SAND2014-15554PE

Realizing Exascale Performance for Uncertainty Quantification

Eric Phipps (etphipp@sandia.gov),
H. Carter Edwards, Jonathan Hu
Sandia National Laboratories
and
Clayton Webster
Oak Ridge National Laboratory

EX14 SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities

July 6, 2014

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.

Sandia National Laboratories

Can Exascale Solve the UQ Challenge?

- UQ means many things
  - Best estimate + uncertainty, model validation, model calibration, …
- A key to many UQ tasks is forward uncertainty propagation
  - Given uncertainty model of input data (aleatory, epistemic, …)
  - Propagate uncertainty to output quantities of interest
- There are many forward uncertainty propagation approaches
  - Monte Carlo, stochastic collocation, polynomial chaos, stochastic Galerkin, …
- Key challenge:
  - Accurately quantifying rare events and localized behavior in high-dimensional uncertain input spaces
  - Can easily require O(10^4-10^6) expensive forward simulations
  - Often can only afford O(10^2) on today's petascale machines

Achieving Exascale Performance Requires New Approaches

- UQ approaches usually implemented as an outer loop
  - Repeated calls of deterministic solver
  - Coarse-grained distributed memory parallelism over samples
  - Can we achieve a 1000x increase in available uncertainty propagation in this manner?

- No increase in clock-speed
  - Must increase parallelism

- Little increase in total node count, dramatic increase in node-level parallelism
  - Must evaluate multiple samples in parallel *on each node*

- Node memory increase of 0.1-0.01 of floating-point capacity
  - Parallel sample evaluations must share data when possible (threads)

- No decrease in latency, latency hiding through instruction-level parallelism & out-of-order execution replaced by hardware multi-threading and vectorization
  - Simulations must exhibit good data locality and expose sufficient fine-grained parallelism
  - Extremely challenging for many simulation algorithms, e.g., sparse linear algebra on complex (unstructured) domains

- UQ is a highly structured calculation
  - Expose new dimensions of fine-grained parallelism through *embedded* approaches
    - Embedded sample propagation
    - Stochastic Galerkin

Polynomial Chaos Expansions (PCE)

- **Steady-state finite dimensional model problem:**

$$\text{Find } u(\xi) \text{ such that } f(u,\xi) = 0,\ \xi : \Omega \to \Gamma \subset R^M, \text{ density } \rho$$

- **(Global) Polynomial Chaos approximation:**

$$u(\xi) \approx \hat{u}(\xi) = \sum_{i=0}^{P} u_i \psi_i(\xi), \quad \langle \psi_i \psi_j \rangle \equiv \int_\Gamma \psi_i(y)\psi_j(y)\rho(y)dy = \delta_{ij}\langle \psi_i^2 \rangle$$

  - Multivariate orthogonal polynomials
  - Typically constructed as tensor products with total order at most N
  - Can be adapted (anisotropic, local support)

- **Non-intrusive polynomial chaos (NIPC, NISP):**

$$u_i = \frac{1}{\langle \psi_i^2 \rangle} \int_\Gamma \hat{u}(y)\psi_i(y)\rho(y)dy \approx \frac{1}{\langle \psi_i^2 \rangle} \sum_{k=0}^{Q} w_k u^k \psi_i(y^k), \quad f(u^k, y^k) = 0$$

  - Sparse-grid quadrature methods for scalability to moderate stochastic dimensions
  - Need to be careful to ensure quadrature rule preserves discrete orthogonality
    - SPAM (Constantine, Eldred, Phipps, CMAME, 2012)
    - Method is equivalent to stochastic collocation

Simultaneous ensemble propagation

- **PDE:**

$$f(u,y)=0$$

- **Propagating *m* samples – block diagonal (nonlinear) system:**

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

- **Commute Kronecker products (just a reordering of DoFs):**

$$F_c(U_c,Y_c)=0,\quad U_c=\sum_{i=1}^{m} u_i\otimes e_i,\quad Y_c=\sum_{i=1}^{m} y_i\otimes e_i,\quad F_c=\sum_{i=1}^{m} f(u_i,y_i)\otimes e_i$$

- **Each sample-dependent scalar replaced by length-*m* array**
  - **Automatically reuse non-sample dependent data**
  - **Sparse accesses amortized across ensemble**
  - **Math on ensemble naturally maps to vector arithmetic**

Potential Speed-up for PDE Assembly

```
import($u$) // halo exchange

for $e = 0$ to $N_{elem}$ do

  // Sparse gather of global solution
  for $i = 0$ to $N_{node}$ do
    $I$ = NodeIndex($e$,$i$)
    $u_e(i) = u(I)$
  end for

  // Evaluate element residual/Jacobian
  $f_e$ = local_residual($u_e$)
  $J_e$ = local_jacobian($u_e$)

  // Sparse scatter into global residual/Jacobian
  for $i = 0$ to $N_{node}$ do
    $I$ = NodeIndex($e$,$i$)
    atomic_add($f(I)$, $f_e(i)$)
    for $j = 0$ to $N_{node}$ do
      $J$ = ElemGraph($e$,$i$,$j$)
      atomic_add($J(I,J)$, $J_e(i,j)$)
    end for
  end for
end for
```

- **Halo exchange**
  - **Amortize MPI latency across ensemble**
- **Gather**
  - **Reuse node-index map (mesh)**
  - **Replace sparse with contiguous loads**
- **Local residual/Jacobian**
  - **Vectorized math**
- **Scatter**
  - **Reuse node-index map and element graph (mesh)**
  - **Replace sparse with contiguous stores**

Potential Speed-up for Sparse Solvers

- Ingredients to sparse linear system solvers (CG, GMRES, …)
  - Sparse matrix-vector products

$$y(i) = \sum_{l=A.row(i)}^{A.row(i+1)} A.vals(l)x(A.col(l))$$

  - Dot-products
  - Preconditioners
    - Relaxation-based (Jacobi, Gauss-Seidel, …)
    - Incomplete factorizations (ILU, IC, …)
    - Polynomial (Chebyshev, …)
    - Multilevel (Algebraic/Geometric multigrid)

- Sparse matrix-vector products
  - Amortize MPI latency in halo exchange
  - Reuse matrix graph
  - Replace sparse with contiguous loads
  - Vector arithmetic

- Dot-products
  - Amortize MPI latency

- Preconditioners
  - Sparse mat-vecs
  - Sparse factorizations/triangular-solves
  - Smaller, more unstructured matrices

Stokhos: Trilinos Tools for Embedded UQ Methods

- Provides "ensemble scalar type"
  - C++ class containing an array with length fixed at compile-time
  - Overloads all math operations by mapping operation across array

$$a = \{a_1, \ldots, a_m\}, \quad b = \{b_1, \ldots, b_m\}, \quad c = a \times b = \{a_1 \times b_1, \ldots, a_m \times b_m\}$$

  - 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.sandia.gov

- Enabled in simulation codes through template-based generic progamming
  - 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
  - For CUDA, currently requires manual modification of parallel launch to use customized execution policies

- Integrated with Tpetra-based solvers for hybrid (MPI+X) parallel linear algebra
  - Exploits templating on scalar type
  - Optimized linear algebra kernels for ensemble scalar type
  - Krylov solvers (Belos), Incomplete factorization preconditioners (Ifpack2), algebraic multigrid preconditioners (MueLu)

Techniques Prototyped in FENL Mini-App

- Simple nonlinear diffusion equation

$$-\kappa \nabla^2 u + u^2 = 0$$

  - 3-D, linear FEM discretization
  - 1x1x1 cube, unstructured mesh
  - KL-like random field model for diffusion coefficient

http://trilinos.sandia.gov

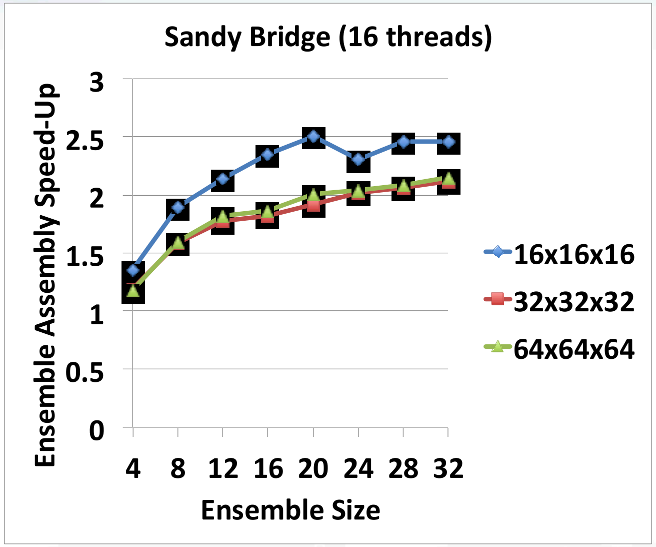
- Hybrid MPI+X parallelism
  - Traditional MPI domain decomposition using threads within each domain

- Employs Kokkos for thread-scalable
  - Graph construction
  - PDE assembly

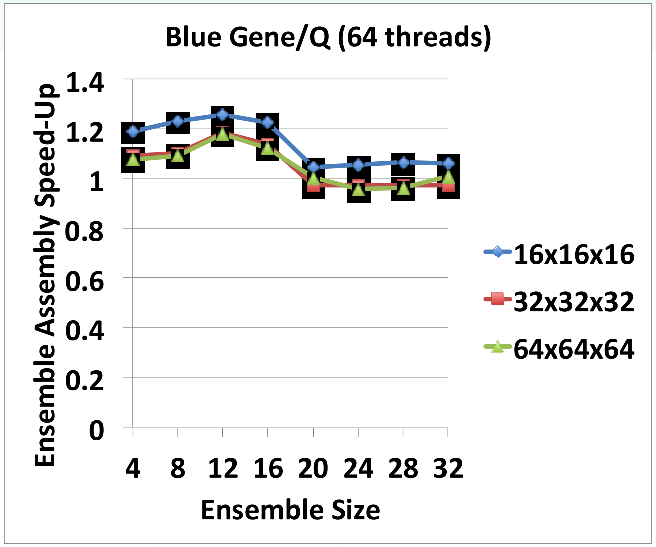
- Employs Tpetra for distributed linear algebra
  - CG iterative solver (Belos package)
  - Smoothed Aggregation AMG preconditioning (MueLu)

- Supports embedded ensemble propagation via Stokhos through entire assembly and solve
  - Samples generated via tensor product & Smolyak sparse grid quadrature

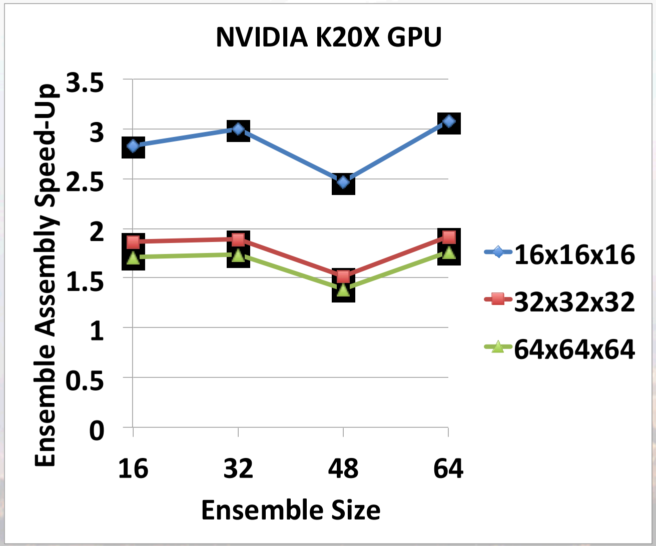
Ensemble Assembly Speed-Up



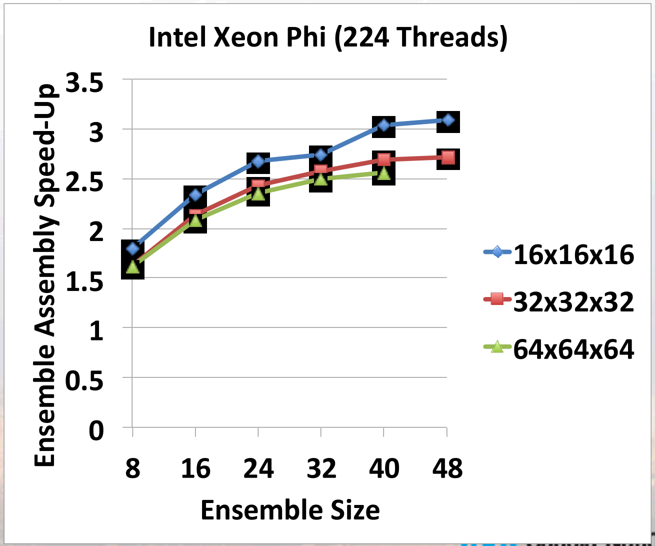
Sandy Bridge (16 threads)



Blue Gene/Q (64 threads)

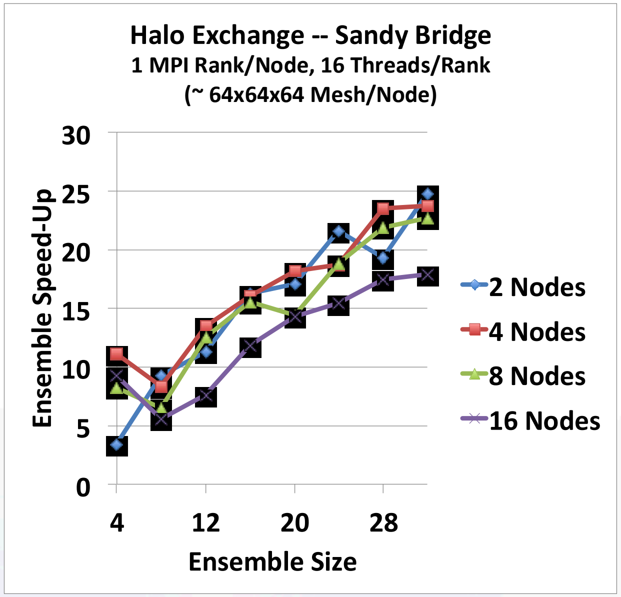


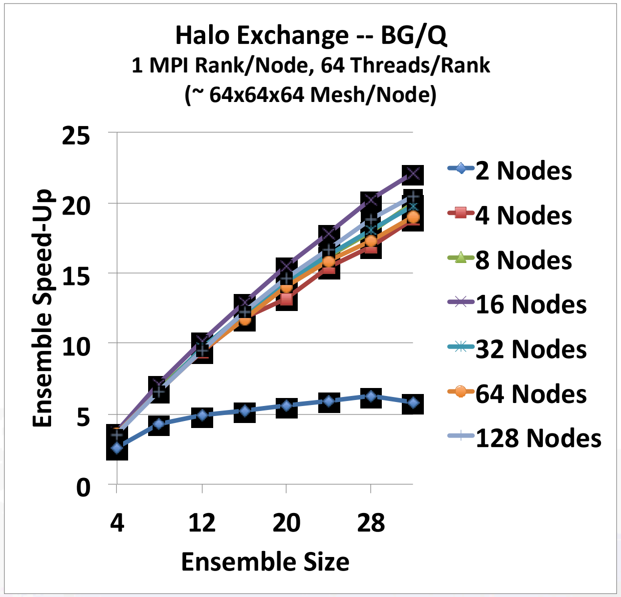
NVIDIA K20X GPU



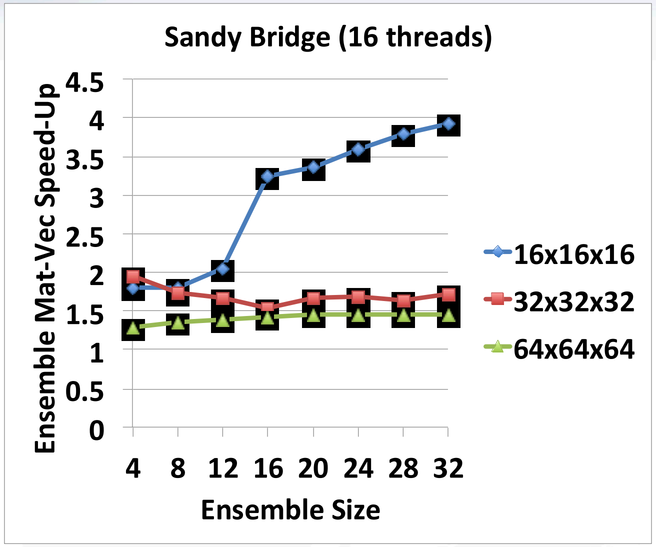
Intel Xeon Phi (224 Threads)

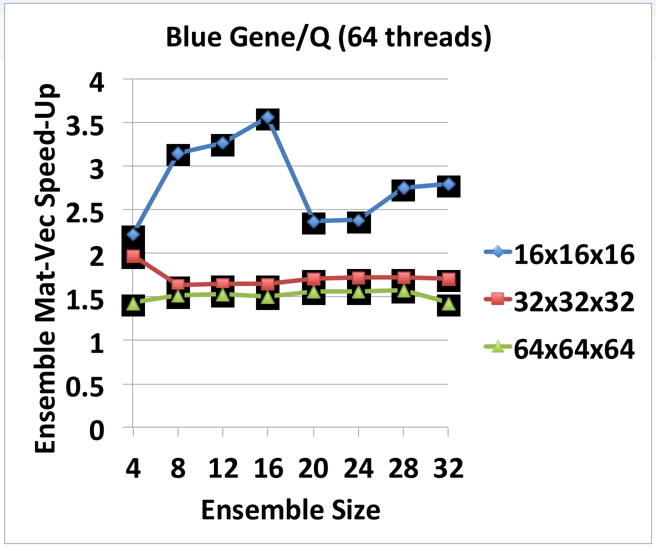
Ensemble MPI Halo-Exchange Speed-Up

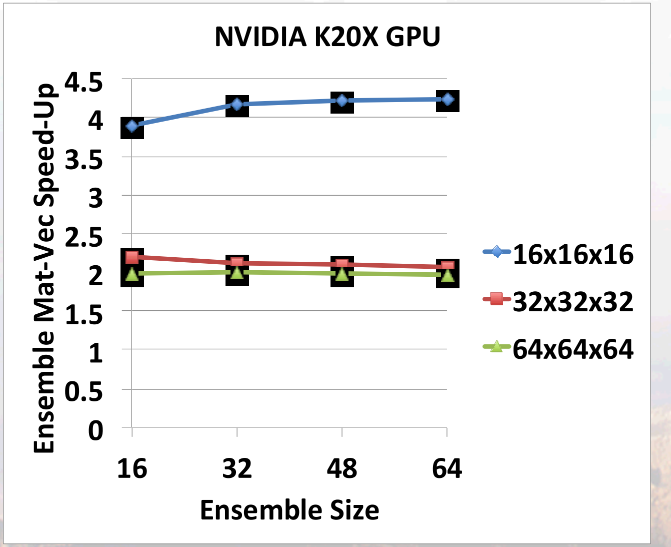


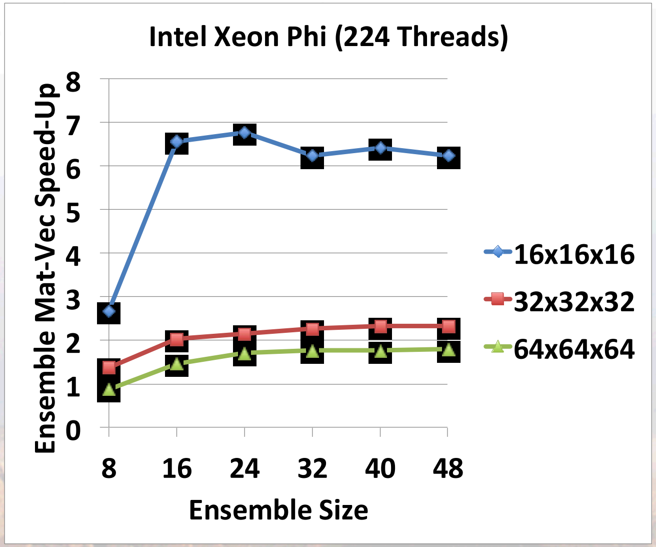


Ensemble Matrix-Vector Product Speed-Up

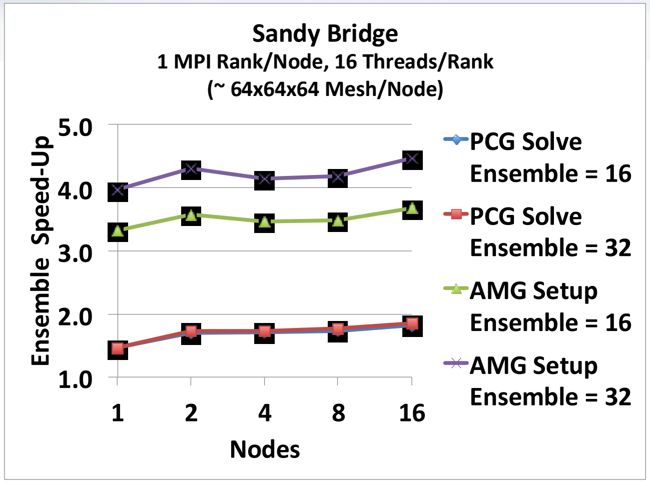


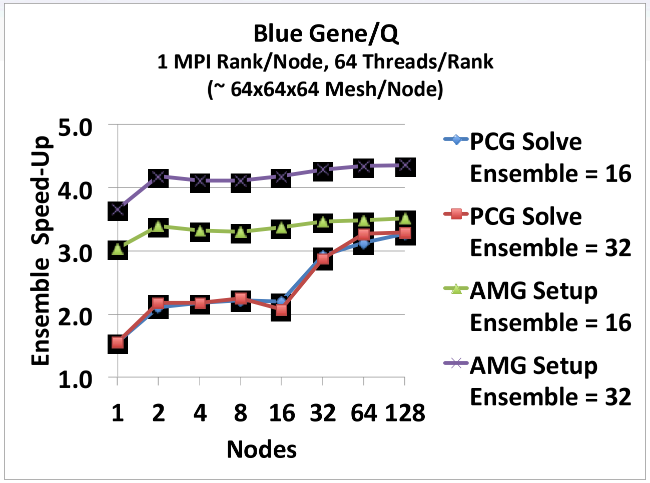


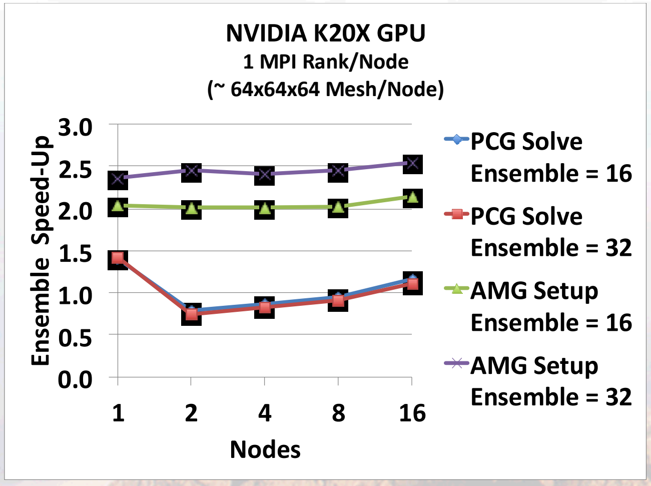


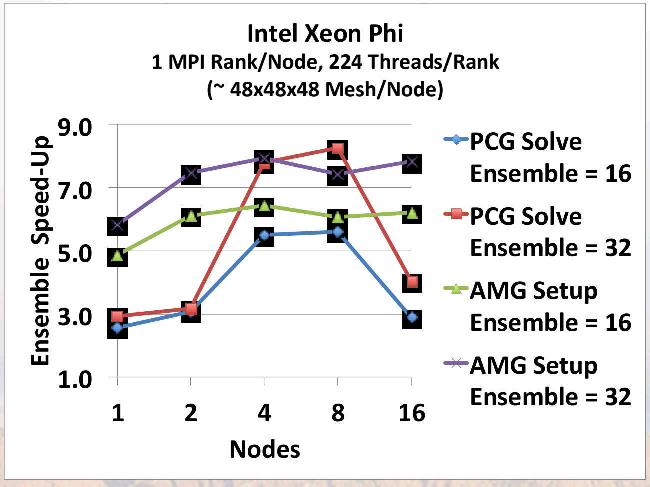


Ensemble AMG-Preconditioned CG Speed-Up


Sandy Bridge
1 MPI Rank/Node, 16 Threads/Rank
(~ 64x64x64 Mesh/Node)


Blue Gene/Q
1 MPI Rank/Node, 64 Threads/Rank
(~ 64x64x64 Mesh/Node)


NVIDIA K20X GPU
1 MPI Rank/Node
(~ 64x64x64 Mesh/Node)


Intel Xeon Phi
1 MPI Rank/Node, 224 Threads/Rank
(~ 48x48x48 Mesh/Node)

Problem with current implementation that will be fixed soon

Embedded Stochastic Galerkin UQ Methods

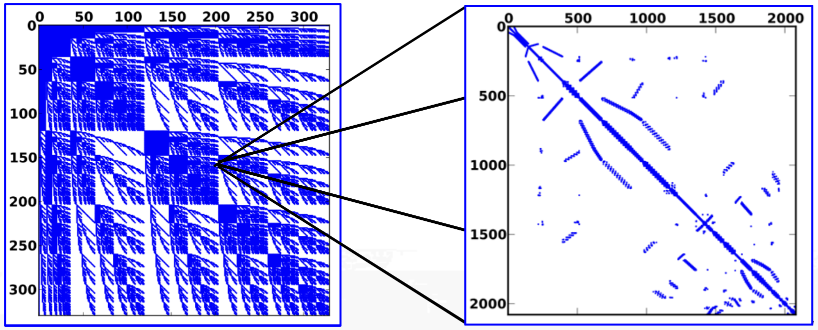
- Stochastic Galerkin method (Ghanem and many, many others…):

$$\hat{u}(\xi) = \sum_{i=0}^{P} u_i \psi_i(\xi) \to f_i(u_0, \ldots, u_P) \equiv \frac{1}{\langle \psi_i^2 \rangle} \int_\Gamma f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \ldots, P$$

- Method generates new coupled spatial-stochastic nonlinear problem (intrusive)

$$F(U) = 0, \quad U = \sum_{i=1}^{P} e_i \otimes u_i, \quad F = \sum_{i=1}^{P} e_i \otimes f_i$$

$$\frac{\partial F}{\partial U} \approx \sum_{k=0}^{P} G_k \otimes A_k, \quad G_k(i,j) \equiv C_{ijk} \equiv \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$$



Stochastic sparsity

Spatial sparsity

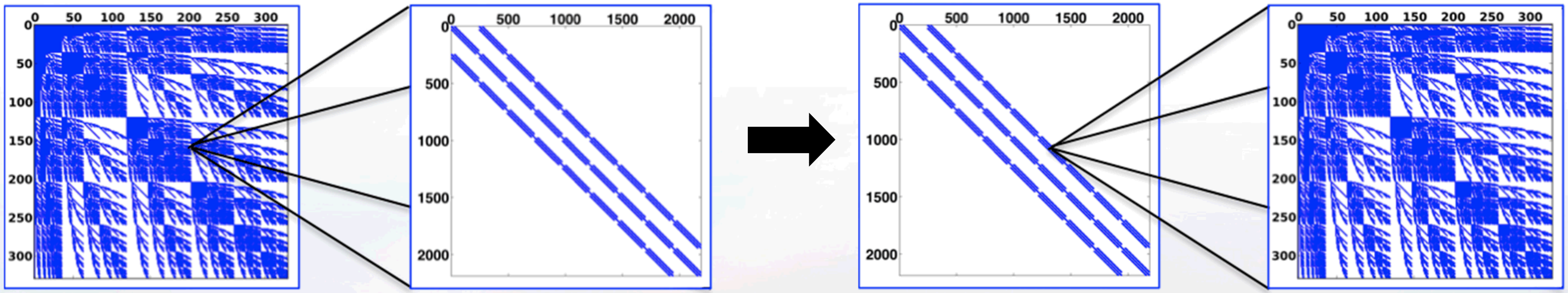
- Advantages:
  - Many fewer stochastic degrees-of-freedom for comparable level of accuracy
- Challenges:
  - Computing SG residual and Jacobian entries in large-scale, production simulation codes
  - Solving resulting systems of equations efficiently, particularly for nonlinear problems
- Stokhos package provides tools for implementing SG methods for large-scale systems
  - Integrates with Kokkos, Tpetra for multicore, MPI parallelism
  - Techniques demonstrated in FENL

Structure of Galerkin Operator

- Operator traditionally organized with outer-stochastic, inner-spatial structure
  - Allows reuse of deterministic solver data structures and preconditioners
  - Makes sense for sparse stochastic discretizations

$$A^{trad} = \sum_{k=0}^{P} G_k \otimes A_k$$

$$A^{com} = \sum_{k=0}^{P} A_k \otimes G_k$$



Stochastic sparsity

Spatial sparsity

Spatial sparsity

Stochastic sparsity

- For nonlinear problems, makes sense to commute this layout to outer-spatial, inner-stochastic
  - Leverage emerging architectures to handle denser stochastic blocks

Commuted SG Matrix Multiply

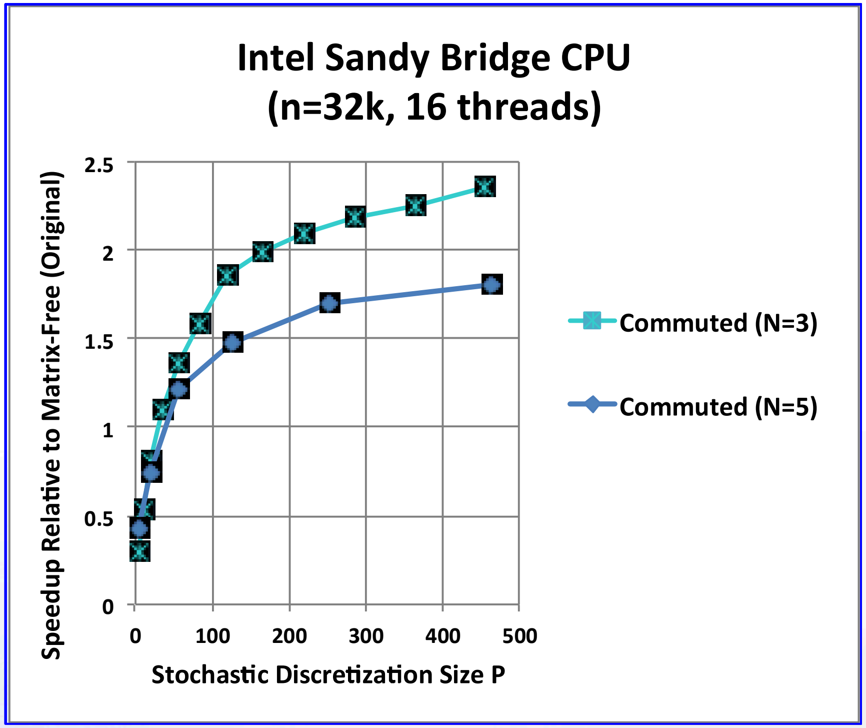
$$Y^{com} = A^{com} X^{com} \implies \sum_{i=0}^{P} y_i \otimes e_i = \left(\sum_{k=0}^{P} A_k \otimes G_k \right) \left(\sum_{j=0}^{P} x_j \otimes e_j \right)$$

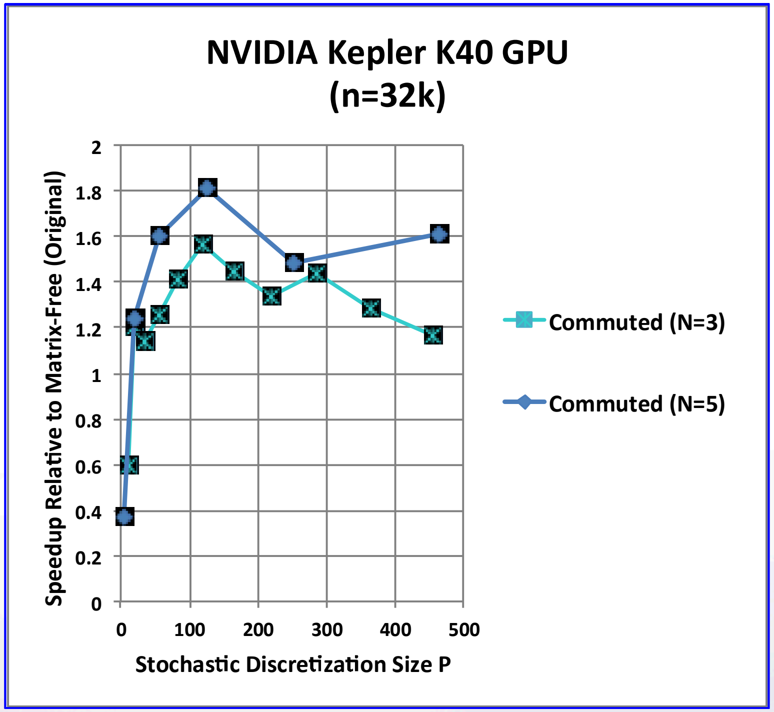
- Two level algorithm
  - Outer: sparse (CRS) matrix-vector multiply algorithm
  - Inner: sparse stochastic Galerkin product

$$\aleph_A(l) = \{m \mid A_0(l,m) \neq 0\} \qquad \aleph_C(i) = \{(j,k) \mid C(i,j,k) \neq 0\}$$

$$\overset{\text{stochastic basis}}{\underset{\text{FEM basis}}{y(i,l)}} = \underset{\text{FEM bases sum}}{\sum_{m\in\aleph_A(l)}} \overset{\text{stochastic bases sum}}{\sum_{(j,k)\in\aleph_C(i)}} \overset{\text{stochastic basis}}{\underset{\text{FEM basis}}{A(k,l,m)}} \overset{\text{stochastic basis}}{\underset{\text{FEM basis}}{x(j,m)}} \overset{\text{triple product}}{C(i,j,k)}$$

Commuted SG Mat-Vec Speed-Up





- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach over original algorithm
  - Performance driven by reading Cijk tensor from memory

Challenges and Opportunities

- Significant effort to refactor simulation codes
  - Codes will likely be refactored anyway for exascale
  - Introduce abstraction at scalar level through *template-based generic programming*

- Solvers/preconditioners optimized for embedded uncertainty propagation
  - Effective stochastic Galerkin preconditioners
  - Reuse preconditioning/solver information across ensemble array
    - Whole preconditioner
    - Reuse multi-grid hierarchy/aggregrates
    - Recycle Krylov bases

- Increased cache pressure
  - Can't make SG/ensemble array too big

- Memory access patterns of SG Cijk tensor
  - Partitioning, balancing, reordering for cache
  - Can we generate it "on-the-fly" without reading from global memory (potentially eliminate bandwidth limit)?

- Propagating samples together requires commonality in solution process
  - Often need to refine UQ discretization near localized behavior/discontinuities/bifurcations
  - How to group samples to exploit commonality when you have it, and separate samples when you don't?

Auxiliary Slides

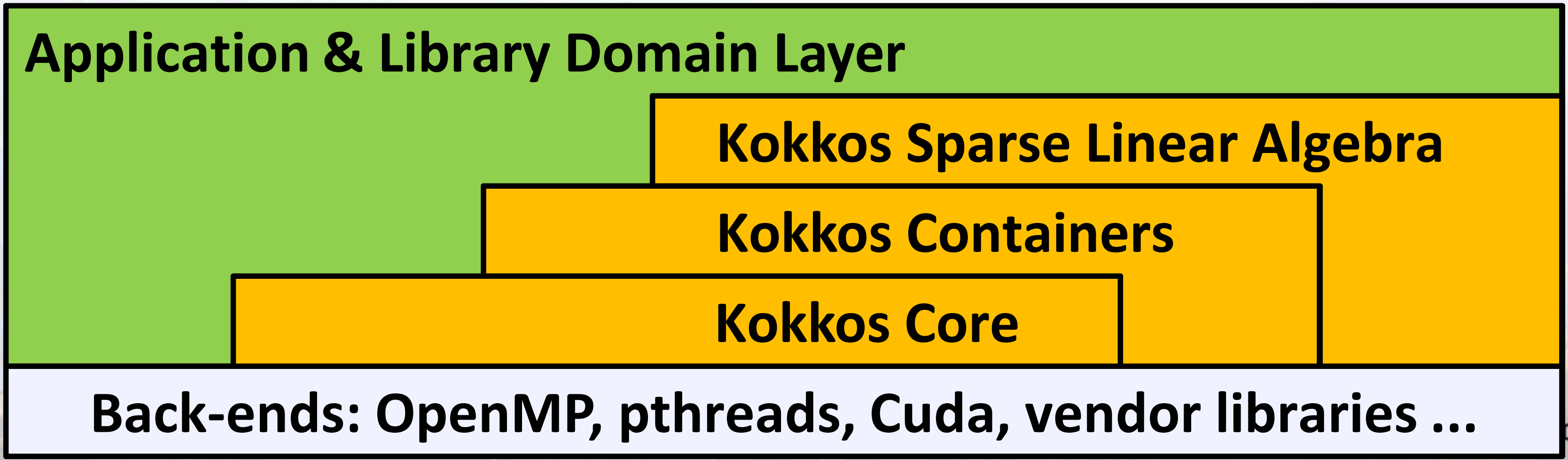
Kokkos: A Manycore Device Performance Portability Library for C++ HPC Applications[*]

- Standard C++ library, not a language extension
  - Core: multidimensional arrays, parallel execution, atomic operations
  - Containers: Thread-scalable implementations of common data structures (vector, map, CRS graph, …)
  - LinAlg: Sparse matrix/vector linear algebra

- Relies heavily on C++ template meta-programming to introduce abstraction without performance penalty
  - Execution spaces (CPU, GPU, …)
  - Memory spaces (Host memory, GPU memory, scratch-pad, texture cache, …)
  - Layout of multidimensional data in memory
  - Scalar type

http://trilinos.sandia.gov

[*] H.C. Edwards, D. Sunderland, C. Trott (SNL)


Application & Library Domain Layer
Kokkos Sparse Linear Algebra
Kokkos Containers
Kokkos Core
Back-ends: OpenMP, pthreads, Cuda, vendor libraries …

Tpetra: Foundational Layer / Library for Sparse Linear Algebra Solvers on Next-Generation Architectures*

- Tpetra: Sandia's templated C++ library for distributed memory (MPI) sparse linear algebra
  - Builds distributed memory linear algebra on top of Kokkos library
  - Distributed memory vectors, multi-vectors, and sparse matrices
  - Data distribution maps and communication operations
  - Fundamental computations: axpy, dot, norm, matrix-vector multiply, ...
  - Templated on "scalar" type: float, double, automatic differentiation, polynomial chaos, ensembles, …

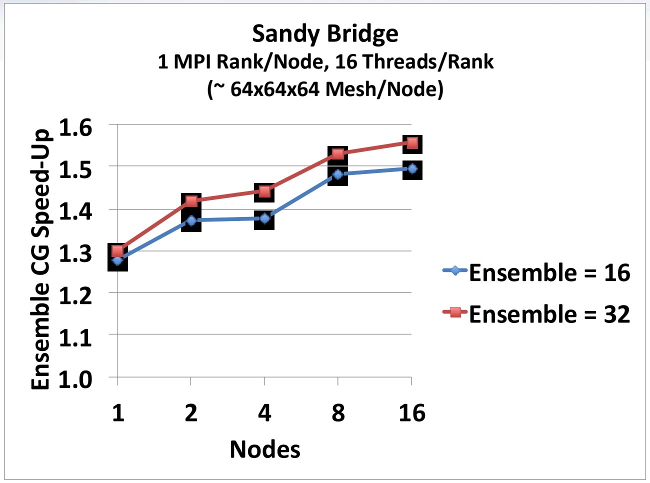
Trilinos

http://trilinos.sandia.gov

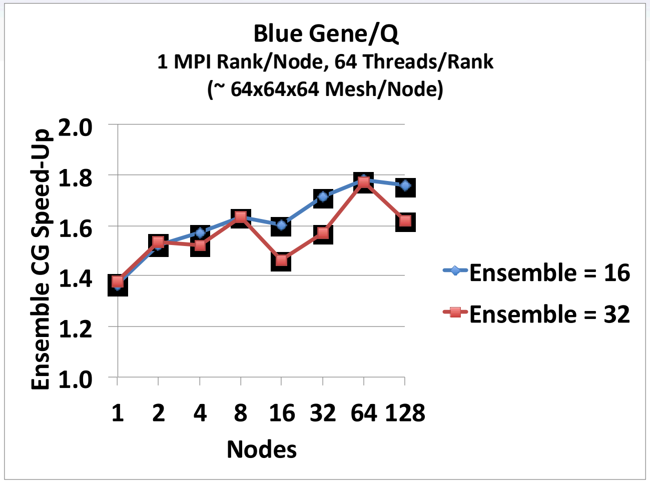
- Higher level solver libraries built on Tpetra
  - Preconditioned iterative algorithms (Belos)
  - Incomplete factorization preconditioners (Ifpack2, ShyLU)
  - Multigrid solvers (MueLu)
  - All templated on the scalar type

*M. Heroux, M. Hoemmen, *et al* (SNL)

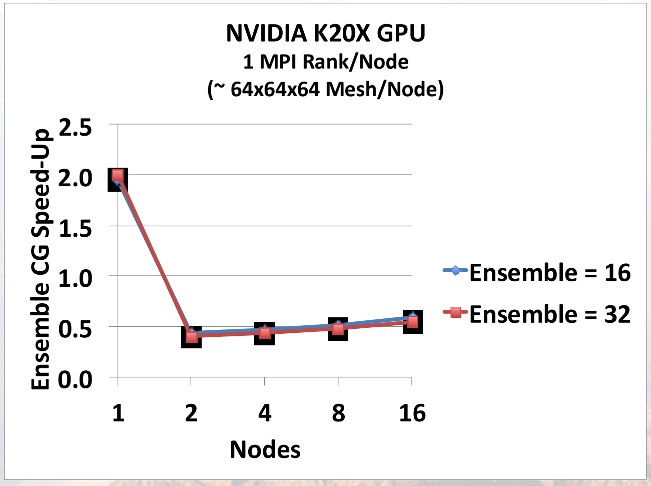
Ensemble CG Speed-Up



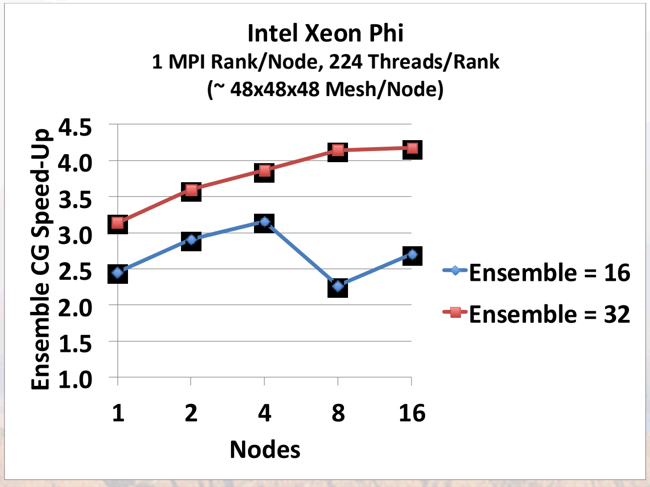
Sandy Bridge
1 MPI Rank/Node, 16 Threads/Rank
(~ 64x64x64 Mesh/Node)



Blue Gene/Q
1 MPI Rank/Node, 64 Threads/Rank
(~ 64x64x64 Mesh/Node)



NVIDIA K20X GPU
1 MPI Rank/Node
(~ 64x64x64 Mesh/Node)



Intel Xeon Phi
1 MPI Rank/Node, 224 Threads/Rank
(~ 48x48x48 Mesh/Node)

Problem with current implementation
that will be fixed soon

SG Linear Systems

- **Stochastic Galerkin Jacobian:**

$$\frac{\partial F}{\partial U} \approx A = \sum_{k=0}^{P} G_k \otimes A_k, \quad A_k = \frac{1}{\langle \psi_k^2 \rangle} \int_\Gamma \frac{\partial f}{\partial u}(\hat{u}(y), y)\psi_k(y)\rho(y)dy, \quad G_k(i,j) = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$$

- **Stochastic Galerkin Newton linear systems:**

$$A\Delta U = -F \implies \left(\sum_{k=0}^{P} G_k \otimes A_k\right)\left(\sum_{k=0}^{P} e_k \otimes \Delta u_k\right) = -\sum_{k=0}^{P} e_k \otimes f_k, \quad e_k = I(:,k) \in \mathbb{R}^{P+1}$$

- **Solution methods:**
  - Form SG matrix directly (expensive)
  - "Matrix-free" approach for iterative linear solvers:

$$Y = AX \implies \sum_{i=0}^{P} e_i \otimes y_i = \left(\sum_{k=0}^{P} G_k \otimes A_k\right)\left(\sum_{j=0}^{P} e_j \otimes x_j\right)$$

$$\implies y_i = \sum_{j=0}^{P}\sum_{k=0}^{P} A_k x_j C_{ijk}, \quad C_{ijk} = G_k(i,j) = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$$

  - Sparsity determined by triple product tensor
  - Only requires operator-apply for each operator PCE coefficient
  - Organize algorithm to minimize operator-vector applies

Multicore-CPU: One-level Concurrency

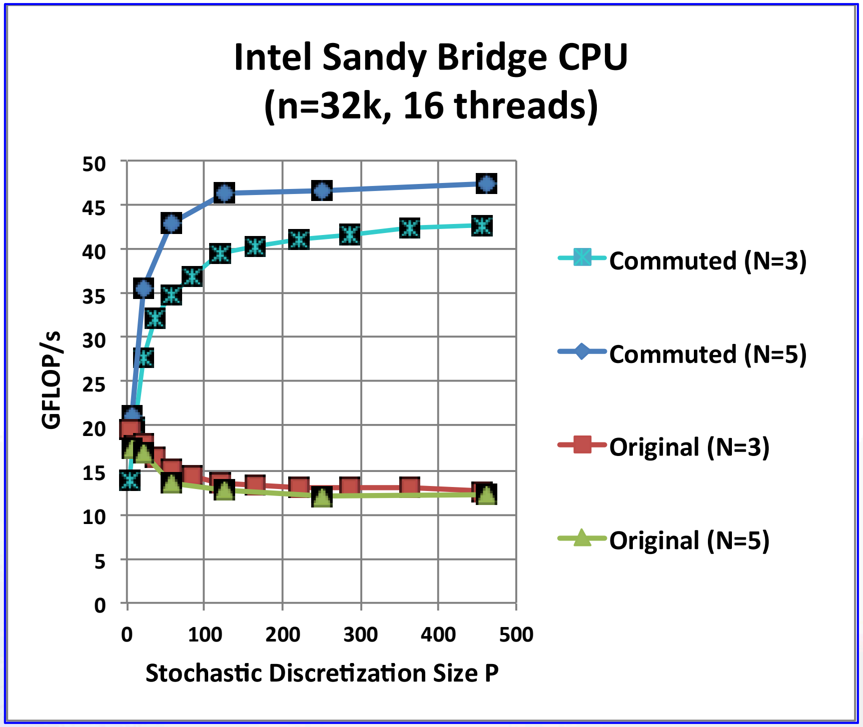
$$y(i,l) = \sum_{m\in\aleph_A(l)} \sum_{(j,k)\in\aleph_C(i)} A(k,l,m)x(j,m)C(i,j,k)$$

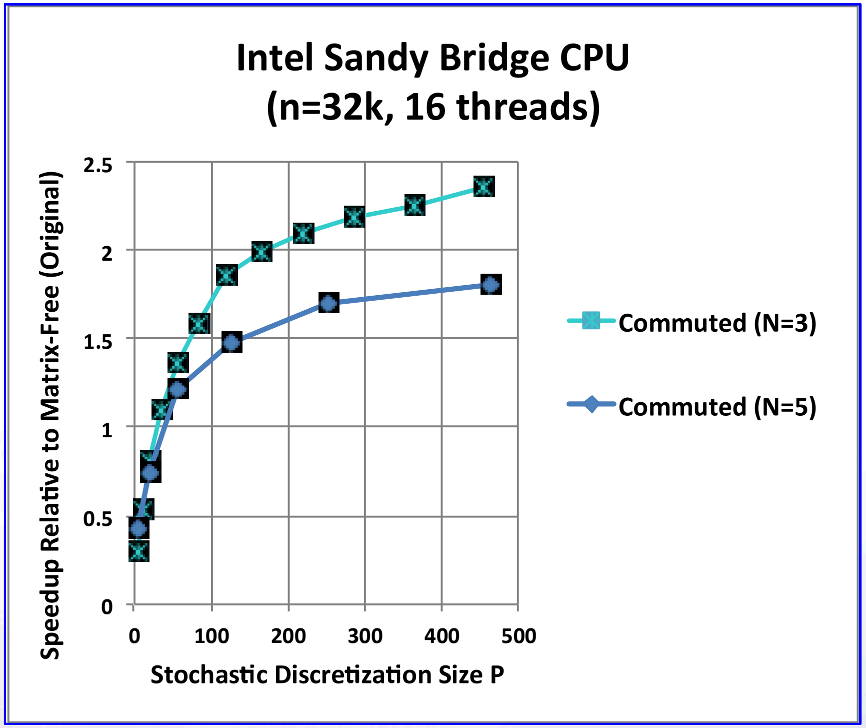
thread parallel

SIMD within a multicore-CPU thread

- **Each FEM row "owned" by a CPU thread**
  - **2 rows per core on Sandy Bridge**
- **Owning CPU thread computes** $y(:,l)$
  - **(j,k) loop vectorized (auto-vectorization or intrinsics) for SIMD parallelism**
  - **Vector width = 4 (AVX) on Sandy Bridge**

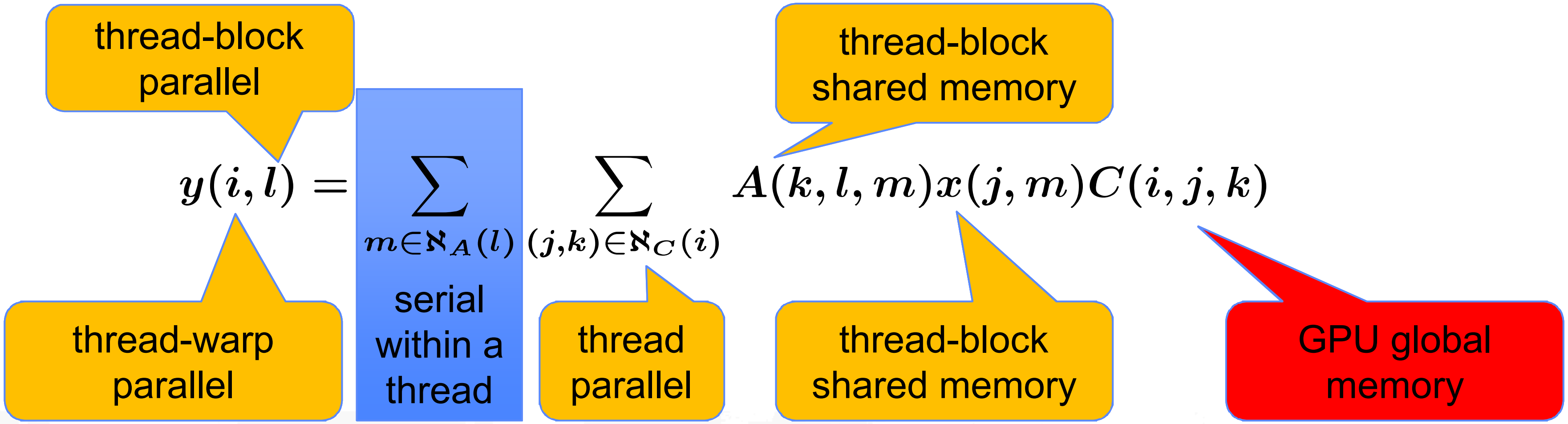
Intel Sandy Bridge CPU





- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach over original algorithm

Manycore-GPU: Two-level Concurrency



$$y(i,l) = \sum_{m\in\aleph_A(l)} \sum_{(j,k)\in\aleph_C(i)} A(k,l,m)x(j,m)C(i,j,k)$$

thread-block parallel

thread-warp parallel

serial within a thread

thread parallel

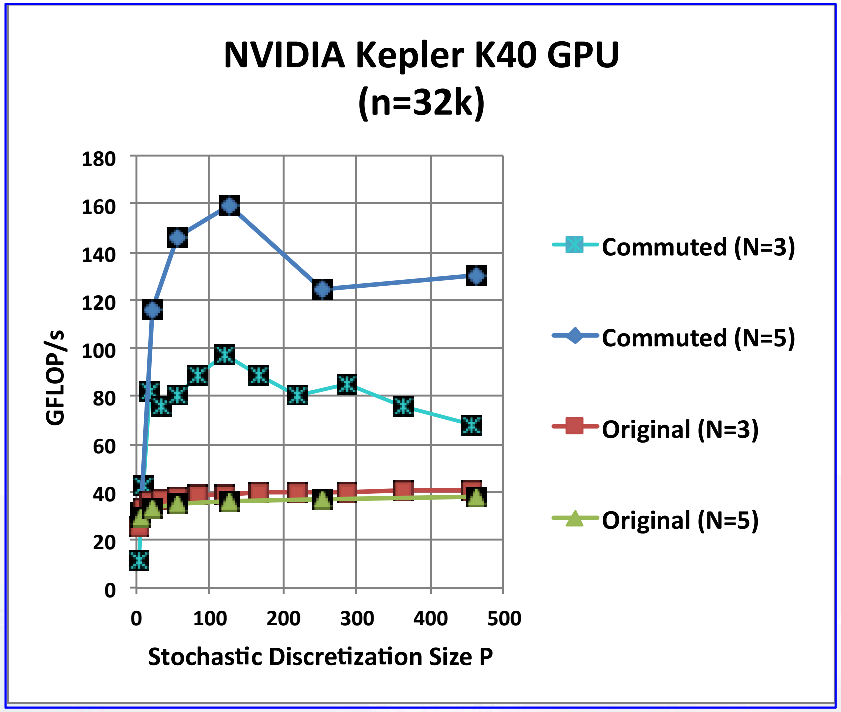
thread-block shared memory

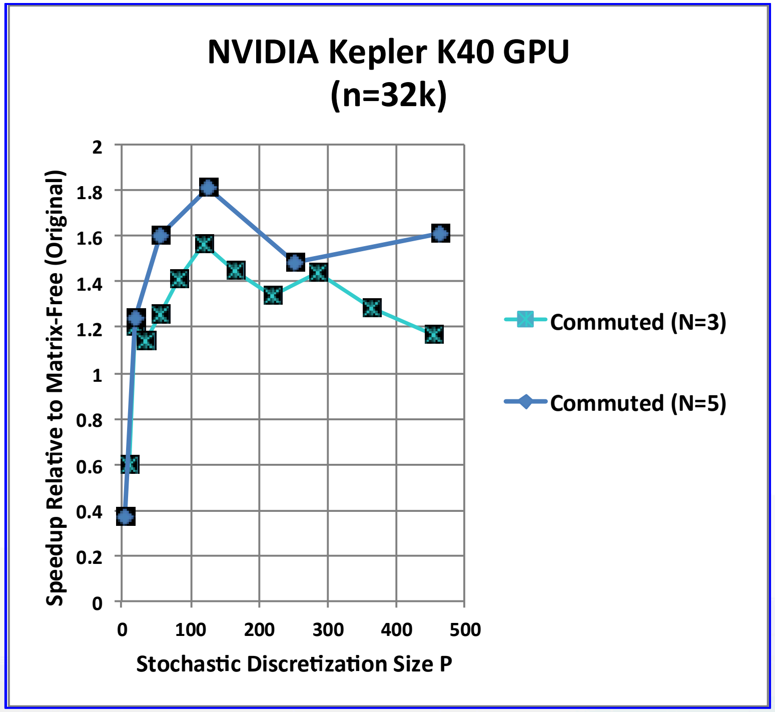
thread-block shared memory

GPU global memory

- **Multiple levels of concurrency:**
  - **Each FEM row owned by a thread-block**
  - **Each warp within a thread-block owns an "i"**
  - **Warps within a thread perform SG multiply in parallel, executing FEM multiply loop serially**
- **Sparse tensor stored in GPU global memory**
  - **Reduce sparse tensor reads by blocking FEM column loop ("m" loop)**
  - **Heuristic to choose block size based on stochastic discretization size to balance shared memory usage (reduces occupancy) and tensor reads**
  - **Pack (i,j) indices into single 32-bit integer**

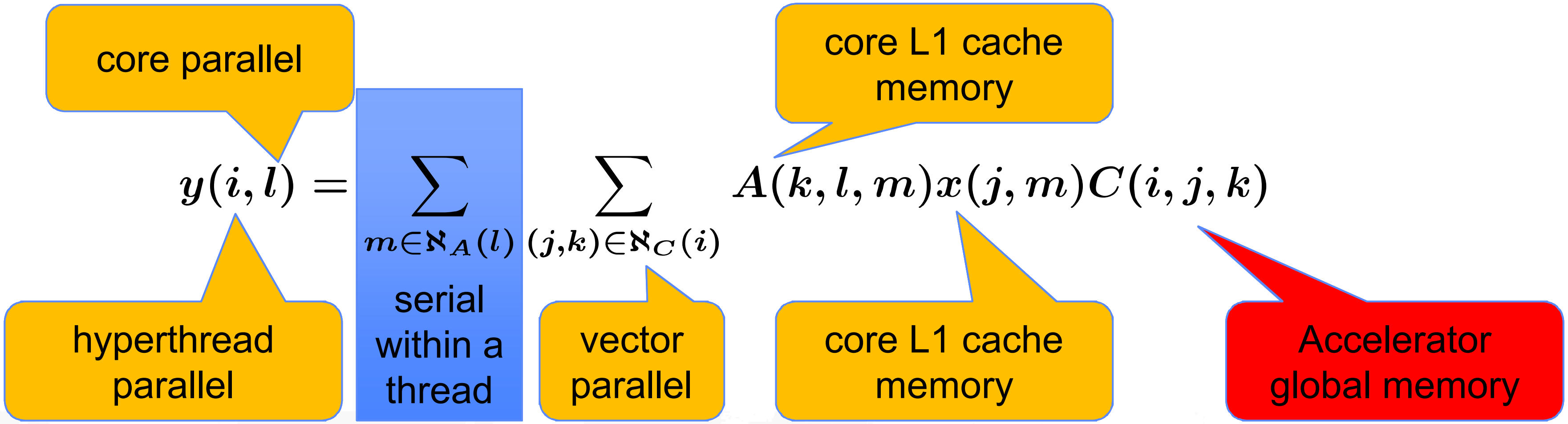
NVIDIA K40 GPU





- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach except for larger stochastic discretizations
  - Too much shared memory usage per CUDA block reduces occupancy

Manycore-Accelerator: Two-level Concurrency



$$y(i,l) = \sum_{m\in\aleph_A(l)} \sum_{(j,k)\in\aleph_C(i)} A(k,l,m)x(j,m)C(i,j,k)$$

core parallel

core L1 cache memory

hyperthread parallel

serial within a thread

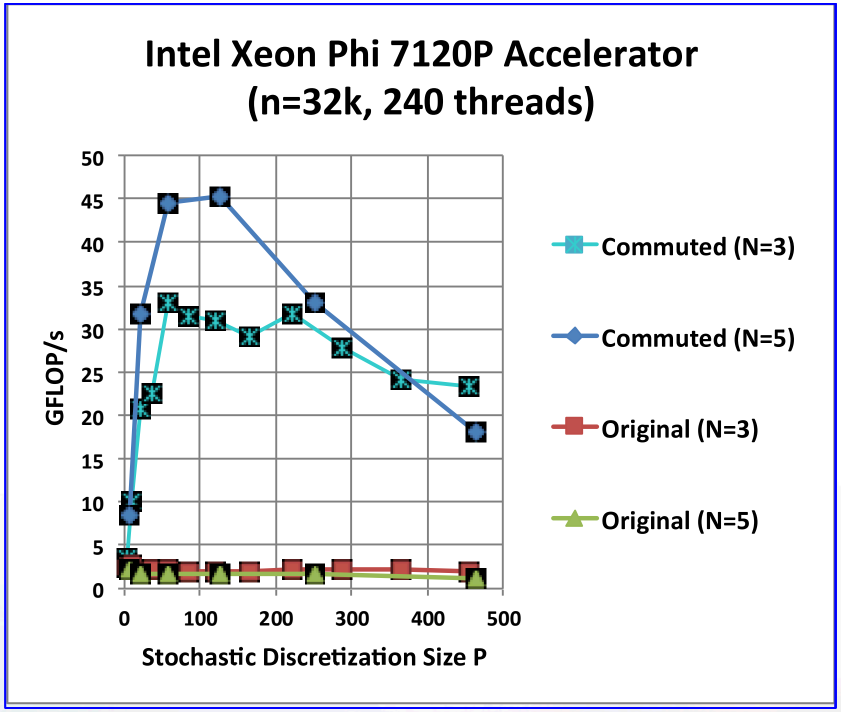
vector parallel

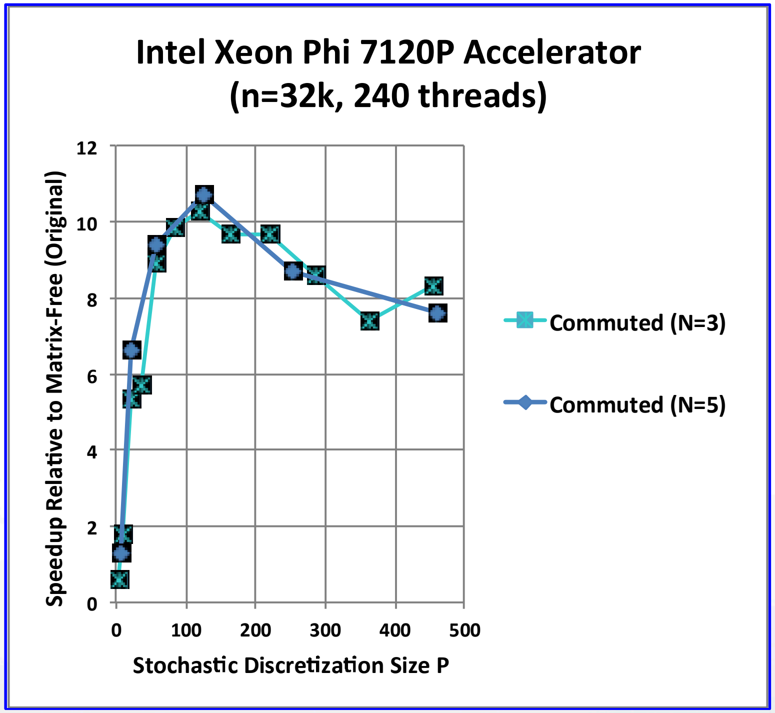
core L1 cache memory

Accelerator global memory

- **Map GPU to accelerator architecture:**
  - **GPU thread -> vector lane**
  - **Thread warp -> hyperthread**
  - **Thread block -> core**
- **Use essentially same algorithm as for GPU, except**
  - **Automatic caching of A, x entries instead of shared-memory loads**
  - **Fixed block size for blocking of FEM column loop (“m” loop)**
  - **No packing of (i,j) indices**

Intel Xeon Phi 7120P Accelerator





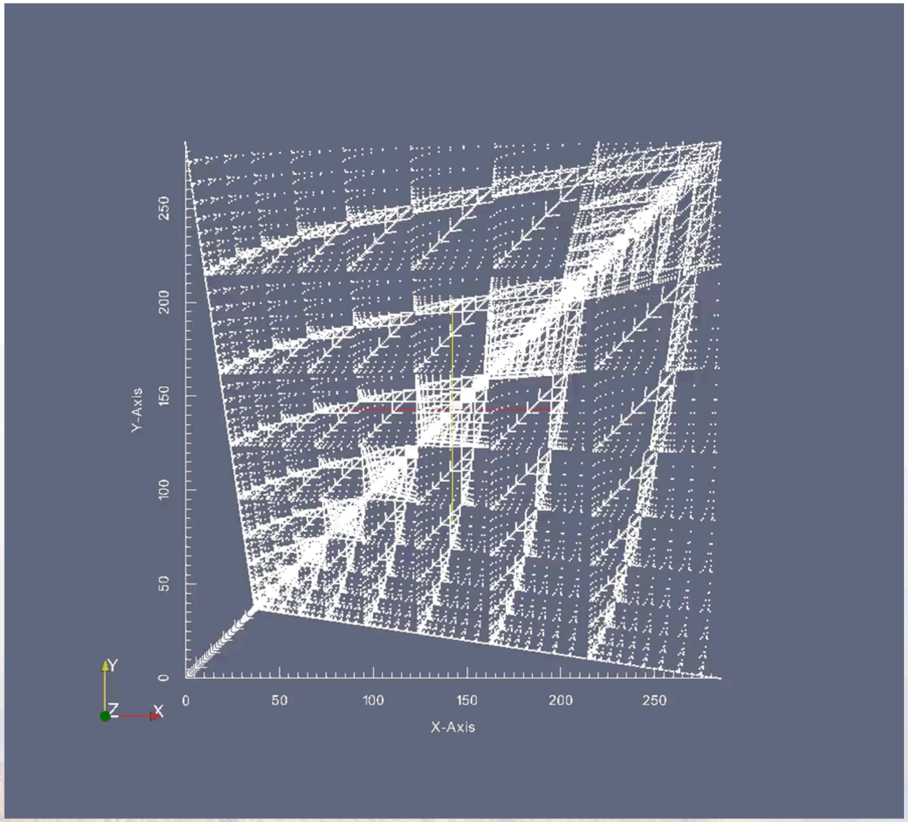
- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach except for larger stochastic discretizations
  - Calculation falls out of L1 cache

Sandia National Laboratories

Performance driven by C(i,j,k) tensor

$$y(i,l) = \sum_{m \in \aleph_A(l)} \sum_{(j,k) \in \aleph_C(i)} A(k,l,m)x(j,m)C(i,j,k)$$

- Precompute and store C
- Given l,m, load A(:,l,m), y(:,l), x(:,m) into cache
- Iterate over non-zero C(i,j,k) entries
- Sparse accesses of A, x, but in fast cache
  - Very fast for GPU
- Lots of reuse of A, x entries
- Can load A, x for multiple values of l,m to reduce reads of C


Y-Axis
X-Axis
Y
Z
X