SAND2017-11988PE

SIMD Scalar Types for Outer-loop Vectorization

Eric Phipps, Kyungjoo Kim, Sivasankaran Rajamanickam, and Michael Tupek
Sandia National Labs
November 1, 2017

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND NO. 2011-XXXXP

Vectorization is hard

Sandia National Laboratories

- Achieving good vectorization is critical for performance on emerging architectures
  - Intel AVX-512 instructions on KNL for 8/16 flops per clock
- We rely on the compiler to auto-vectorize code, which typically doesn't work well
  - Compiler can only vectorize simple loops with no carry dependencies
  - Loop trip count usually unknown at compile time
- Even if it does vectorize, the quality is usually poor
  - Potentially unnecessary peel, tail loops and masking instructions
- Compiler can only auto-vectorize inner-most loops
  - Trip count is usually small and is not a multiple of SIMD width
- Best chance for good vectorization is vectorizing outer loops
  - Most parallelism
  - The compiler has no chance of doing this for you

Vectorization through SIMD scalar types

Sandia National Laboratories

- Groups at SNL have been exploring use of SIMD scalar types for implementing outer-loop vectorization
  - C++ derived class that acts like a normal scalar
    - Contains an array of floating-point data matched to architecture's SIMD width
  - Implements all math operations (+,-,*,/,sin,sqrt,...)
    - May use vector intrinsics for those operations (e.g., _m256_add_pd())
  - Replace scalar type (float/double) with SIMD type
    - Templates can be used to facilitate this, but isn't required

- Discuss 3 use-cases here:
  - Uncertainty quantification
  - PDE assembly
  - Solvers

CRS-Format Matrix-Vector Product

Sandia National Laboratories

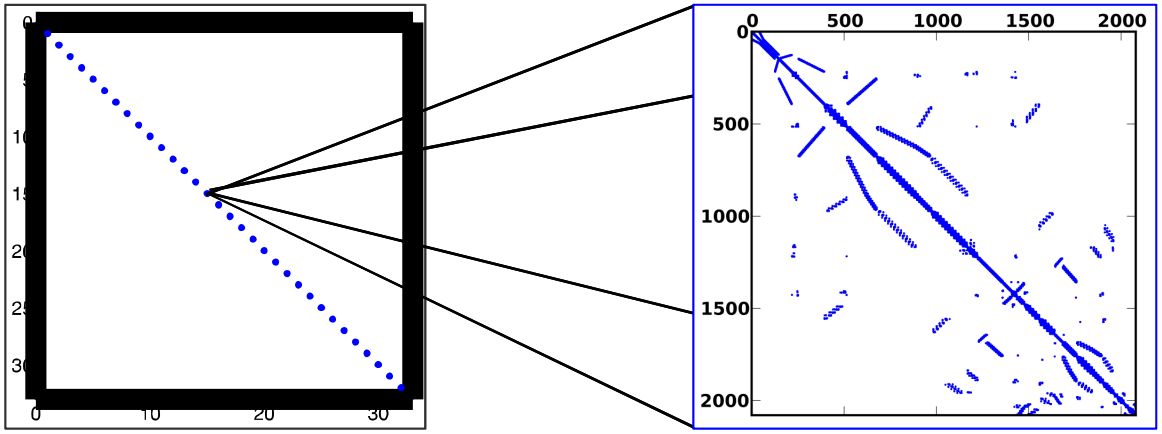
```
// CRS Matrix for an arbitrary floating-point type T
template <typename T>
struct CrsMatrix {
    int num_rows;      // number of rows in matrix
    int num_entries;  // number of nonzeros in matrix
    int *row_map;      // starting index of each row, [0,num_rows+1)
    int *col_entry;   // column indices for each nonzero, [0,num_entries)
    T   *values;       // matrix values of type T, [0,num_entries)
};

// Serial CRS matrix-vector product for arbitrary floating-point type T
template <typename T>
void crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; row<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry] * x[col];
        }
        y[row] = sum;
    }
}
```

Simultaneous ensemble propagation

- PDE: $$f(u, y) = 0$$
- Propagating m samples – block diagonal (nonlinear) system:

$$F(U, Y) = 0,\ U = \sum_{i=1}^{m} e_i \otimes u_i,\ Y = \sum_{i=1}^{m} e_i \otimes y_i,\ F = \sum_{i=1}^{m} e_i \otimes f(u_i, y_i),$$

$$\frac{\partial F}{\partial U} = \sum_{i=1}^{m} e_i e_i^T \otimes \frac{\partial f}{\partial u_i}$$



  - Spatial DOFs for each sample stored consecutively

Ensemble Matrix-Vector Product

Sandia National Laboratories

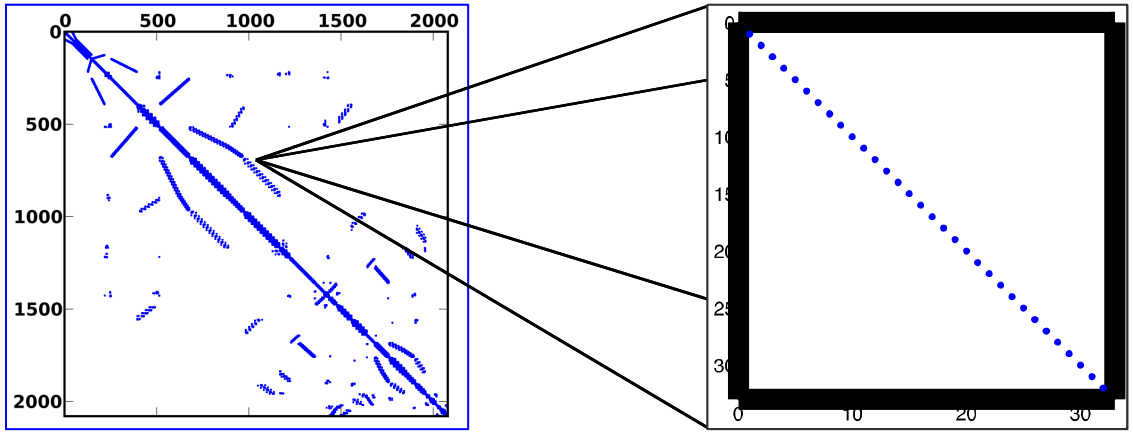
```
// Ensemble matrix-vector product
template <typename T, int m>
void ensemble_crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
   for (int e=0; e < m; ++e) {
      for (int row=0; i<A.num_rows; ++row) {
         const int entry_begin = A.row_map[row];
         const int entry_end   = A.row_map[row+1];
         T sum = 0.0;
         for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry + e*A.num_entries] * x[col + e*A.num_rows];
         }
         y[row + e*A.num_rows] = sum;
      }
   }
}
```

Simultaneous ensemble propagation

- Commute Kronecker products:

$$\tilde{F}(\tilde{U},\tilde{Y})=0,\ \tilde{U}=\sum_{i=1}^{m}u_i\otimes e_i,\ \tilde{Y}=\sum_{i=1}^{m}y_i\otimes e_i,\ \tilde{F}=\sum_{i=1}^{m}f(u_i,y_i)\otimes e_i,$$

$$\frac{\partial\tilde{F}}{\partial\tilde{U}}=\sum_{i=1}^{m}\frac{\partial f}{\partial u_i}\otimes e_i e_i^T$$



  - *m* sample values for each DOF stored consecutively

Commuted, Ensemble Matrix-Vector Product

Sandia National Laboratories

```
// Ensemble matrix-vector product using commuted layout
template <typename T, int m>
void ensemble_commuted_crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; i<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum[m];
        for (int e=0; e < m; ++e)
            sum[e] = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            for (int e=0; e < m; ++e) {
                sum[e] += A.values[entry*m + e] * x[col*m + e];
            }
        }
        for (int e=0; e < m; ++e)
            y[row*m + e] = sum[e];
    }
}
```

- Automatically reuse non-sample dependent data
- Sparse access latency amortized across ensemble
- Math on ensemble naturally maps to vector arithmetic
- Communication latency amortized across ensemble

C++ Ensemble Scalar Type

Sandia National Laboratories

```
// Ensemble scalar type
template <typename U, int m>
struct Ensemble {
    U val[m];
    Ensemble(const U& v) { for (int e=0; e<m; ++e) val[m] = v; }
    Ensemble& operator=(const Ensemble& a) {
        for (int e=0; e<m; ++e) val[m] = a.val[m];
        return *this;
    }
    Ensemble& operator+=(const Ensemble& a) {
        for (int e=0; e<m; ++e) val[m] += a.val[m];
        return *this;
    }
    // ...
};

template <typename U, int m>
Ensemble<U,m> operator*(const Ensemble<U,m>& a, const Ensemble<U,m>& b) {
    Ensemble<U,m> c;
    for (int e=0; e<m; ++e) c.val[e] = a.val[e]*b.val[e];
    return c;
}

// ...
```

Ensemble Matrix-Vector Product Through Operator Overloading

Sandia National Laboratories

- Original matrix-vector product routine, instantiated with T = Ensemble<double,m> scalar type:

```
// Serial Crs matrix-vector product for arbitrary floating-point type T
template <typename T>
void crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; row<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry] * x[col];
        }
        y[row] = sum;
    }
}
```

Stokhos: Trilinos Tools for Embedded UQ Methods

Sandia National Laboratories

- Provides ensemble scalar type
  - Uses expression templates to fuse loops

$$d = a \times b + c = \{a_1 \times b_1 + c_1, \ldots, a_m \times b_m + c_m\}$$

http://trilinos.org

- Enabled in simulation codes through template-based generic programming
  - Template C++ code on scalar type
  - Instantiate template code on ensemble scalar type
- Integrated with Kokkos (Edwards, Sunderland, Trott) for many-core parallelism
  - Specializes Kokkos data-structures, execution policies to map vectorization parallelism across ensemble
- Integrated with Tpetra-based solvers for hybrid (MPI+X) parallel linear algebra
  - Exploits templating on scalar type
  - Krylov solvers (Belos)
  - Algebraic multigrid preconditioners (MueLu)
  - Incomplete factorization, polynomial, and relaxation-based preconditioners/smoothers (Ifpack2)
  - Sparse-direct solvers (Amesos2)

Techniques Prototyped in FENL Mini-App*

Sandia National Laboratories

- Simple nonlinear diffusion equation

$$-\nabla \cdot (\kappa(x,y)\nabla u) + u^2 = 0,$$

$$\kappa(x,y) = \kappa_0 + \sigma \sum_{i=1}^{M} \sqrt{\lambda_i}\kappa_i(x) y_i$$

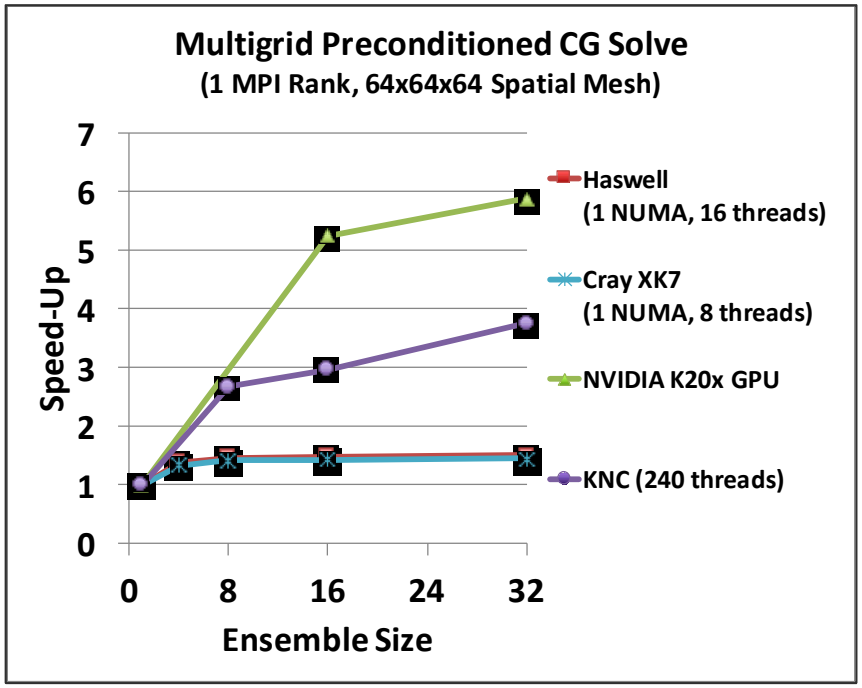
  - 3-D, linear FEM discretization
  - 1x1x1 cube, unstructured mesh
  - KL truncation of exponential random field model for diffusion coefficient
  - Trilinos-couplings package

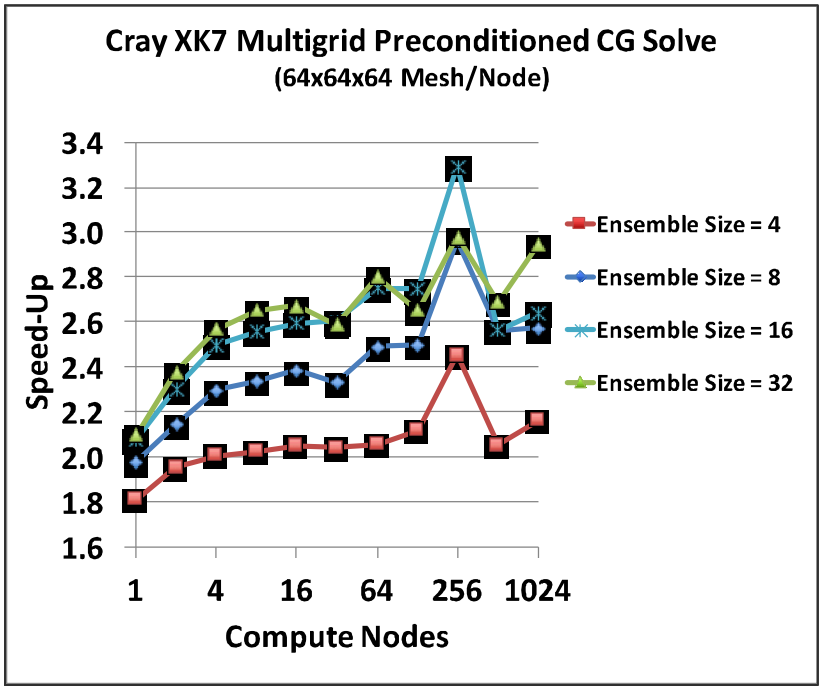
http://trilinos.org

- Hybrid MPI+X parallelism
  - Traditional MPI domain decomposition using threads within each domain
- Employs Kokkos for thread-scalable
  - Graph construction
  - PDE matrix/RHS assembly
- Employs Tpetra for distributed linear algebra
  - CG iterative solver (Belos package)
  - Smoothed Aggregation AMG preconditioning (MueLu)
- Supports embedded ensemble propagation via Stokhos through entire assembly and solve
  - Samples generated via local and global sparse grids (TASMANIAN)

*Phipps, et al, *Embedded Ensemble Propagation for Improving Performance, Portability and Scalability of Uncertainty Quantification on Emerging Computational Architectures*, SISC, 2017

Total Linear Solve Speed-up





$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$

- Smoothed-aggregation algebraic multigrid preconditioning (MueLu)
  - Chebyshev smoothers
  - Sparse-direct coarse-grid solver (Amesos2/Basker)
  - Multi-jagged parallel repartioning (Zoltan2)

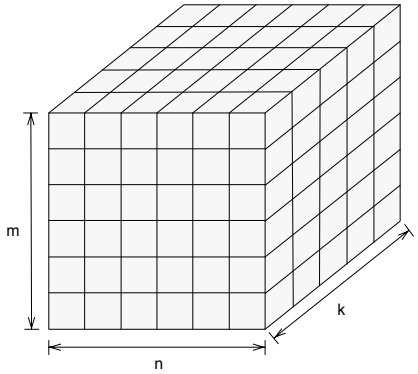
Ensemble Grouping

Sandia National Laboratories

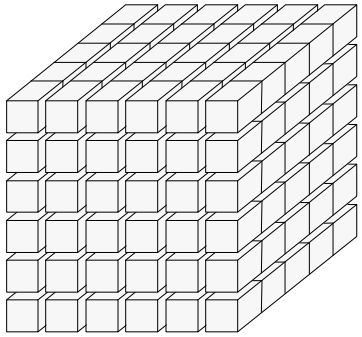
- For these problems, computational work driven by the number of (preconditioned) solver iterations
- Special case of “ensemble divergence”, where different samples in the ensemble would take diverging code paths
  - Biggest challenge for effective use of ensembles on hard problems
- Solution: group samples to minimize divergence
  - M. D’Elia, et al, Ensemble Grouping Strategies for Embedded Stochastic Collocation Methods Applied to Anisotropic Diffusion Problems, to appear in SIAM JUQ.
  - M. D’Elia, et al, Surrogate-based Ensemble Grouping Strategies for Embedded Stochastic Collocation Methods, submitted to SIAM JUQ, 2017 (arXiv: 1705.02003).

Block Line Preconditioner

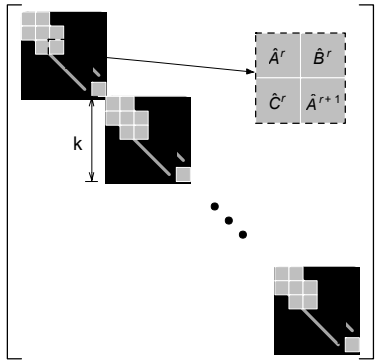
- Consider a block sparse system arising from coupled multi-physics problems.
- Line preconditioner is built by approximating the problem domain as a collection of lines of elements.
  - Block tridiagonal matrix.
  - Tridiagonal blocks factored and solved
- Block size b = 3, 5, 9 and 15, determined by physical problem
  - Poor vectorization within blocks



Problem domain



Extracted line elements



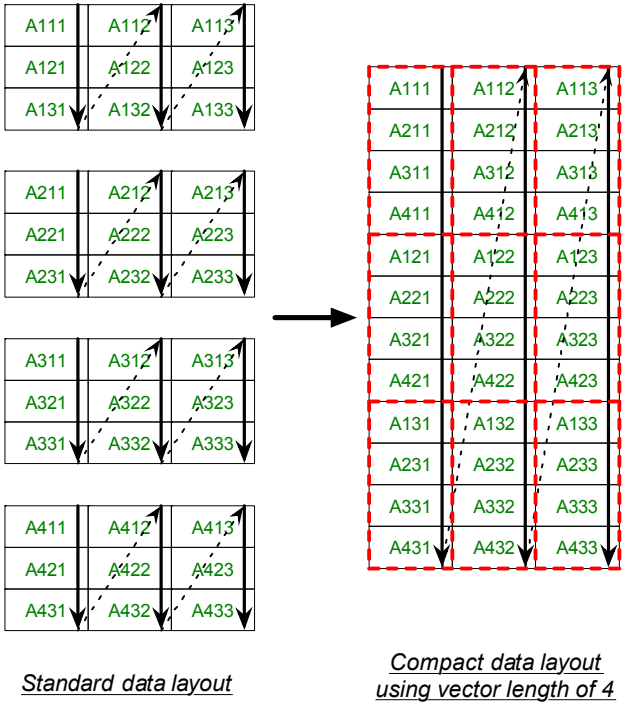
A set of block tridiagonal matrices

1 for T in $\{T_0, T_1, \cdots, T_{m\times n-1}\}$ do in parallel
2 for $r \leftarrow 0$ to $k-2$ do
3 $\hat{A}^r := LU(\hat{A}^r)$;
4 $\hat{B}^r := L^{-1}\hat{B}^r$;
5 $\hat{C}^r := \hat{C}^r U^{-1}$;
6 $\hat{A}^{r+1} := \hat{A}^{r+1} - \hat{C}^r \hat{B}^r$;
7 $\hat{A}^{k-1} := LU(\hat{A}^{k-1})$;

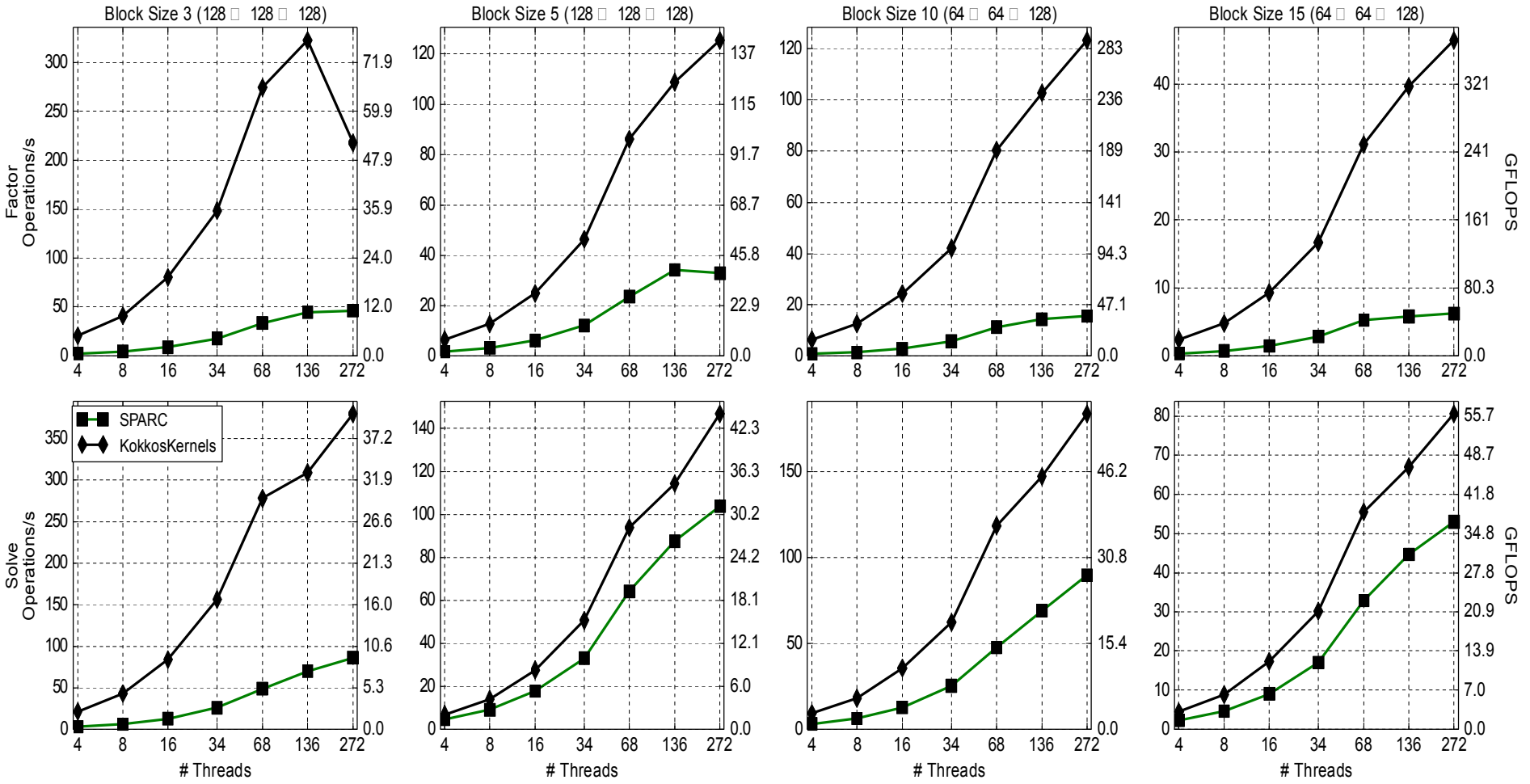
Compact Data Layout (SIMD type)

- Collection of matrices stored with SIMD scalar type
  - Interleaves data across matrices
  - SIMD type becomes basic computing unit and all scalar operations are transformed to vector operations.
- Compact LU, TRSM and GEMM are implemented aiming for small dense matrix problems in Kokkos-Kernels
  - https://github.com/kokkos/kokkos-kernels
  - Provide layered APIs that correspond to Kokkos hierarchical parallelism (batch parallelism incorporating locality)
  - In collaboration with Intel, compact batched BLAS is adopted as standard (available in Intel MKL 18).

```
// computing unit
struct Vector AVX256D {
  union {
    __m256d v;
    double s[4];
  };
};

// overload arithmetic operators (+-*/)
Vector AVX256D
operator +( Vector AVX256D const &a,
            Vector AVX256D const &b) {
  return __mm256_add_pd(a, b);
}
```

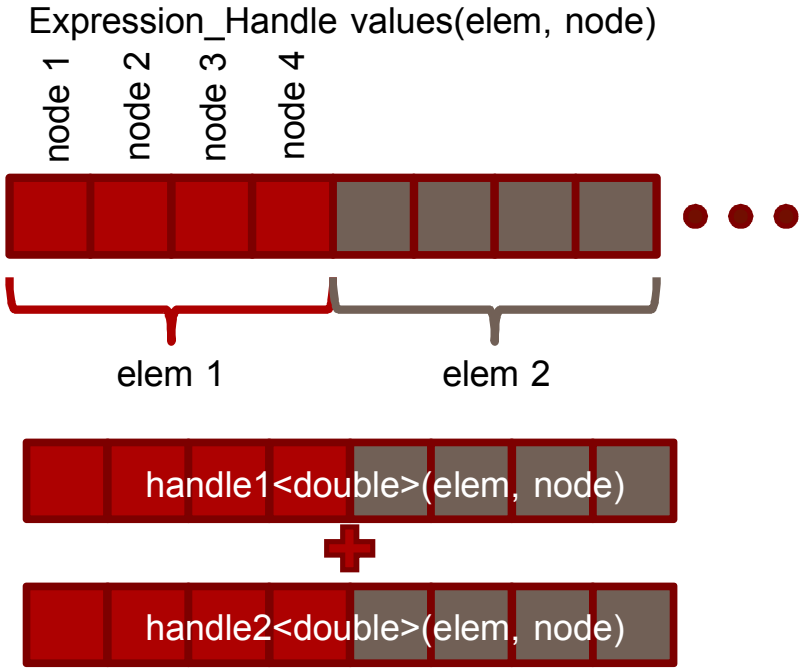


Performance Improvement on KNL

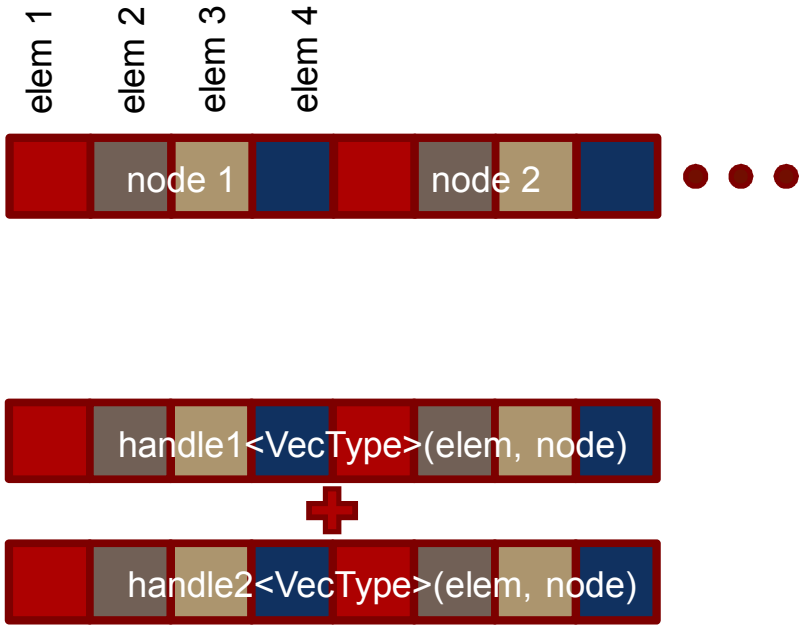


Vectorization for PDE assembly

- Vectorize across mesh cells for PDE residual/Jacobian evaluation using simd::Double scalar type
  - Source terms
  - Material models
  - …



4 elem x 4 node = 16 scalar additions



4-wide intrinsic: 4 elem x 4 node = 4 vector additions

Sierra SSE2/AVX/AVX512 interface

Sandia National Laboratories

Simd.h:

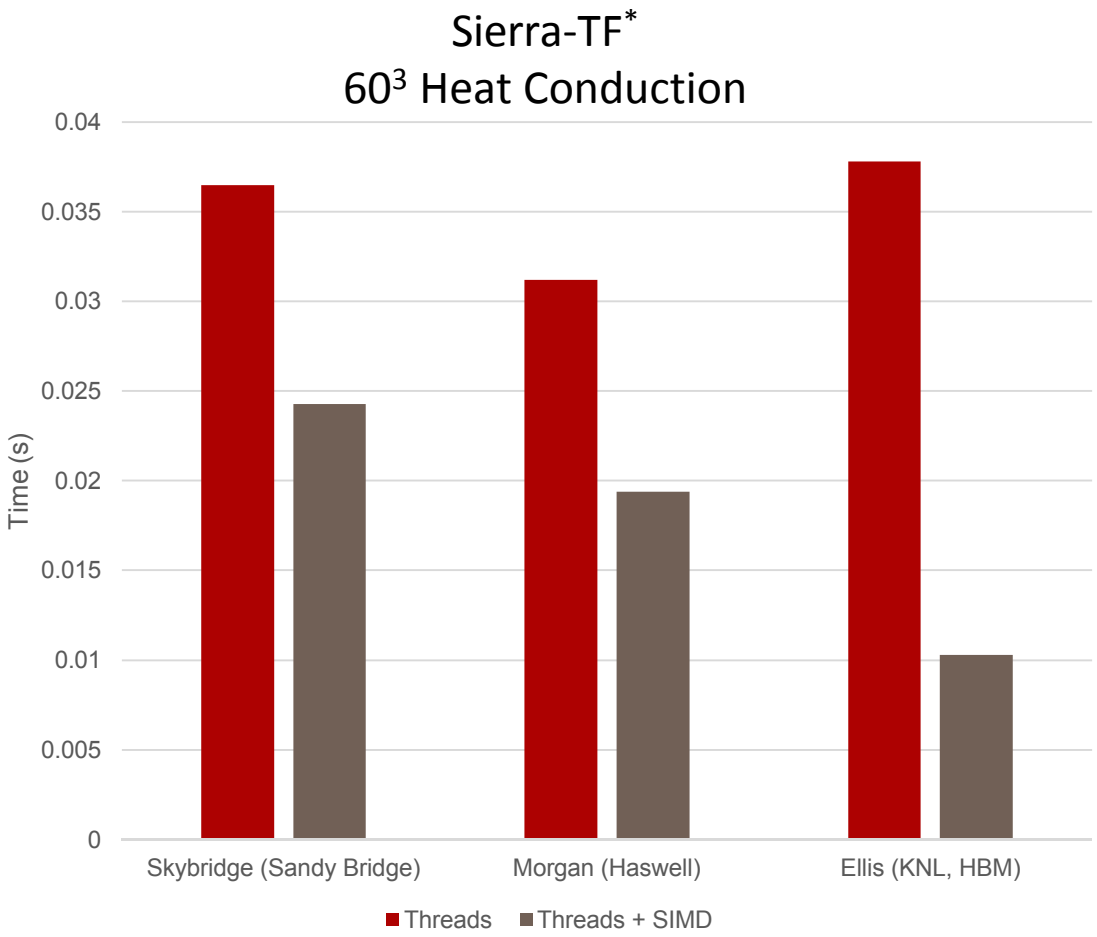
```
#if defined(AVX)
    const int num_doubles = 4;
    class simd::Double { __m256d d };
#elif defined(SSE2)
    const int num_doubles = 2;
    class simd::Double { __m128d d };
#else
    const int num_doubles = 1;
    typedef double simd::Double;
#end
```

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);
```

Real applications!

- 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)
  - ExaWind ECP project
  - 2-3x speedup on Haswell, KNL
- SPARC (2018)



*Courtesy of J. Clausen (1541)

Conclusions

Sandia National Laboratories

- SIMD scalar types provide a straightforward means of implementing outer-loop vectorization
  - Provides high-quality vectorization for even the most complex code
  - Doesn't rely on compiler's ability to do complex auto-vectorization
- Effective use of templates makes it easier to incorporate, but is not required
- Each code at Sandia that is ported to Kokkos finds they don't achieve good vectorization performance on KNL
  - Using the scalar-type approach is our path forward
- 3 SIMD scalar types presented here are being merged and made available in the Kokkos ecosystem
  - Should be done in early FY18

Acknowledgements

- This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

- This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) as well as the National Nuclear Security Administration, Advanced Technology Development and Mitigation program. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility.

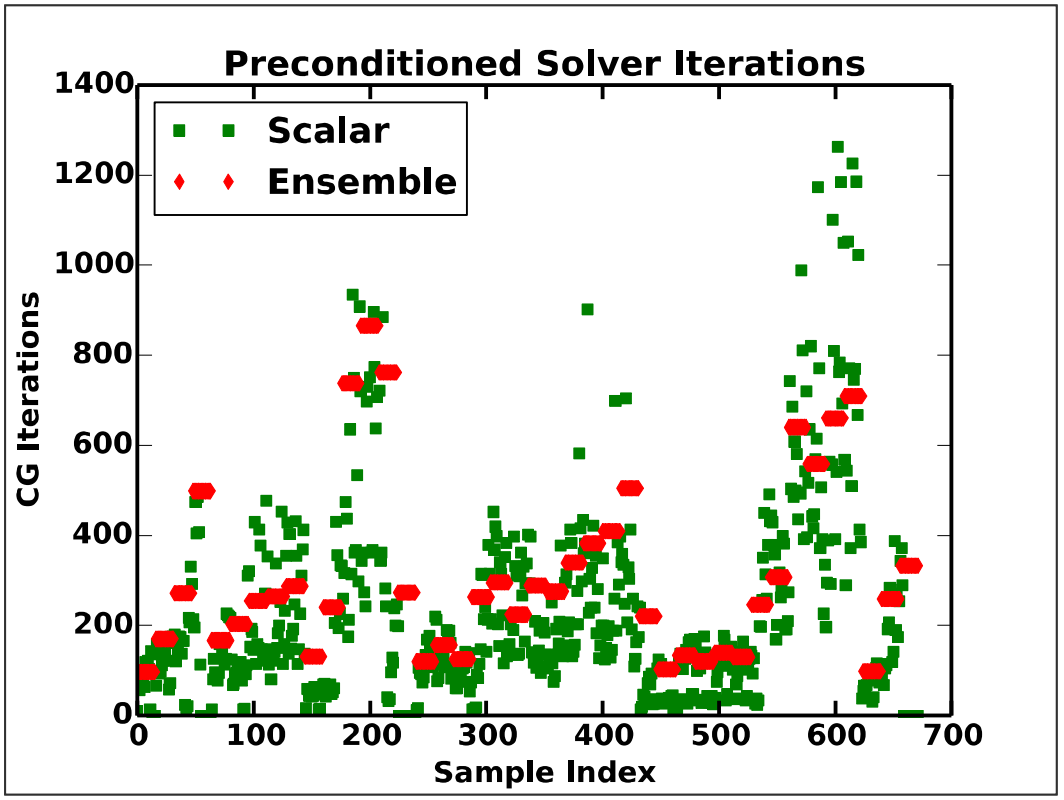
Sandia National Laboratories

Backup Slides

Highly Anisotropic Diffusion

$$-\nabla \cdot (K(x,y)\nabla u) + u^2 = 0,$$
$$K(x,y) = \mathrm{diag}(\kappa(x,y), 1, 1)$$
$$\kappa(x,y) = 1 + 100\exp(\sqrt{300}\sum_{i=1}^{M}\sqrt{\lambda_i}\kappa_i(x)y_i))$$

- Decision on how to group samples will strongly impact performance



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.

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!