SAND2013-4606C

Exploring Emerging Manycore Architectures for Uncertainty Quantification Through Embedded Stochastic Galerkin Methods

Eric Phipps (etphipp@sandia.gov),
H. Carter Edwards and Jonathan Hu, Jakob Ostient
Sandia National Laboratories

The Mathematics of Finite Elements and Applications

June 11-14, 2013

SAND 2013-xxxx

Forward UQ

- UQ means many things
  - Best estimate + uncertainty
  - Model validation
  - Model calibration
  - Reliability analysis
  - Robust design/optimization
  - …
- 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
  - NISP/NIPC
  - Regression PCE (“point/probabilistic collocation”)
  - Stochastic Galerkin
  - …
- Key challenges:
  - Achieving good accuracy
  - High dimensional uncertain spaces
  - Expensive forward simulations

Emerging Architectures Motivate New Approaches

- UQ approaches usually implemented as an outer loop
  - Repeated calls of deterministic solver
- Single-point forward simulations use very little available node compute power (unstructured, implicit)
  - 3-5% of peak FLOPS on multi-core CPUs (P. Lin, Charon, RedSky)
  - 2-3% on contemporary GPUs (Bell & Garland, 2008)
- Emerging architectures leading to dramatically increased on-node compute power
  - Not likely to translate into commensurate improvement in forward simulation
  - Many simulations/solvers don't contain enough fine-grained parallelism
- Can this be remedied by inverting the outer UQ/inner solver loop?
  - Add new dimensions of parallelism through *embedded* approaches

Outline

- Polynomial chaos-based UQ approaches
  - Non-intrusive spectral projection (NISP/NIPC)
  - Stochastic Galerkin (SG)
- Tools for implementing SG methods in large-scale PDE codes
- Application to model 3-D mechanics problems
- Reordering SG mat-vecs for contemporary multicore architectures

Polynomial Chaos Expansions (PCE)

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

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

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

$$u(\xi) \approx \hat{u}(\xi) = \sum_{i=0}^{P} u_i \Psi_i(\xi), \quad \langle \Psi_i \Psi_j \rangle \equiv \int_\Gamma \Psi_i(x)\Psi_j(x)\rho(x)dx = \delta_{ij}\langle \Psi_i^2 \rangle$$

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

$$u_i = \frac{1}{\langle \Psi_i^2 \rangle} \int_\Gamma \hat{u}(x)\Psi_i(x)\rho(x)dx \approx \frac{1}{\langle \Psi_i^2 \rangle} \sum_{k=0}^{Q} w_k \bar{u}_k \Psi_i(x_k), \quad f(\bar{u}_k, x_k) = 0$$

  - **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**

Embedded Stochastic Galerkin UQ Methods

- **Steady-state stochastic problem (for simplicity):**

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

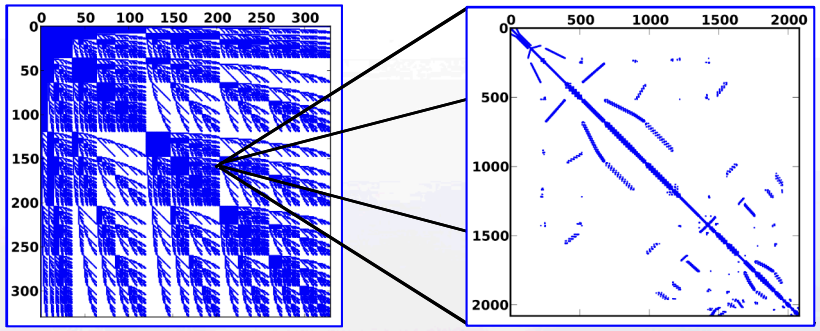
- **Stochastic Galerkin method (Ghanem and many, many others…):**

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

  - **Multivariate orthogonal basis of total order at most N – (generalized polynomial chaos)**

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

$$0 = F(U) = \begin{bmatrix} F_0 \\ F_1 \\ \vdots \\ F_P \end{bmatrix}, \quad U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_P \end{bmatrix} \quad \frac{\partial F}{\partial U} :$$



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**

Traditional Approach for SG Operator

- **Galerkin equations and Jacobian blocks:**

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

$$\frac{\partial F_i}{\partial u_j} = \frac{1}{\langle \psi_i^2 \rangle} \int_\Gamma \frac{\partial f}{\partial u}(\hat{u}(y), y) \psi_i(y) \psi_j(y) \rho(y) dy \approx \sum_{k=0}^{P} J_k \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle} \implies \frac{\partial F}{\partial U} = \sum_{k=0}^{P} G_k \otimes J_k,$$

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

  - **Polynomial chaos expansion of the deterministic Jacobian operator.**

- **This is used to implement matrix-vector products without forming the SG Jacobian explicitly (matrix-free):**

$$\frac{\partial F_i}{\partial u_j} \approx \sum_{k=0}^{P} J_k \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle} \implies \left(\frac{\partial F}{\partial U} V \right)_i = \sum_{j=0}^{P} \sum_{k=0}^{P} J_k v_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$$

- **Sparsity determined by triple products** $C_{ijk} = G_k(i,j) = \dfrac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$
  - **Symmetric for orthonormal bases**

Stokhos: Trilinos Tools for Embedded Stochastic Galerkin UQ Methods

- Eric Phipps, Chris Miller, Habib Najm, Bert Debusschere, Omar Knio

http://trilinos.sandia.gov

- Tools for describing SG discretization
  - Stochastic bases, quadrature rules, etc…
- C++ operator overloading library for automatically evaluating SG residuals and Jacobians
  - Replace low-level scalar type with orthogonal polynomial expansions
  - Leverages Trilinos Sacado automatic differentiation library

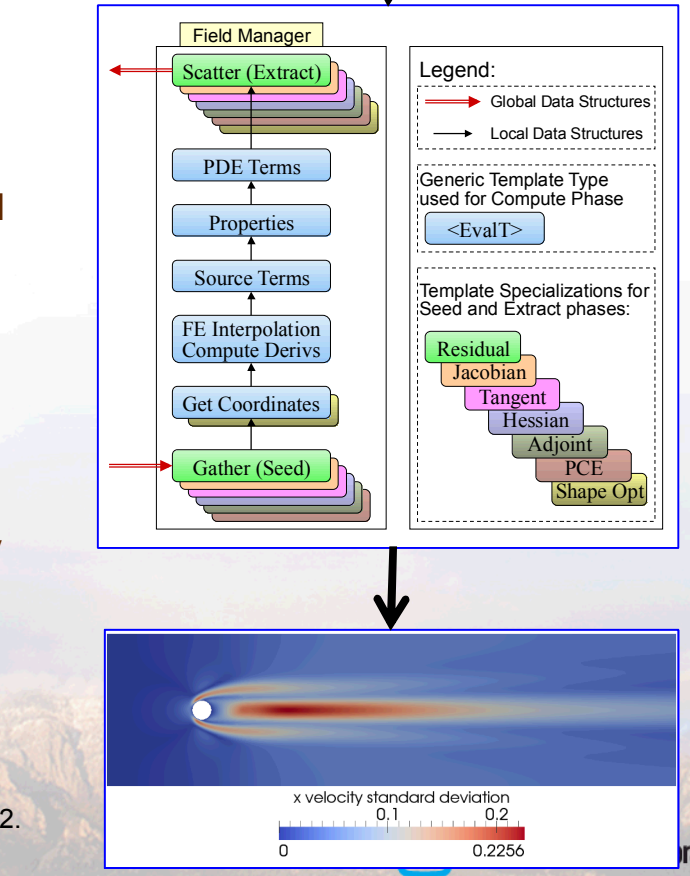
$$a = \sum_{i=0}^{P} a_i \psi_i,\ b = \sum_{j=0}^{P} b_j \psi_j,\ c = ab \approx \sum_{k=0}^{P} c_k \psi_k,\ c_k = \sum_{i,j=0}^{P} a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}$$

- Tools forming and solving SG linear systems
  - SG matrix operators
  - Stochastic preconditioners
  - Hooks to Trilinos parallel solvers and preconditioners
- Nonlinear SG application code interface
  - Connect SG methods to nonlinear solvers, time integrators, optimizers, …

Incorporating Embedded SG Methods in Large-scale Simulations

- **Template-based generic programming[1,2]**
  - Template relevant code on scalar type
  - Instantiate template code on different types for embedded calculations
  - Derivatives: Sacado operator overloading-based AD library
  - SG expansions: Stokhos overloaded operators
- **Element-driven approach**
  - Apply TBGP only at "element-fill" level
  - Developers write templated C++ code for element fill calculations (physics)
  - Handwritten glue code for initializing/extracting derivatives from/to global data structures
- **Demonstrated by Albany code (SNL)**
  - Salinger et al
  - Unstructured Galerkin FEM
  - Pulls together numerous Trilinos packages in a fully functional code for rapid development of complex physics
  - Incompressible fluids, thermo-mechanics, neutronics, …
  - Embedded analysis algorithms

```
template <class ScalarType>
inline ScalarType simple_function(const ScalarType& u) {
  return 1.0/(std::pow(std::log(u),2.0) + 1.0);
}
```



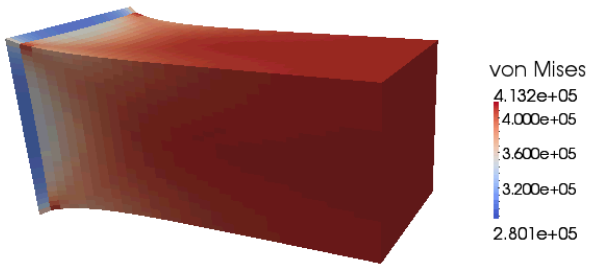
[1,2]Pawlowski, Phipps, Salinger et al, Journal of Scientific Programming, vol. 20 (2-3), 2012.

3-D Linear & Nonlinear Elasticity Model Problems[1]

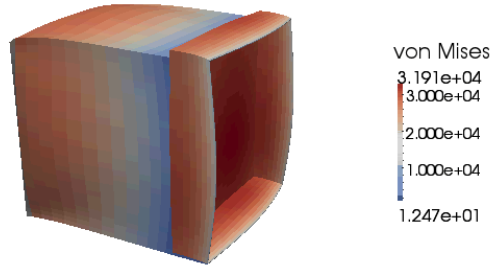
- Linear finite elements, 32x32x32 mesh
  - Nonlinear: neo-Hookean strain energy potential
- Uncertain Young's modulus random field
  - Truncated KL expansion (exponential covariance)
- Albany/LCM code (Salinger, Ostien, et al)
  - Trilinos discretization and solver tools
  - Automatic differentiation
  - Embedded UQ
  - MPI parallelism

Trilinos

http://trilinos.sandia.gov



Displacement (Mean)

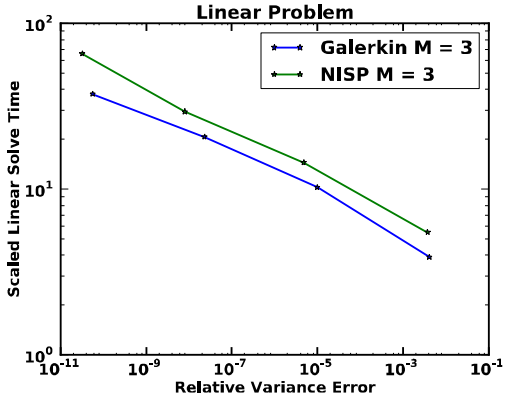


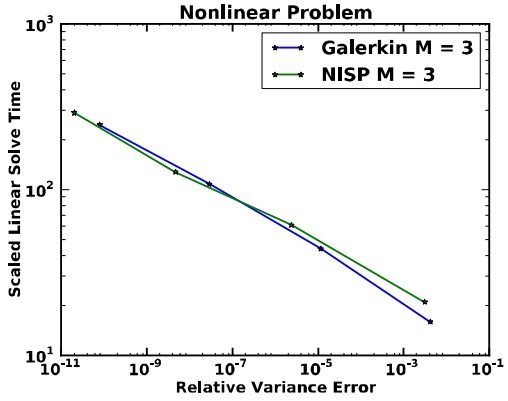
Displacement (Std. Dev.)

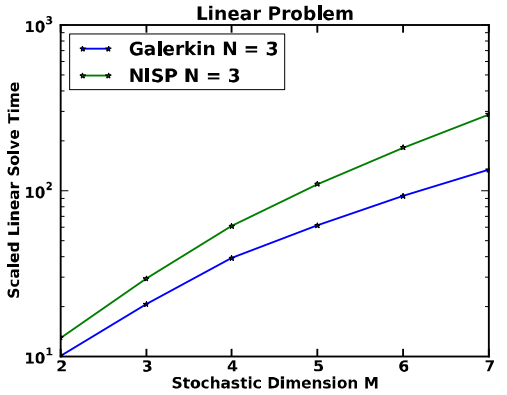
[1]Phipps, Edwards, Hu and Ostien, International Journal of Computer Mathematics, submitted.

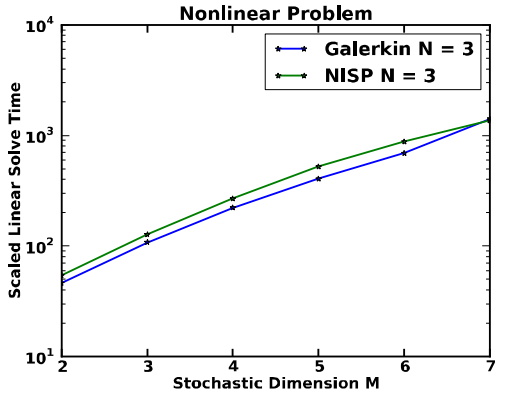
Solve Performance

- Comparison to non-intrusive polynomial chaos/spectral projection (NISP)
  - Isotropic sparse-grid quadrature
  - Gauss-Legendre abscissas
  - Linear growth rules





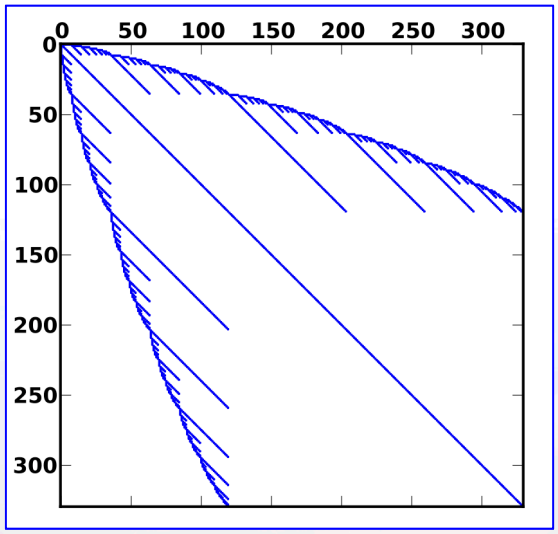




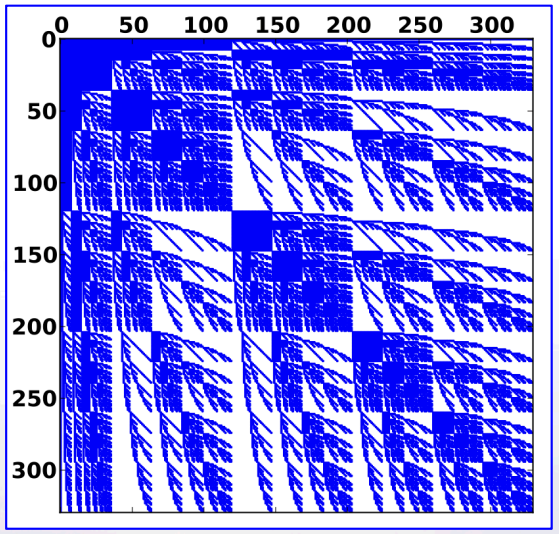
Comparison Between Linear and Nonlinear PDEs

- Difference in performance due to dramatically reduced sparsity of the stochastic Galerkin operator
  - Increased cost of matrix-vector products

Linear Problem



Nonlinear Problem



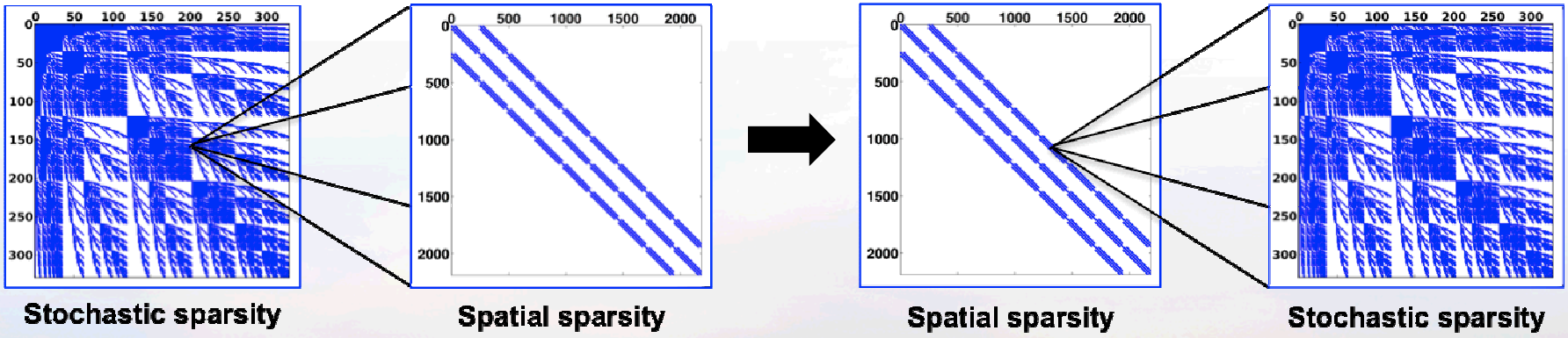
- On-going R&D
  - Improved stochastic preconditioning
  - Dimension reduction for SG Jacobian operator
  - Multicore acceleration

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

$$J^{trad} = \sum_{k=0}^{P} G_k \otimes J_k$$

$$J^{com} = \sum_{k=0}^{P} J_k \otimes G_k$$



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

SG Mat-Vec = Orthogonal Polynomial Multiply

- **Traditional layout: matrix-valued polynomial times vector-valued polynomial:**

$$J(\xi) = \sum_{i=0}^{P} J_i \psi_i(\xi), \quad v(\xi) = \sum_{i=0}^{P} v_i \psi_i(\xi), \quad w(\xi) = J(\xi)v(\xi) \approx \sum_{i=0}^{P} w_i \psi_i(\xi)$$

$$\Longrightarrow \; w_i = \sum_{j,k=0}^{P} J_k v_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle} = \left[\left(\sum_{k=0}^{P} G_k \otimes J_k \right) v^{trad} \right]_i = \left[J^{trad} v^{trad} \right]_i$$

- **Commuted layout: scalar polynomial multiplication:**

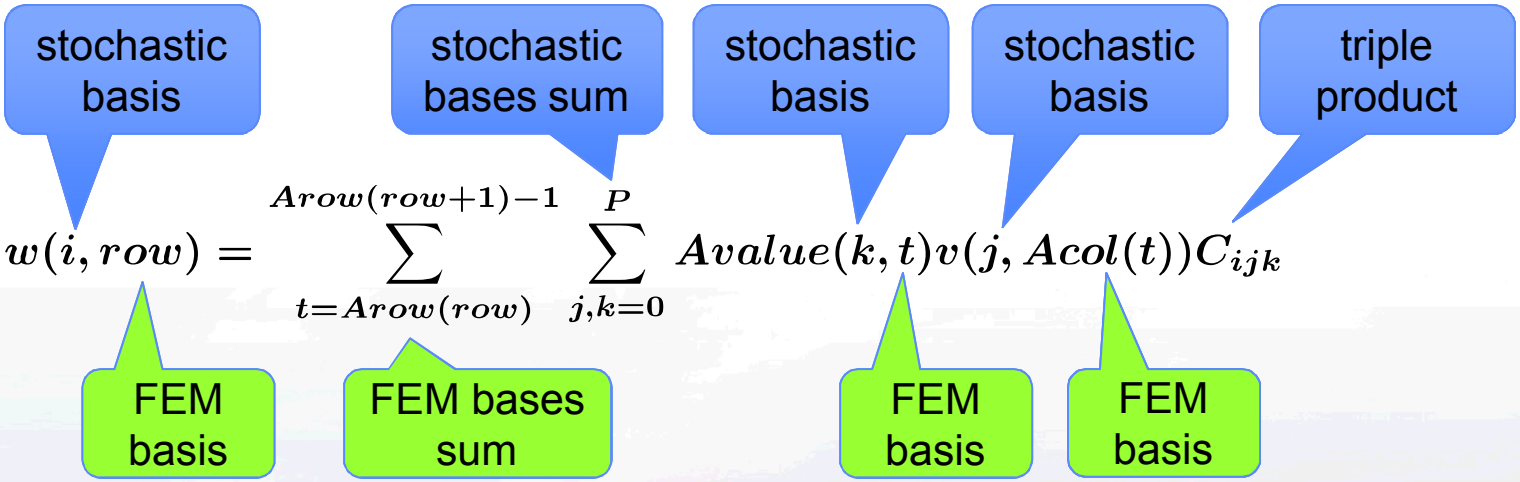
$$J_{ij}(\xi) = \sum_{k=0}^{P} J_{ijk} \psi_k(\xi), \quad v_j(\xi) = \sum_{k=0}^{P} v_{jk} \psi_k(\xi), \quad w_i(\xi) = \sum_{j=0}^{n-1} J_{ij}(\xi) v_j(\xi) \approx \sum_{k=0}^{P} w_{ik} \psi_k(\xi)$$

$$\Longrightarrow \; w_{ik} = \sum_{j=0}^{n-1} \sum_{l,m=0}^{P} J_{ijl} v_{jm} \frac{\langle \psi_k \psi_l \psi_m \rangle}{\langle \psi_m^2 \rangle} = \left[\left(\sum_{l=0}^{P} J_l \otimes G_l \right) v^{com} \right]_{ik} = \left[J^{com} v^{com} \right]_{ik}$$

- **Either way, we have the choice of forming the blocks or using the polynomial algorithm directly**

Commuted SG Matrix Orthogonal Polynomial Multiply

- Two level algorithm
  - Outer: traditional CRS matrix-vector multiply algorithm
  - Inner: orthogonal polynomial multiply



- Symmetric sparse tensor stored in compressed format:

$$w(i,row) = \sum_{t=Arow(row)}^{Arow(row+1)-1} \sum_{n=Crow(i)}^{Crow(i+1)-1} (Avalue(C(n).k,t)v(C(n).j,Acol(t)) + Avalue(C(n).j,t)v(C(n).k,Acol(t)))C(n).value$$

- Opportunities for iteration concurrency: row , i , t , n

Commuted SG Matrix Dense Block Multiply

- Replace inner orthogonal polynomial multiply with dense matrix-vector

$$w(i,row) = \sum_{t=Arow(row)}^{Arow(row+1)-1} \sum_{j=0}^{P} v(j,Acol(t)) \sum_{k=0}^{P} Avalue(k,t) C_{ijk} = Ablock((i,j),t)$$

- Symmetric diagonal storage:

$$Ablock((i,j),t) = \begin{bmatrix} a_{00} & a_{10} & \cdots & a_{(P-1)0} & a_{P0} \\ a_{10} & a_{01} & \cdots & a_{(P-2)1} & a_{(P-1)1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{(P-1)0} & a_{(P-2)1} & \cdots & a_{0(P-1)} & a_{1(P-1)} \\ a_{P0} & a_{(P-1)1} & \cdots & a_{1(P-1)} & a_{0P} \end{bmatrix}$$

- Trade eliminated inner sparse indexing for
  - Increased memory costs: $O(P) \rightarrow O(P^2)$ terms
  - Increased flops if blocks aren't really dense
  - Cost of pre-assembling blocks

Multicore Architectures

- CPU – Quad-socket Intel Sandy Bridge
  - 8 cores/socket x 4 sockets x = 32 threads (Hyperthreading disabled)

- GPU – NVIDIA M2090
  - Fermi architecture
  - 512 cores, 665 GFLOPS peak (double precision)
  - Hierarchical thread structure:
    - Thread blocks must execute independently
    - Each block contains multiple warps (up to 32)
    - Each warp contains 32 threads which are synchronized (SIMT)
  - ~6 GB global memory which all threads can access (slow)
  - Each block accesses 48 kB shared memory (fast)
  - Hardware hides latency of global memory access by fast context switching of active block
    - To achieve high performance, threads need to work with shared memory
    - Programmer controls movement of data between host memory, GPU global memory, and block shared memory

Multicore-CPU: One-level Concurrency

$$w(i,row) = \sum_{t=Arow(row)}^{Arow(row+1)-1} \sum_{n=Crow(i)}^{Crow(i+1)-1} (Avalue(C(n).k,t)v(C(n).j,Acol(t)) + Avalue(C(n).j,t)v(C(n).k,Acol(t)))C(n).value$$

parallel

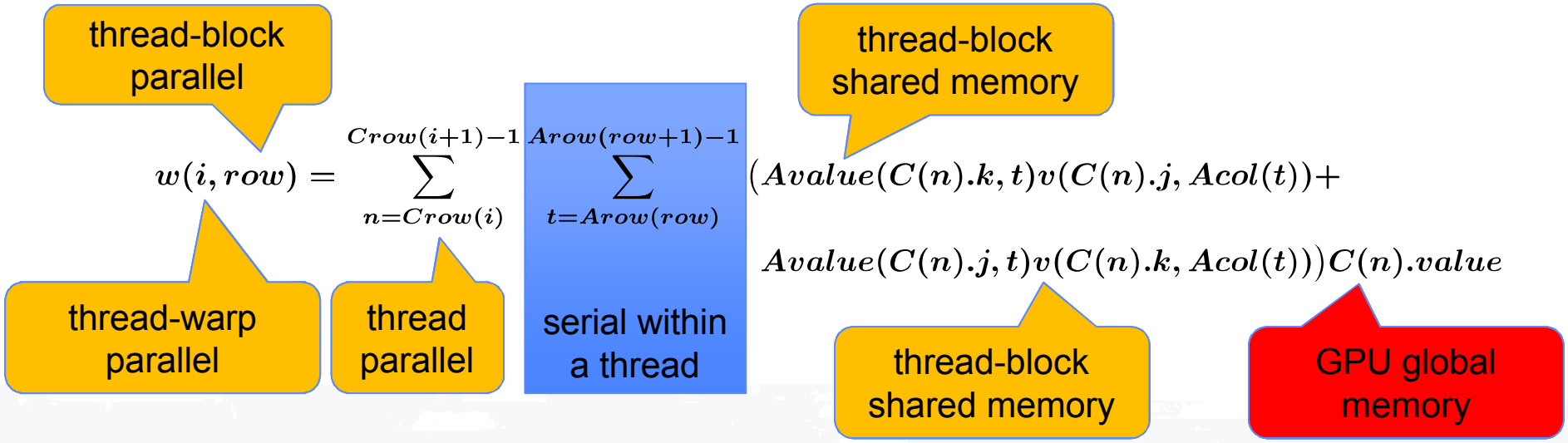
$$w(i,row) = \sum_{t=Arow(row)}^{Arow(row+1)-1} \sum_{j=0}^{P} v(j,Acol(t))Ablock((i,j),t)$$

parallel

serial within a multicore-CPU thread

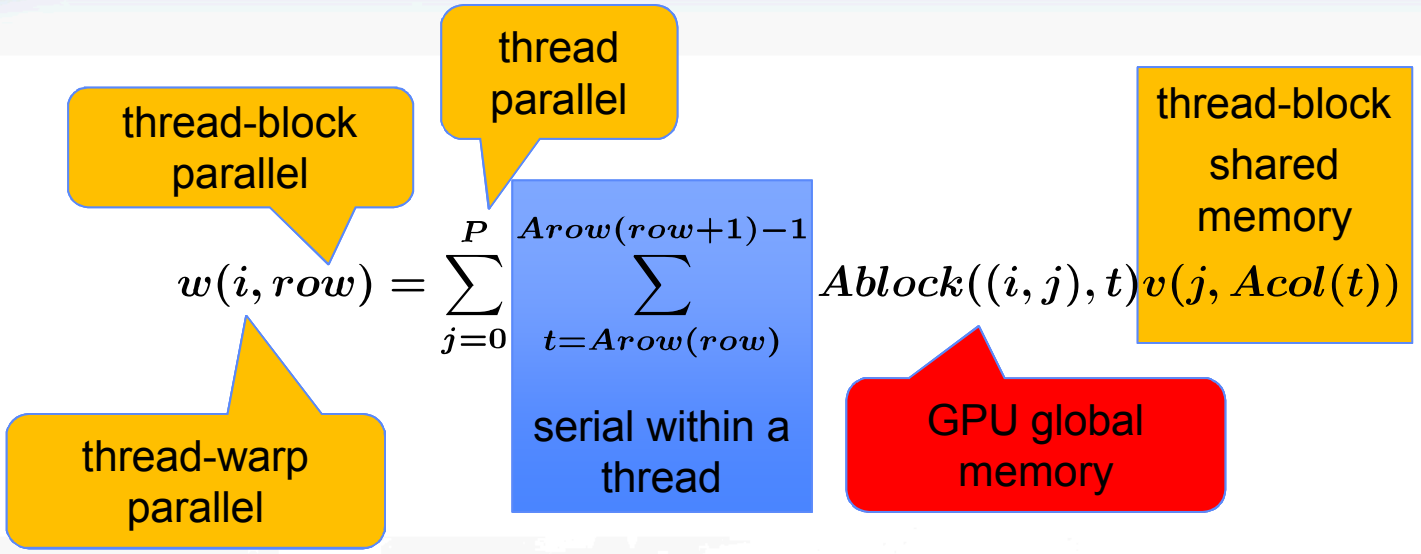
- Each block row "owned" by a CPU thread
- Owning CPU thread computes $w(*,row)$ in serial

Manycore-GPU with Inner Polynomial Multiply: Two-level Concurrency



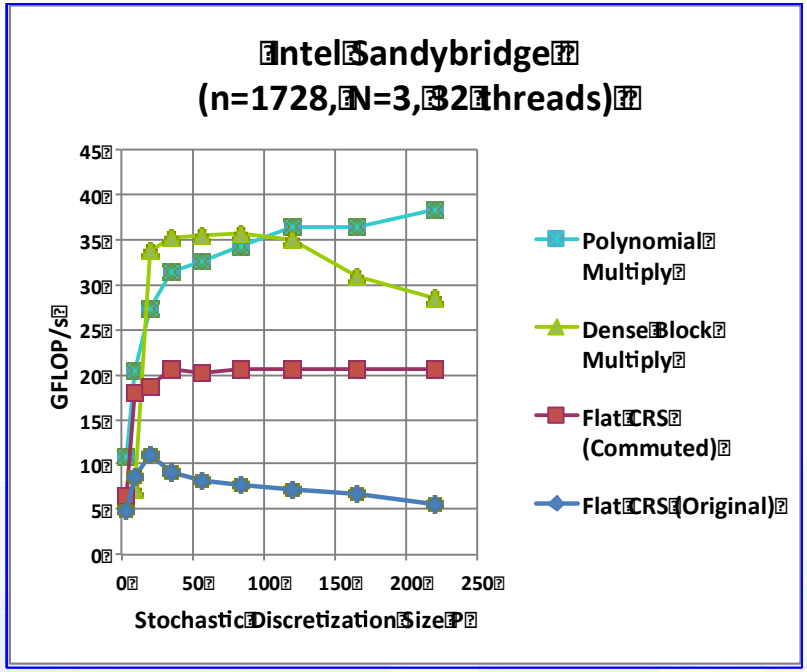
- **Multiple levels of concurrency:**
  - **Each row owned by a thread-block**
  - **Each warp within a thread-block owns an "i"**
  - **Warps within a thread perform polynomial multiply in parallel, executing CRS loop serially**
- **Currently sparse tensor stored in GPU global memory**

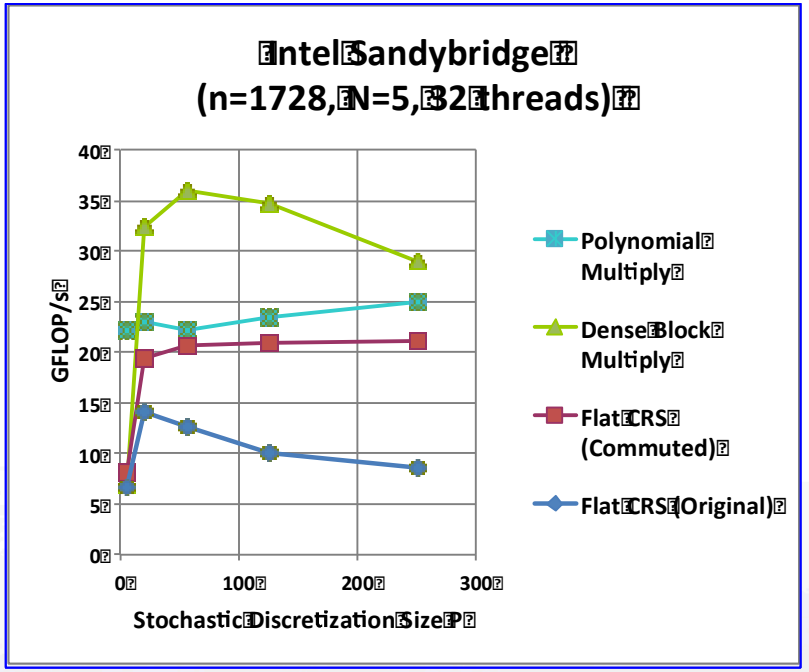
Manycore-GPU with Inner Block Multiply: Two-level Concurrency



- **Multiple levels of concurrency:**
  - **Each row owned by a thread-block**
  - **Each warp within a thread-block owns an "i"**
  - **Warps within a thread perform block multiply in parallel, executing CRS loop serially**
- **Currently blocks stored in GPU global memory**

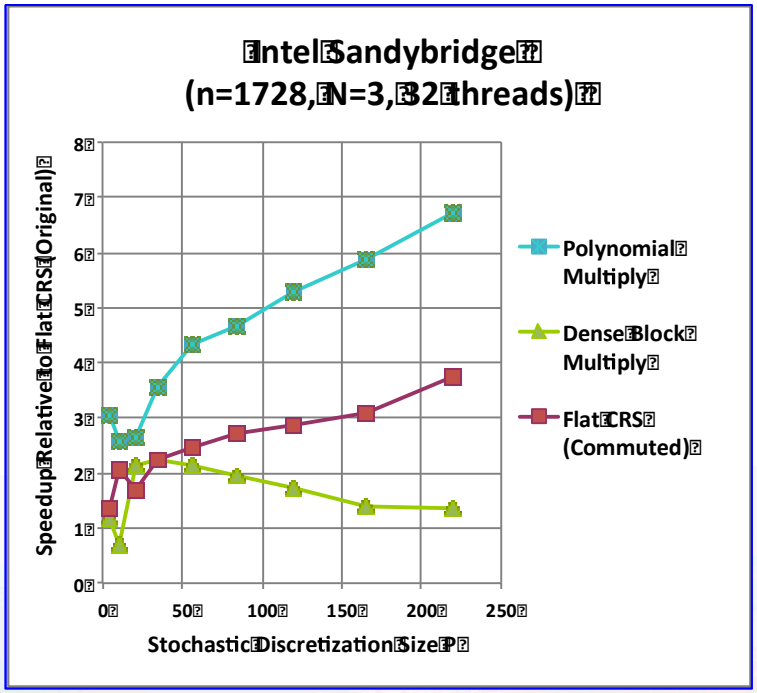
GFLOPS – Intel Sandy Bridge CPU

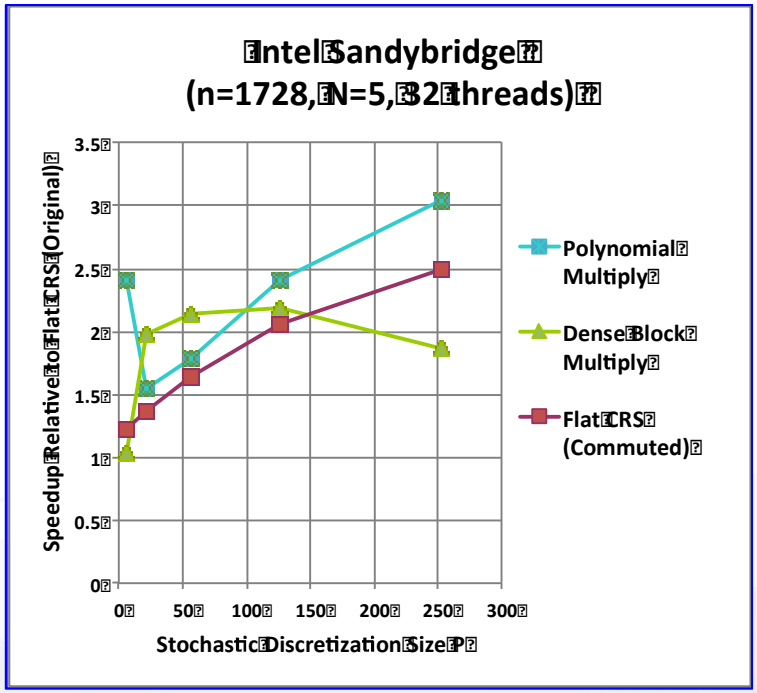




- Standard 3-D first-order FEM grid (12x12x12)
  - Small FEM size due to large memory usage by block and flat-CRS approaches
- N = polynomial order (larger N, denser blocks)

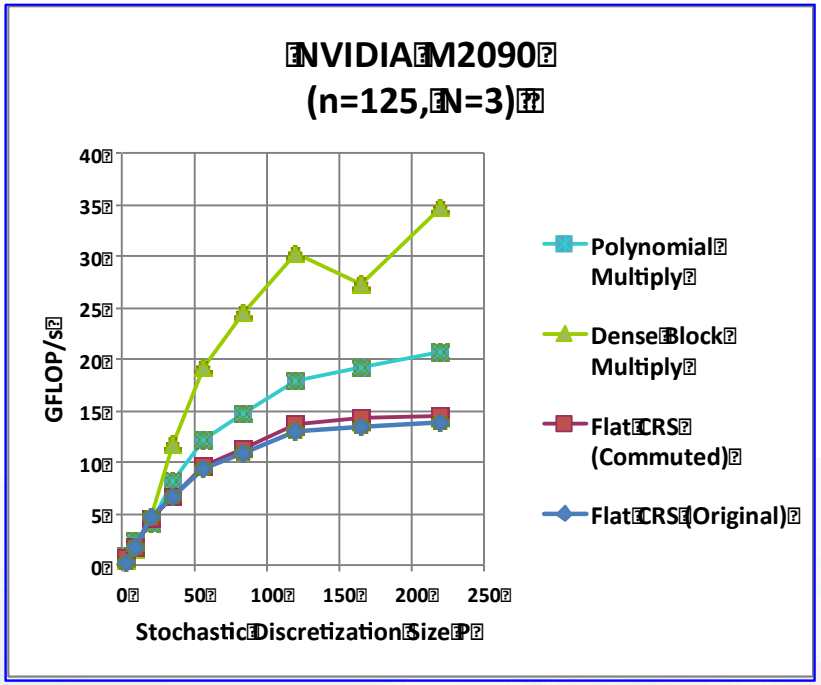
Speedup Relative to Flat CRS

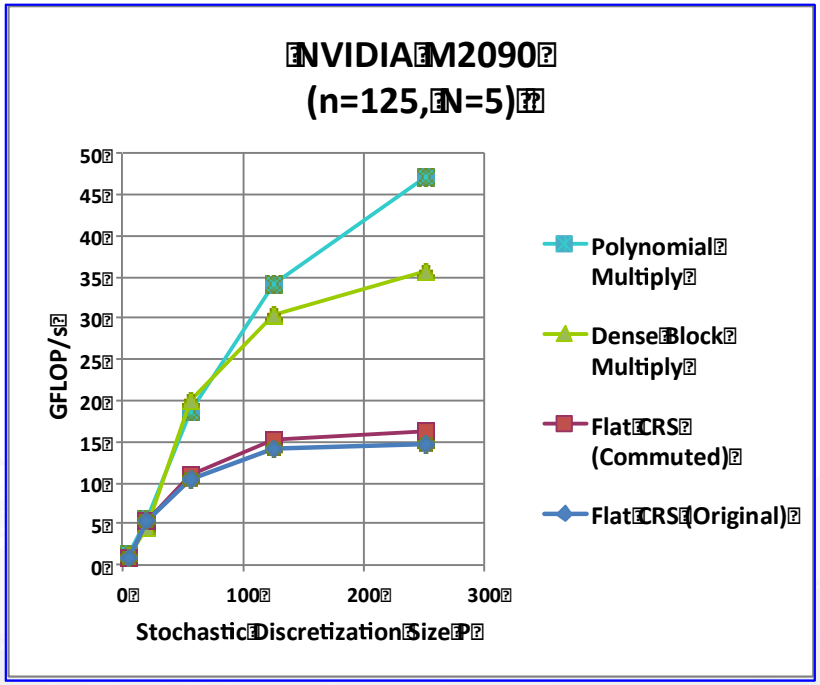




- **Polynomial approach is the clear winner**

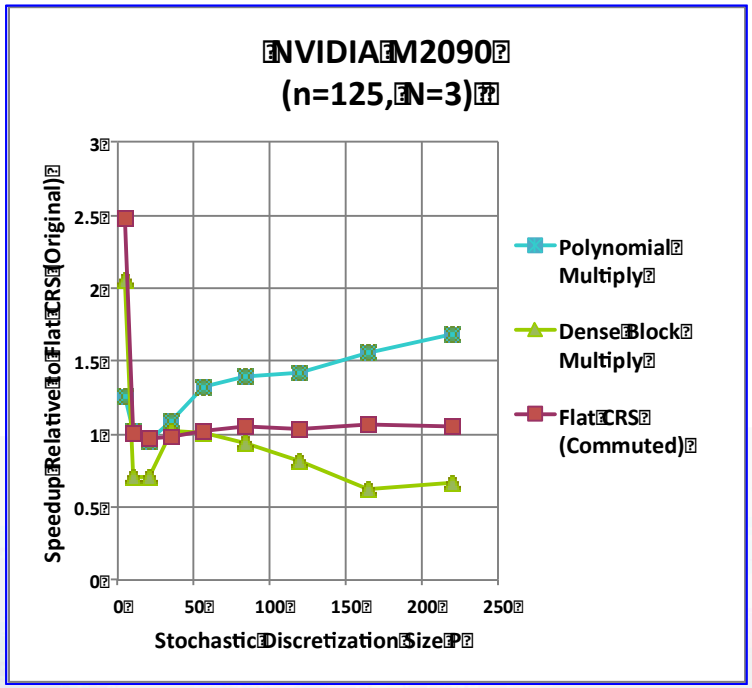
GFLOPS – NVIDIA M2090

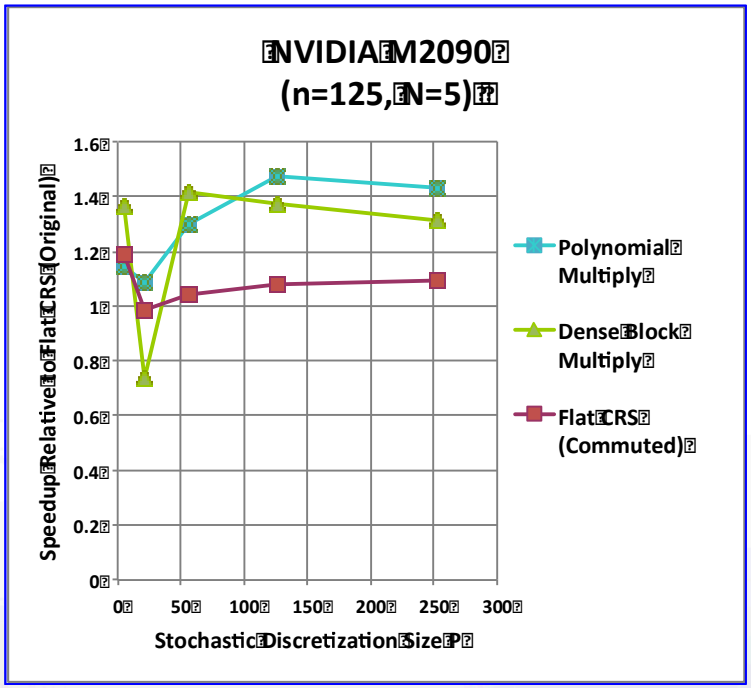




- Standard 3-D first-order FEM grid (5x5x5)
  - Small FEM size due to large memory usage by block and flat-CRS approaches
- N = polynomial order (larger N, denser blocks)

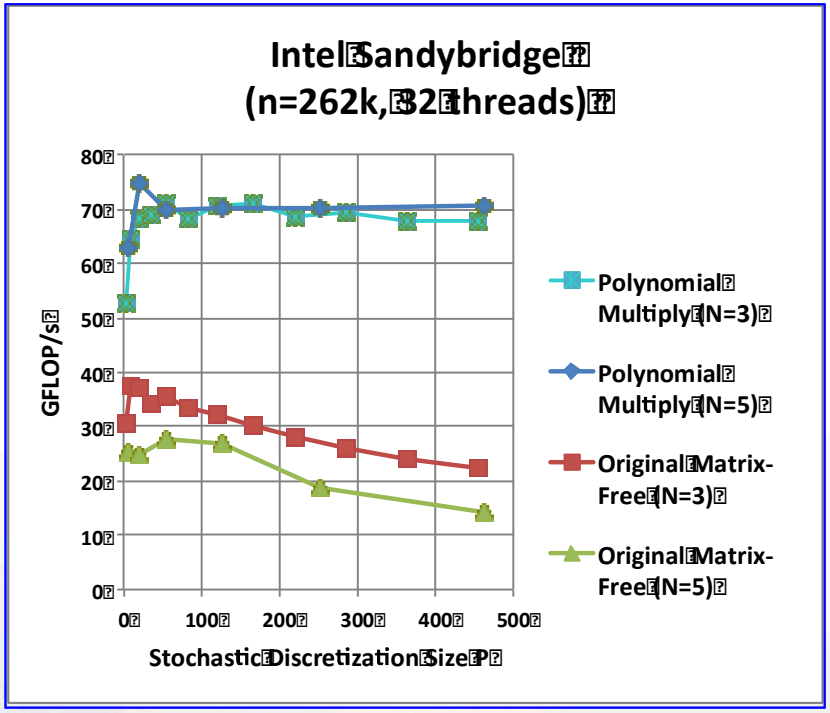
Speedup Relative to Flat CRS

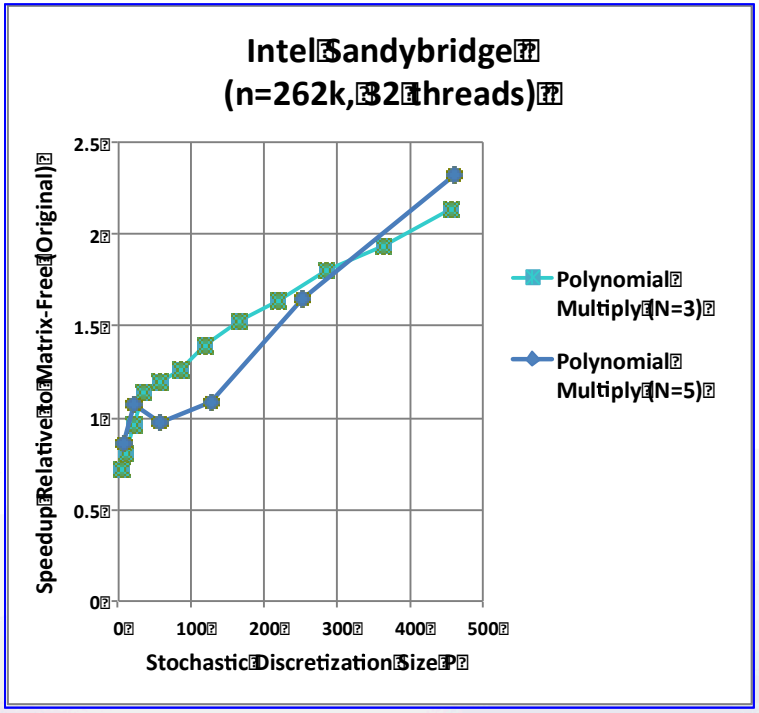




- **Polynomial approach is the clear winner**

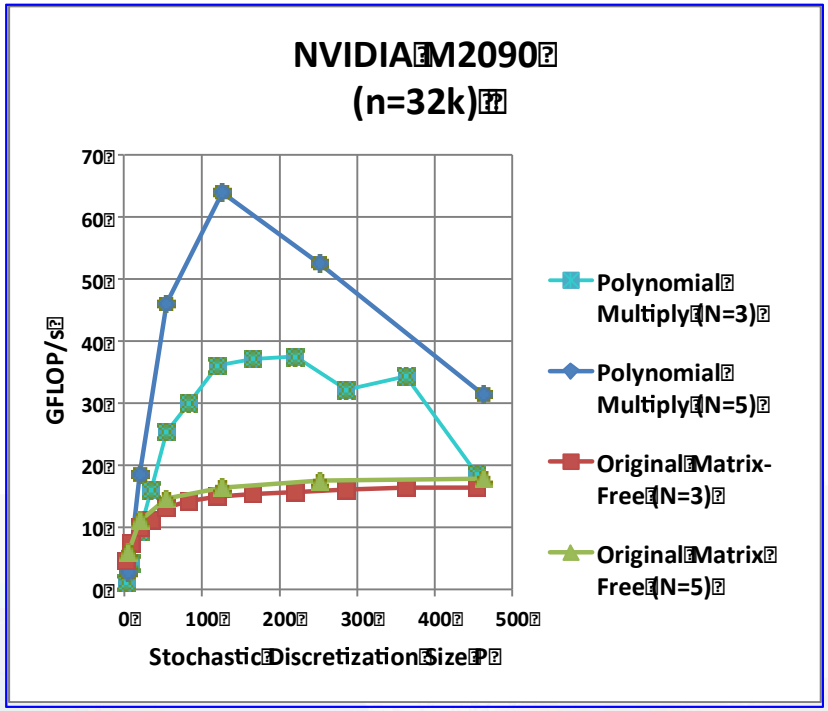
Comparison to Original Matrix-Free

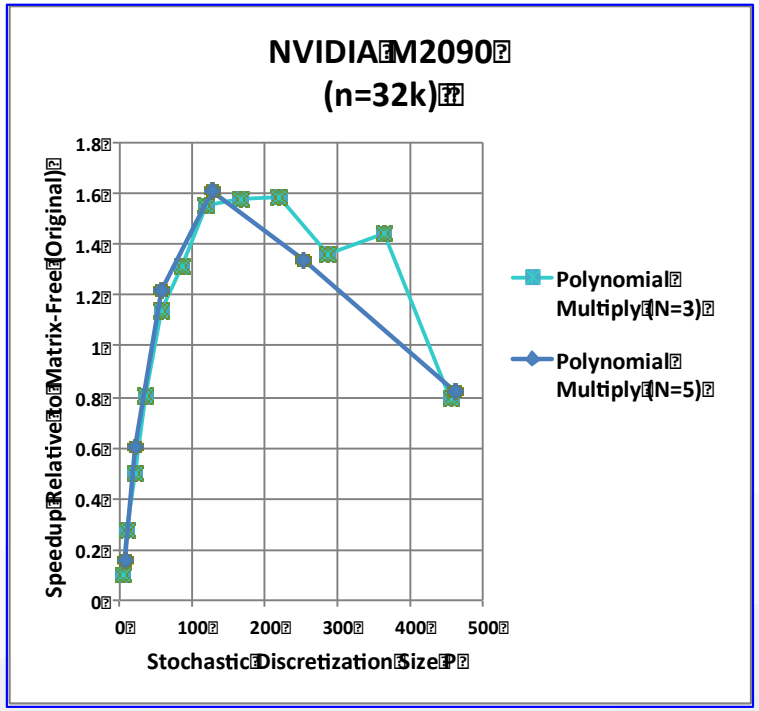




- Reasonable FEM size (64x64x64)
- Significant speedup of polynomial approach over original matrix-free algorithm

Comparison to Original Matrix-Free





- Reasonable FEM size (32x32x32)
- Significant speedup of polynomial approach except for larger stochastic discretizations
  - Too much shared memory usage per CUDA block reduces occupancy

Concluding Remarks

- With proper software infrastructure stochastic Galerkin methods are feasible
  - Template-based generic programming
  - Stokhos/Trilinos solver tools
- Stochastic Galerkin method exhibits reasonable performance for small to moderate numbers of random variables
- Reordering layout improves performance of SG matrix-vector product
  - Significant improvement with commuted polynomial multiply
  - Significant additional levels of parallelism
- Performance issues with GPU version of algorithm
  - Tiling of sparse tensor to reduce shared memory consumption
  - Reordering to improve memory bandwidth performance
- Mat-vecs are only part of the picture
  - Preconditioning with commuted layout requires significant R&D

Extra Slides

What is Automatic Differentiation (AD)?

- Technique to compute analytic derivatives without hand-coding the derivative computation
- How does it work -- freshman calculus
  - Computations are composition of simple operations (+, *, sin(), etc…) with known derivatives
  - Derivatives computed line-by-line, combined via chain rule
- Derivatives accurate as original computation
  - No finite-difference truncation errors
- Provides analytic derivatives without the time and effort of hand-coding them

$$y = \sin(e^x + x \log x), \quad x = 2$$

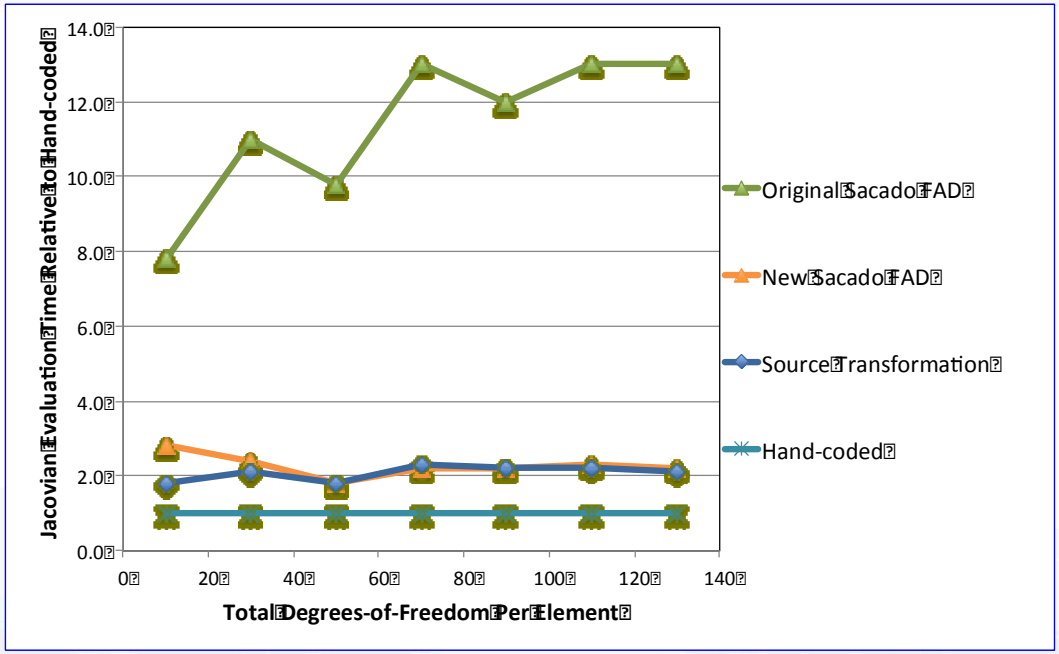
	x	$\frac{d}{dx}$
$x \leftarrow 2$	2.000	1.000
$t \leftarrow e^x$	7.389	7.389
$u \leftarrow \log x$	0.301	0.500
$v \leftarrow xu$	0.602	1.301
$w \leftarrow t + v$	7.991	8.690
$y \leftarrow \sin w$	0.991	-1.188

Sacado: AD Tools for C++ Codes

- Several modes of Automatic Differentiation
  - Forward
  - Reverse
  - Univariate Taylor series
  - Modes can be nested for various forms of higher derivatives
- Sacado uses operator overloading-based approach for C++ codes
  - Phipps, Gay (SNL ASC)
  - Sacado provides C++ data type for each AD mode
  - Replace scalar type (e.g., double) with template parameter
  - Instantiate template code on various Sacado AD types
  - Mathematical operations replaced by overloaded versions
  - Expression templates to reduce overhead

http://trilinos.sandia.gov

Our AD Tools Perform Extremely Well



- Simple set of representative PDEs
  - Total degrees-of-freedom = number of nodes x number of PDEs for each element
- Operator overloading overhead is nearly zero
- 2x cost relative to hand-coded, optimized Jacobian (very problem dependent)

AD to TBGP

- **Benefits of templating**
  - Developers only develop, maintain, test one templated code base
  - Developers don't have to worry about what the scalar type really is
  - Easy to incorporate new scalar types
- **Templates provide a deep interface into code**
  - Can use this interface for more than derivatives
  - Any calculation that can be implemented in an operation-by-operation fashion will work
- **We call this extension Template-Based Generic Programming (TBGP)**
  - Extended precision
    - Shadow double
  - Floating point counts
  - Logical sparsity
  - Uncertainty propagation
    - Intrusive stochastic Galerkin/polynomial chaos
    - Simultaneous ensemble propagation
  - 2 papers under revision to Jou. Sci. Prog.

Generating SG Residual/Jacobian Entries Through Automatic Differentiation (AD)

- Trilinos package Sacado provides AD capabilities to C++ codes
  - AD relies on known derivative formulas for all intrinsic operations plus chain rule
  - AD data types & overloaded operators
  - Replace scalar type in application with Sacado AD data types

- Similar approach can be used to apply SG projections in an operation by operation manner

$$\text{Given } a(y) = \sum_{i=0}^{P} a_i \psi_i(y),\ b = \sum_{i=0}^{P} b_i \psi_i(y),\ \text{find } c(y) = \sum_{i=0}^{P} c_i \psi_i(y)$$

$$\text{such that } \int_\Gamma \left(c(y) - \phi(a(y), b(y))\right) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \ldots, P$$

  - Simple formulas for addition, subtraction, multiplication, division
  - Transcendental operations are more difficult

SG Projections of Intermediate Operations

- **Addition/subtraction**

$$c = a \pm b \Rightarrow c_i = a_i \pm b_i$$

- **Multiplication**

$$c = a \times b \Rightarrow \sum_i c_i \psi_i = \sum_i \sum_j a_i b_j \psi_i \psi_j \rightarrow c_k = \sum_i \sum_j a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}$$

- **Division**

$$c = a/b \Rightarrow \sum_i \sum_j c_i b_j \psi_i \psi_j = \sum_i a_i \psi_i \rightarrow \sum_i \sum_j c_i b_j \langle \psi_i \psi_j \psi_k \rangle = a_k \langle \psi_k^2 \rangle$$

- **Several approaches for transcendental operations**
  - **Taylor series and time integration (Fortran UQ Toolkit by Najm, Debusschere, Ghanem, Knio)**
  - **Tensor product and sparse-grid quadrature (Dakota)**

- **These ideas allow the implementation of Sacado "AD" types for intrusive stochastic Galerkin methods**
  - **Easy transition once code is setup to use AD**

Templated Components Orthogonalize Physics and Embedded Algorithm R&D