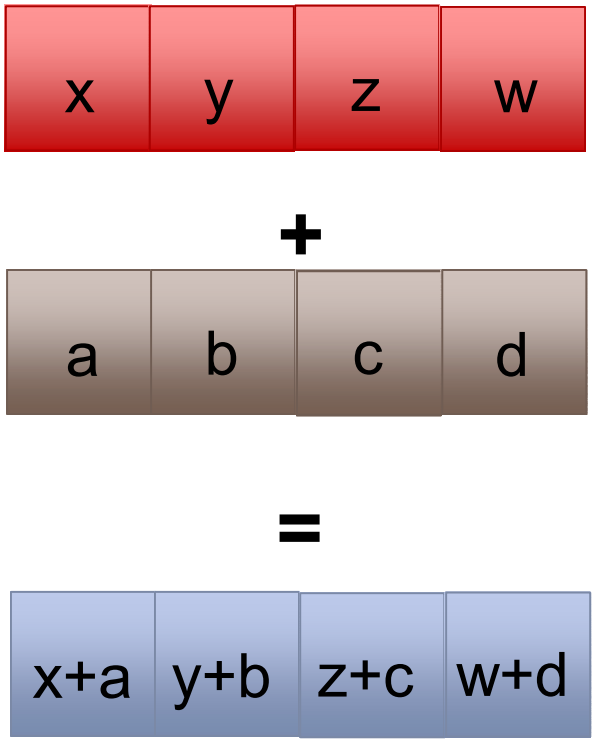


What is SIMD?

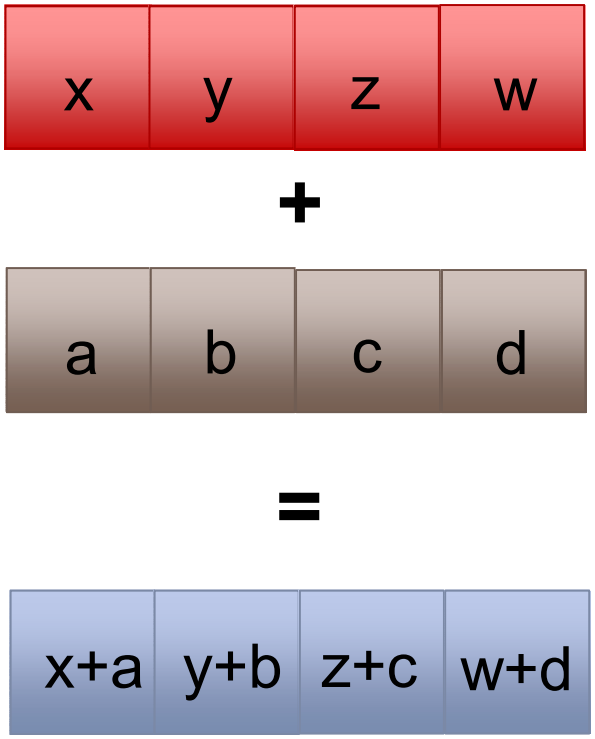
Sandia National Laboratories

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)
  - 32 doubles



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:

- Tensor33 multiply
- Eigenvectors
- Constitutive law evaluations

Auto-vectorization

Sandia National Laboratories

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

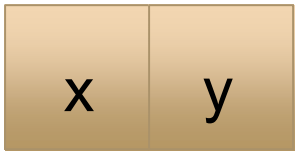
```
for (int i=0; i < N; ++i) {
  a[i] = b[i] + c[i] * d[i];
}
```

- "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

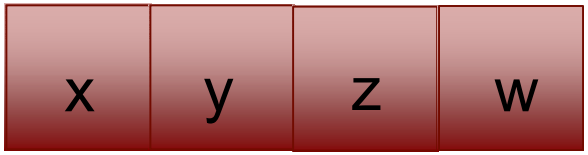
SSE2/AVX intrinsics (Intel, AMD)

Sandia National Laboratories

__m128d (2 doubles)

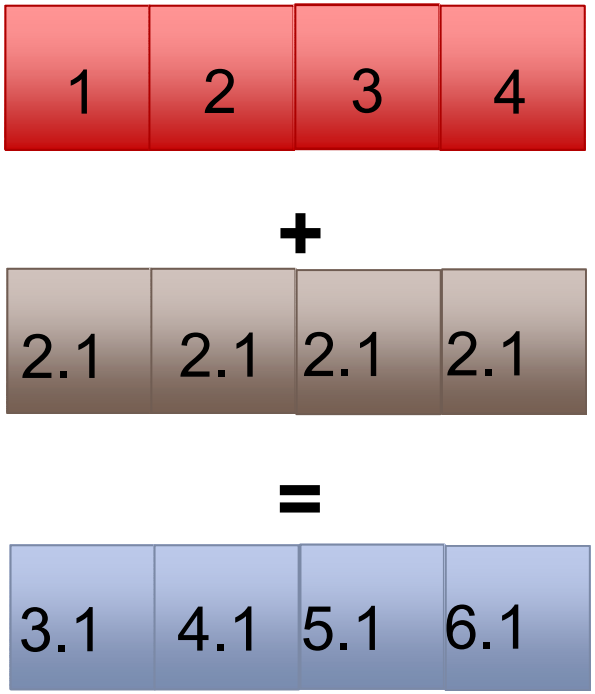


__m256d (4 doubles)



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);
```

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”

Sandia National Laboratories

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)

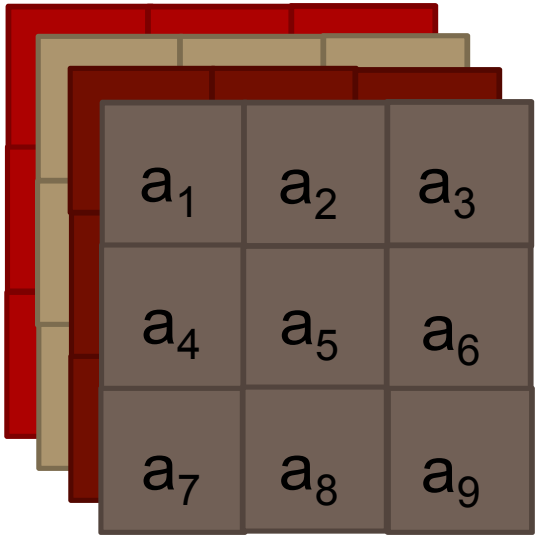
AVX (on chama, Intel compiler)

- **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.

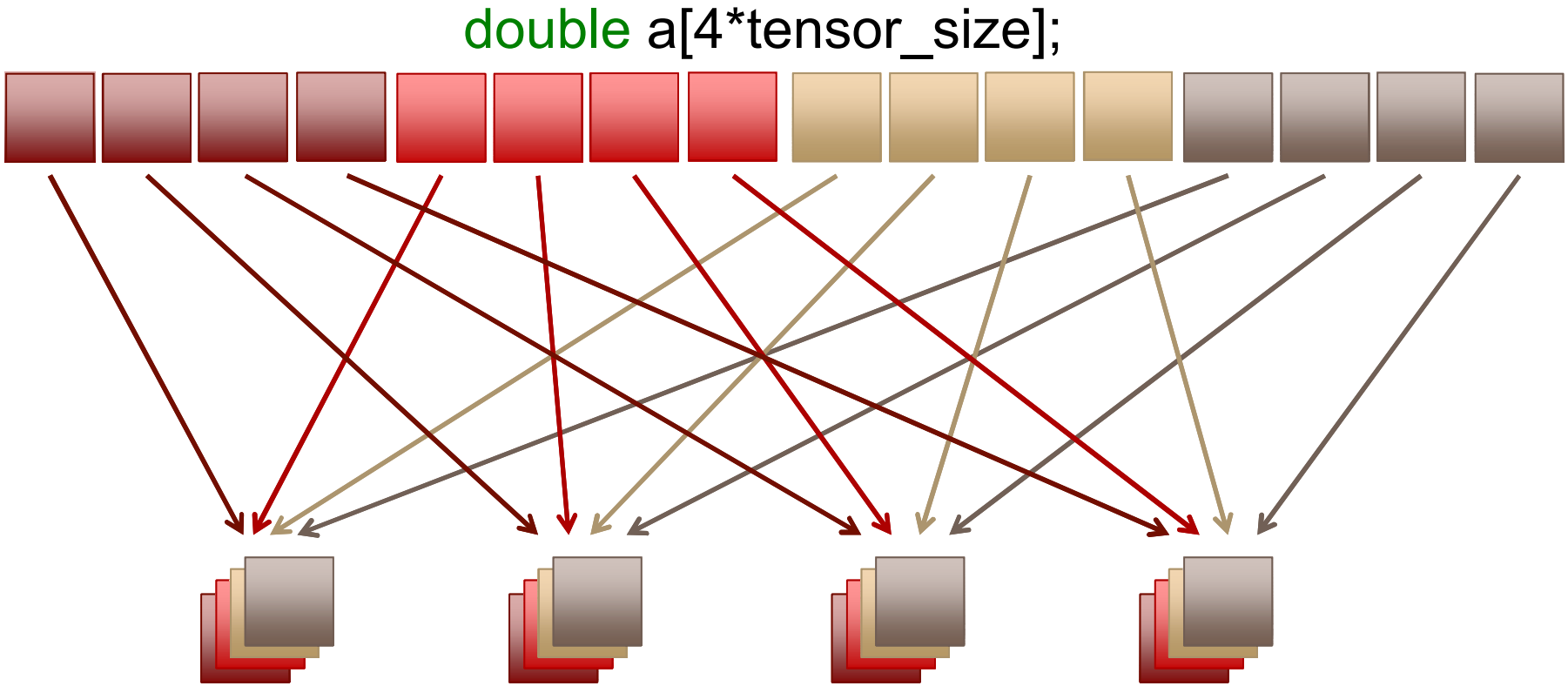
SIMD Tensor class



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




```
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**