CUDA for Real-Time Multigrid Finite Element Simulation of Soft Tissue Deformations

Christian Dick

Computer Graphics and Visualization Group
Technische Universität München, Germany

Motivation

- Real-time physics-based simulation of deformable objects
  - Various applications in medical surgery training and planning
- Finite element method (FEM) in combination with a geometric multigrid solver
  - Physics-based material constants
  - Mathematically sound
  - Easy handling of boundaries
  - Linear-time complexity of the solver in the number of unknowns
- Goal: Exploit the GPU's massive computing power and memory bandwidth to significantly increase simulation update rates / increase FE resolution

Take-Away

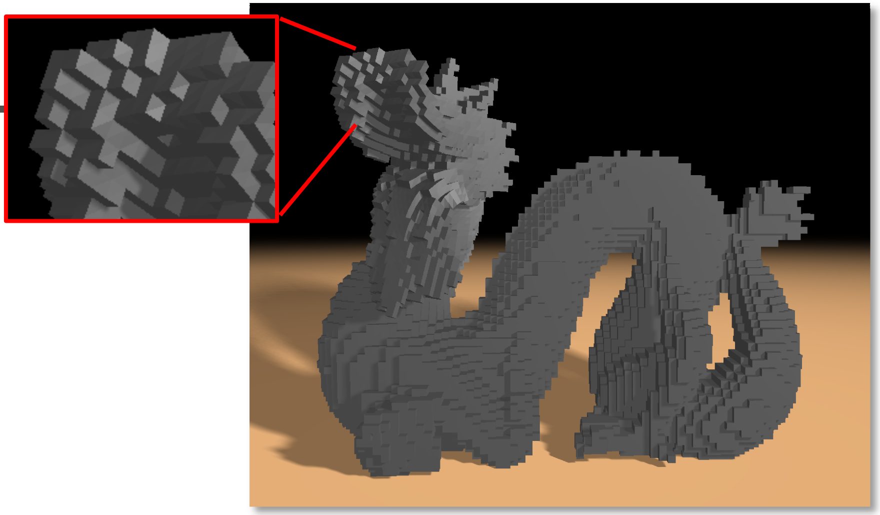
- First fully GPU-based geometric multigrid solver for real-time FEM simulation of deformable objects (linear elasticity, co-rotated strain)
  - Matrix-free FEM and multigrid formulation suited for the GPU
  - Highly efficient CUDA implementation
    - Data structures, memory layout, parallelization
  - Detailed performance analysis
    - Up to 27x faster than 1 CPU core / 4x faster than 8 CPU cores
    - Up to 56 GFLOPS (single) / 34 GFLOPS (double precision) *(sustained performance)*
    - Up to 88 GB/s memory throughput *(sustained performance)*
    - Simulation rates:
      - 120 time steps/sec for 12,000 hexahedral elements
      - 11 time steps/sec for 269,000 hexahedral elements

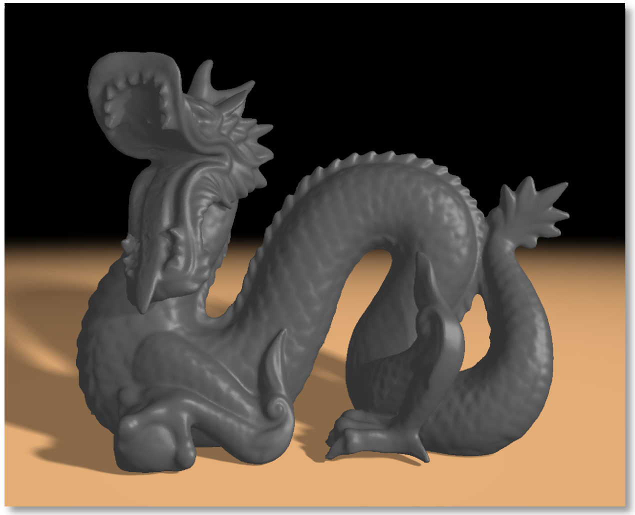
Deformable Objects on the GPU

- Architecture of the NVIDIA Fermi GPU
  - 15 multiprocessors, each with 32 CUDA cores (ALUs) for integer- and floating-point arithmetic operations
  - GPU executes thousands of threads in parallel
- Requirements for an algorithm to run *efficiently* on the GPU
  - Restructure algorithm to expose fine-grained parallelism
    (one thread per data element)
  - Avoid execution divergence of threads in the same warp
  - Choose memory layouts which enable coalescing of device memory accesses
  - Only threads in the same thread block can communicate and be synchronized efficiently (global synchronization only via separate kernel calls)
  - Very limited resources (registers, shared memory) per thread

Our Approach

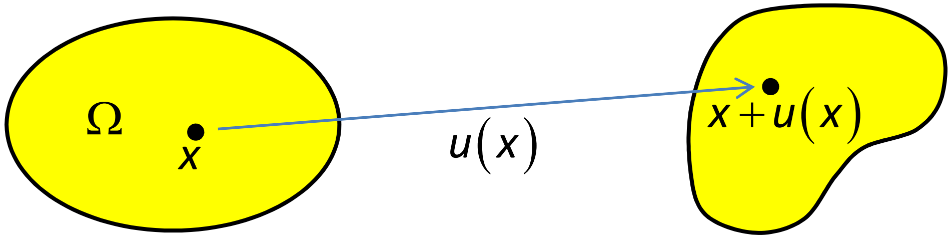
- Hexahedral (tri-linear) finite elements on a uniform Cartesian grid
- Linear elasticity, co-rotated strain
- Geometric multigrid solver
- CUDA API to flexibly access all resources on the graphics card
- Advantages:
  - Numerical stencil of regular shape enables efficient GPU implementation
  - FE model and multigrid hierarchy generation is easy and fast
  - Only one pre-computed element stiffness matrix (greatly reduces memory requirements)





Elasticity & FEM

- Deformation of an object is described by a displacement field $u : \Omega \rightarrow \mathbb{R}^3$



Undeformed configuration Deformed configuration

- *u* is determined by minimizing the functional (potential energy)

$$E(u) = \frac{1}{2}\int_{\Omega} \varepsilon^T \sigma \, dx - \int_{\Omega} f^T u \, dx \rightarrow \min$$

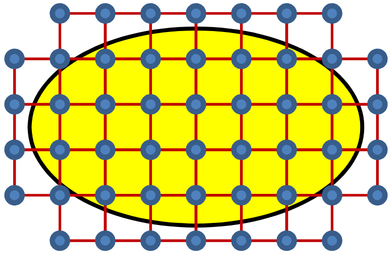
ε : Strain

σ : Stress

f : External forces

Elasticity & FEM

- For the numerical solution, *u* and *f* are discretized by
  - a finite element decomposition of the object's domain (regular hexahedra) and
  - a nodal basis formed by the elements' shape functions (tri-linear interpolation)



$$u(x) = \sum_{\text{Nodes } i} u_i \varphi_i(x)$$

u_i : Displacement at Node *i*

φ_i : Basis function for Node *i*

- For a single element, the functional is minimized by solving a linear system

$$E^e\left(u^e\right) = \frac{1}{2}\int_\Omega \left(u^e\right)^T B^T D B u^e \, dx - \int_\Omega \left(f^e\right)^T u^e \, dx \rightarrow \min$$

$$\Leftrightarrow \boxed{\underbrace{B^T D B}_{K^e} u^e = f^e}$$

K^e : Element stiffness matrix

u^e : Linearized u_i

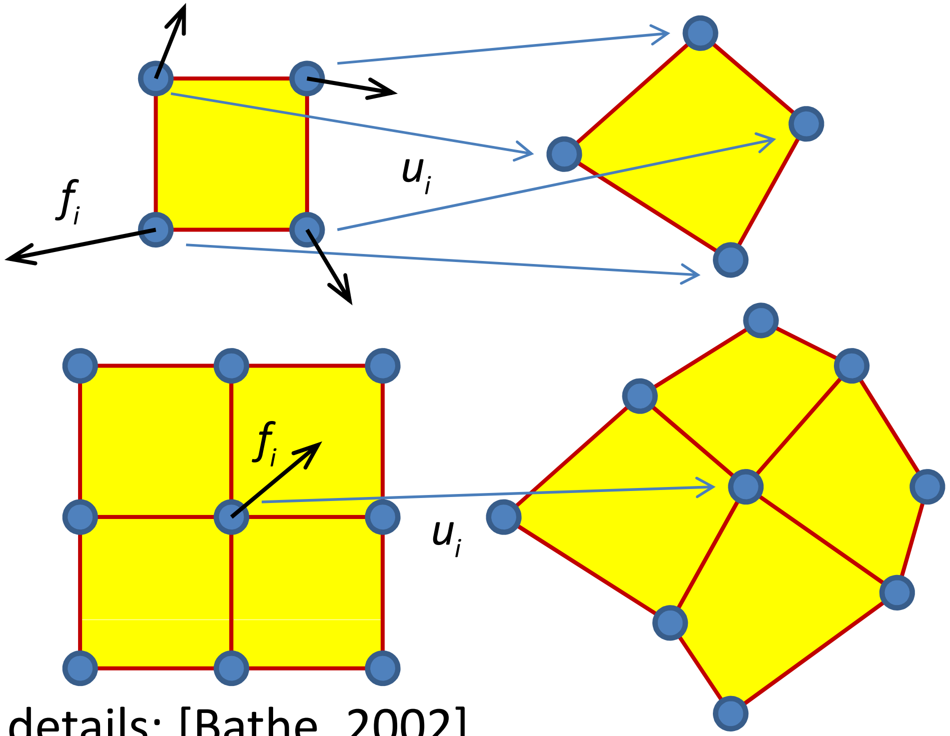
f^e : Linearized f_i

B : Strain matrix

D : Material law

Elasticity & FEM

- The global system of equations is assembled from the individual elements by considering sharing of vertices



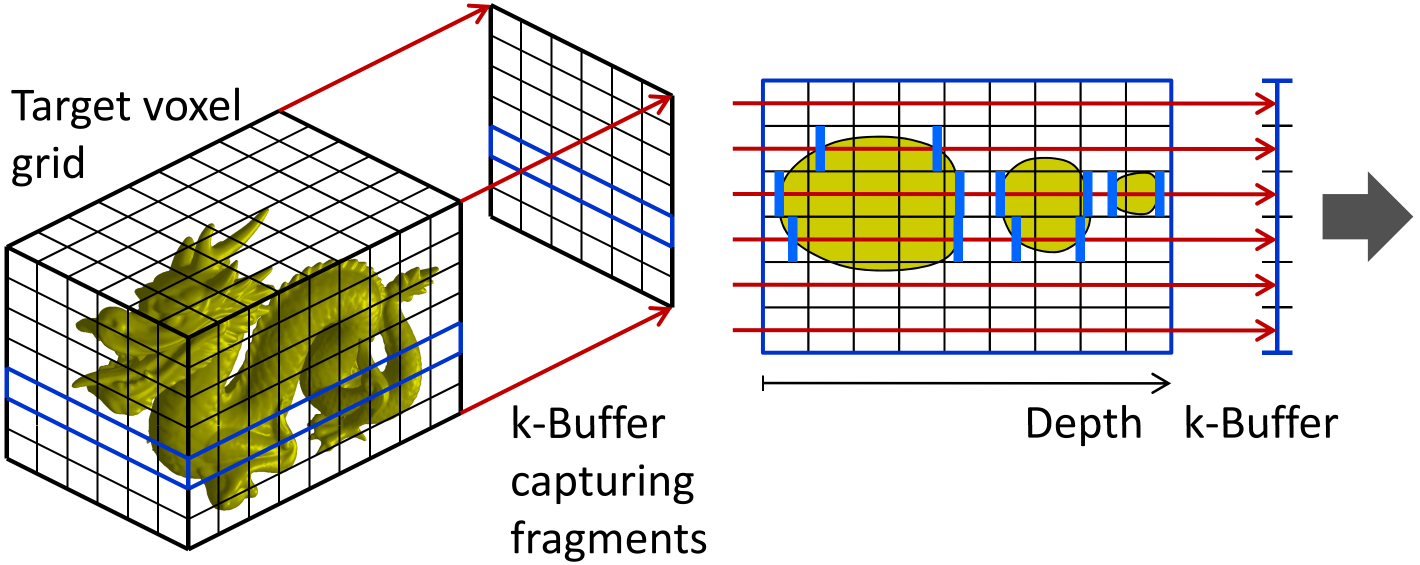
Per element

$$\begin{tikzcd} K^e u^e = f^e \arrow[d, "\text{Sharing of } u_i \text{ and } f_i"] \\ Ku = f \end{tikzcd}$$

- For more details: [Bathe, 2002]

Finite Element Discretization

- Hexahedral finite element discretization on a uniform Cartesian grid
  – Easily obtained from a surface triangle mesh by voxelization
    - A voxel is classified as “inside” iff the voxel’s center is inside the object
    - The “inside” voxels correspond to the finite elements



0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0
1	1	1	1	0	1	1	0	1
0	1	1	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

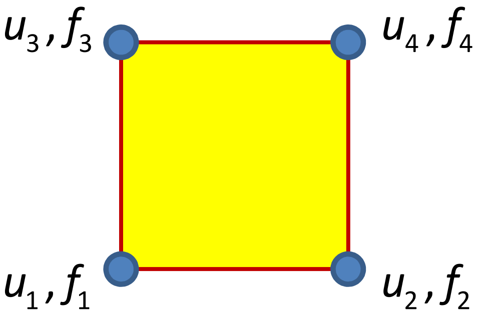
Per-Element Equations (static)

- Per-element equations

$$\boxed{\sum_{j=1}^{8} K_{ij} u_j = f_i} \quad , \quad i = 1, \ldots, 8$$

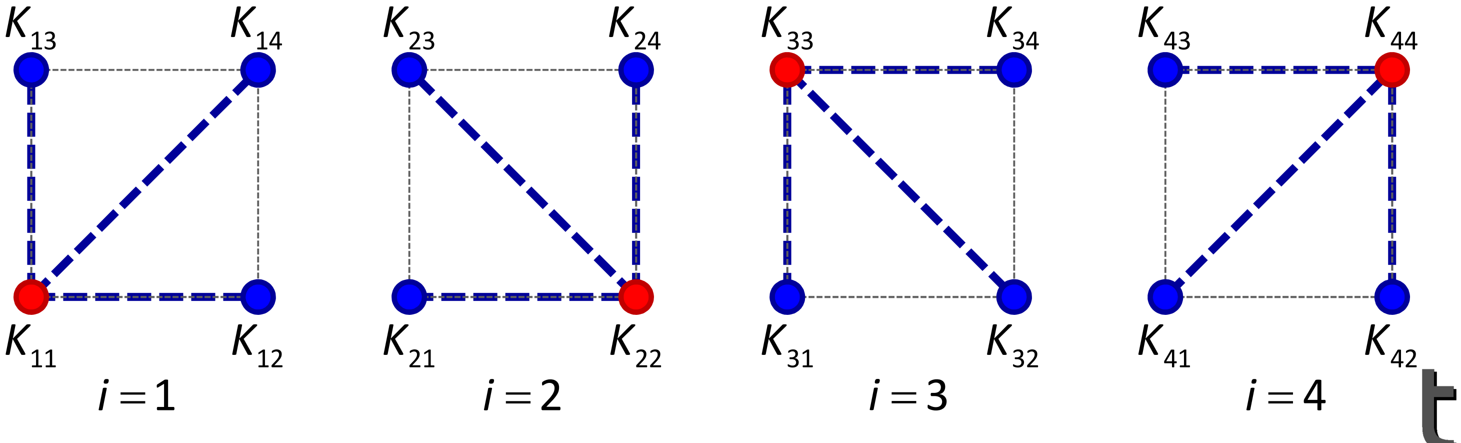
u_i : Displacements

f_i : Forces



e is now omitted

K : Element stiffness matrix consisting of 8x8 3x3-matrix coefficients



Per-Element Equations (co-rotated, static)

- Co-rotated strain

Rotate forces from initial into current configuration

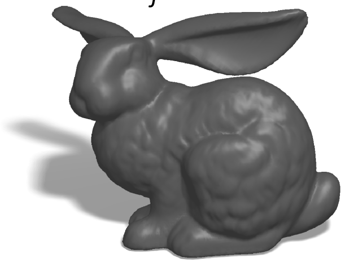
Rotate displacements back into initial configuration

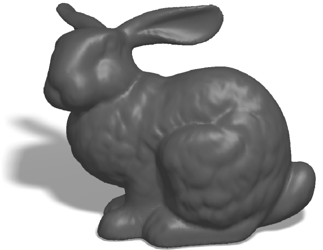
$$\sum_{j=1}^{8} \boxed{RK_{ij} \boxed{\left(R^T\left(p_j^0 + u_j\right) - p_j^0\right)}} = f_i \quad , \quad i = 1,\dots,8$$

R : Element rotation

p_j^0: Undeformed vertex positions

$$\boxed{\sum_{j=1}^{8} \underbrace{RK_{ij}R^T}_{\hat{A}_{ij}} u_j = \underbrace{f_i - \sum_{j=1}^{8} RK_{ij}\left(R^T p_j^0 - p_j^0\right)}_{\hat{b}_i}}$$


Linear strain


Linear, co-rotated strain

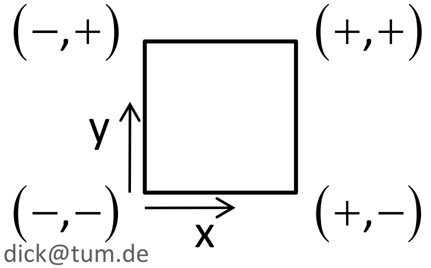
- R is obtained by polar decomposition of the element's average deformation gradient (5 iterations)

$$R_0 = I_3 + \frac{1}{4}\sum_{i=1}^{8} u_i^{old}\left(\pm\frac{1}{a}, \pm\frac{1}{b}, \pm\frac{1}{c}\right)$$

$$R_{n+1} = \frac{1}{2}\left(R_n + \left(R_n^{-1}\right)^T\right)$$

a, b, c : Edge lengths of hexahedral element

Signs :



Per-Element Equations (co-rotated, dynamic)

- Dynamics

$$\sum_{j=1}^{8}\left(M_{ij}\ddot{u}_j + C_{ij}\dot{u}_j + \hat{A}_{ij}u_j\right) = \hat{b}_i \quad , \quad i = 1,\ldots,8$$

M: Mass matrix

C : Damping matrix

- Newmark time integration

$$\dot{u}_i = \frac{2}{dt}\left(u_i - u_i^{old}\right) - \dot{u}_i^{old} \qquad , \qquad \ddot{u}_i = \frac{4}{dt^2}\left(u_i - u_i^{old} - \dot{u}_i^{old}dt\right) - \ddot{u}_i^{old}$$

dt: Time step length

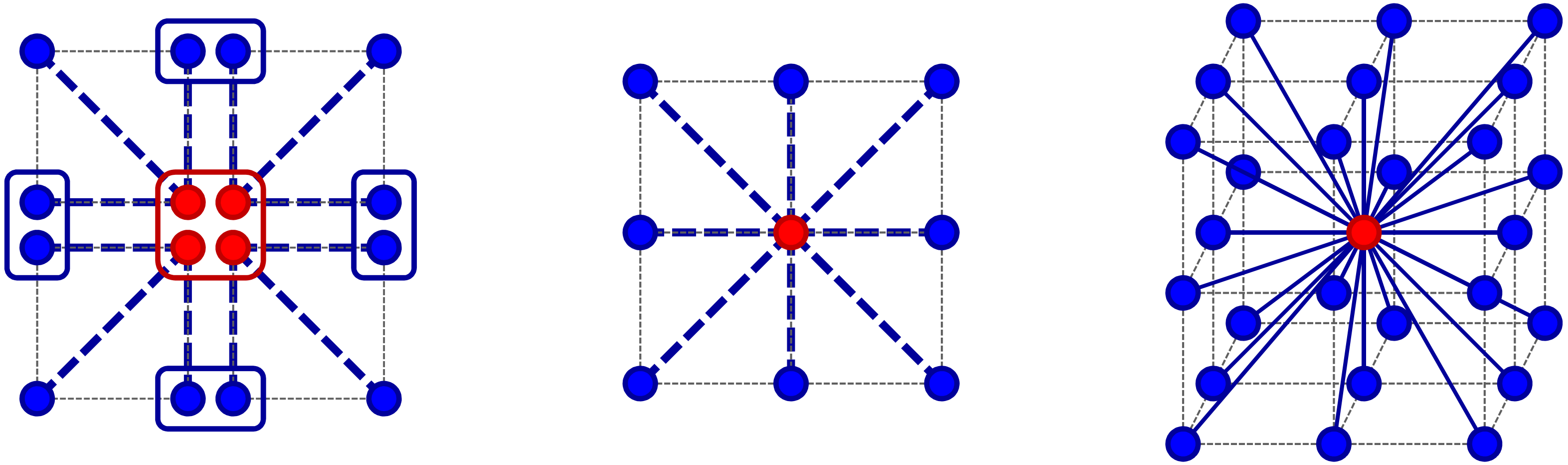
$$\boxed{\underbrace{\sum_{j=1}^{8}\left(\frac{4}{dt^2}M_{ij} + \frac{2}{dt}C_{ij} + \hat{A}_{ij}\right)}_{A_{ij}}u_j = \underbrace{\hat{b}_i + \sum_{j=1}^{8}\left[M_{ij}\left(\frac{4}{dt^2}\left(u_j^{old} + \dot{u}_j^{old}dt\right) + \ddot{u}_j^{old}\right) + C_{ij}\left(\frac{2}{dt}u_j^{old} + \dot{u}_j^{old}\right)\right]}_{b_i}}$$

- Mass proportional damping: $C_{ij} = \alpha M_{ij}$
- Mass lumping: $M_{ii} = m_i I_3 \quad , \quad M_{ij} = 0$ for $i \neq j$

mass of vertex i (1/8 element mass)

Per-Vertex Equations

- Per-vertex equations: Add stencil coefficients of shared vertices



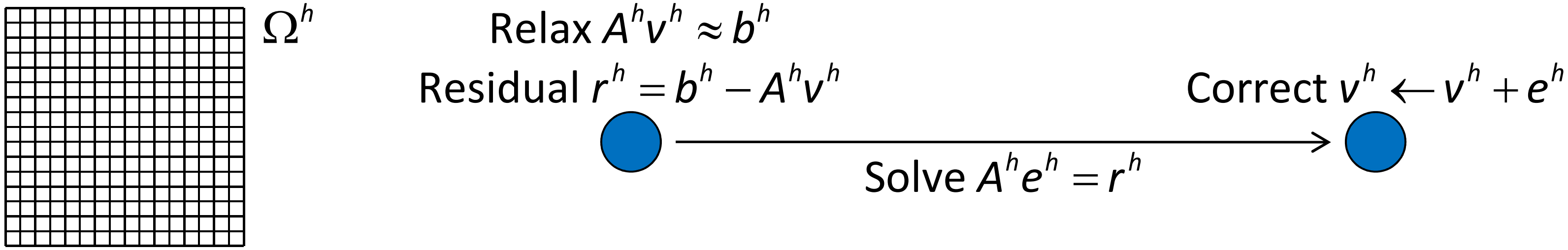
- Stencil on a 3^3 domain of adjacent vertices: 27 3x3-matrix coefficients

$$\sum_{\boldsymbol{i}=(-1,-1,-1)^T}^{(1,1,1)^T} A_{\boldsymbol{i}}^{\boldsymbol{x}} u_{\boldsymbol{x}+\boldsymbol{i}} = b_{\boldsymbol{x}}$$

- Dirichlet boundary conditions: Replace equation by $I_3 u_{\boldsymbol{x}} = 0$ and in the other equations set the respective coefficient $A_{\boldsymbol{i}}$ corresponding to this vertex to 0

Geometric Multigrid

- Solve $A^h u^h = b^h$, current approximate solution v^h



Ω^h

Relax $A^h v^h \approx b^h$

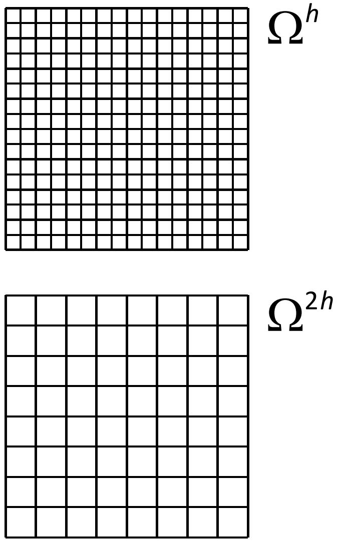
Residual $r^h = b^h - A^h v^h$

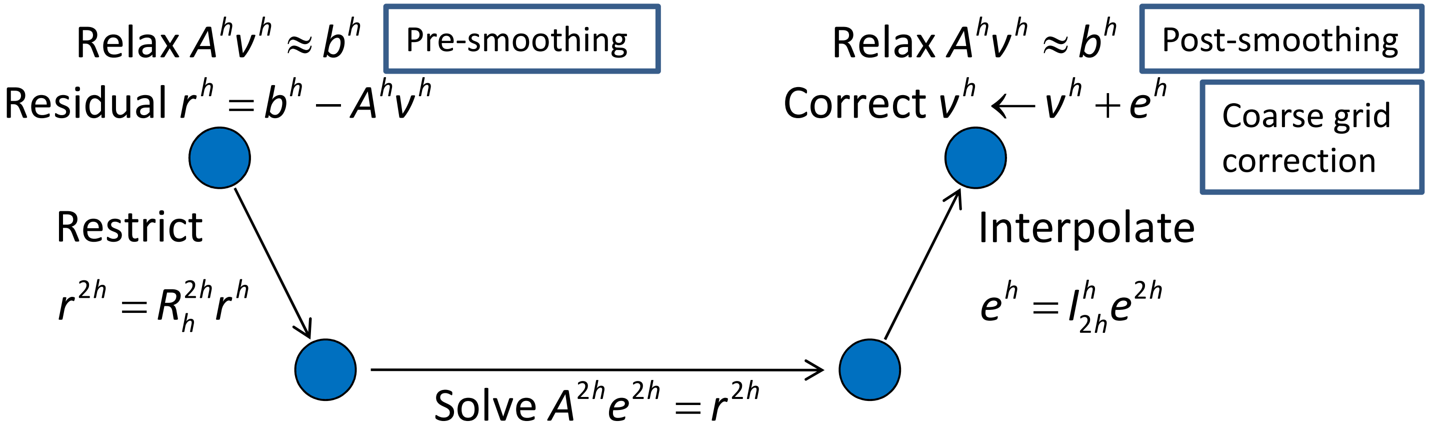
Solve $A^h e^h = r^h$

Correct $v^h \leftarrow v^h + e^h$

Geometric Multigrid

- Solve $A^h u^h = b^h$, current approximate solution v^h





Relax $A^h v^h \approx b^h$ Pre-smoothing

Residual $r^h = b^h - A^h v^h$

Restrict

$r^{2h} = R_h^{2h} r^h$

Solve $A^{2h} e^{2h} = r^{2h}$

Interpolate

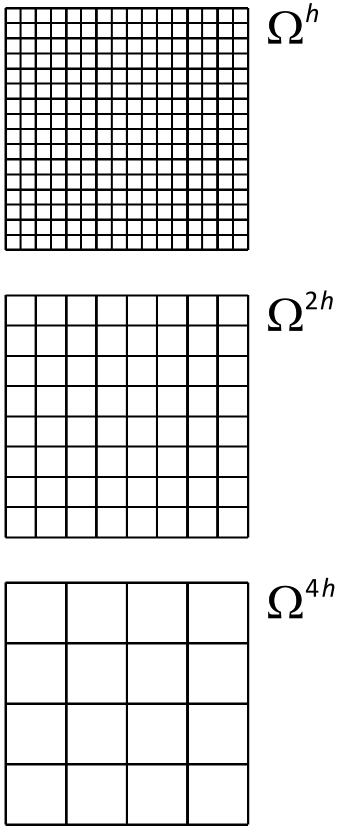
$e^h = I_{2h}^h e^{2h}$

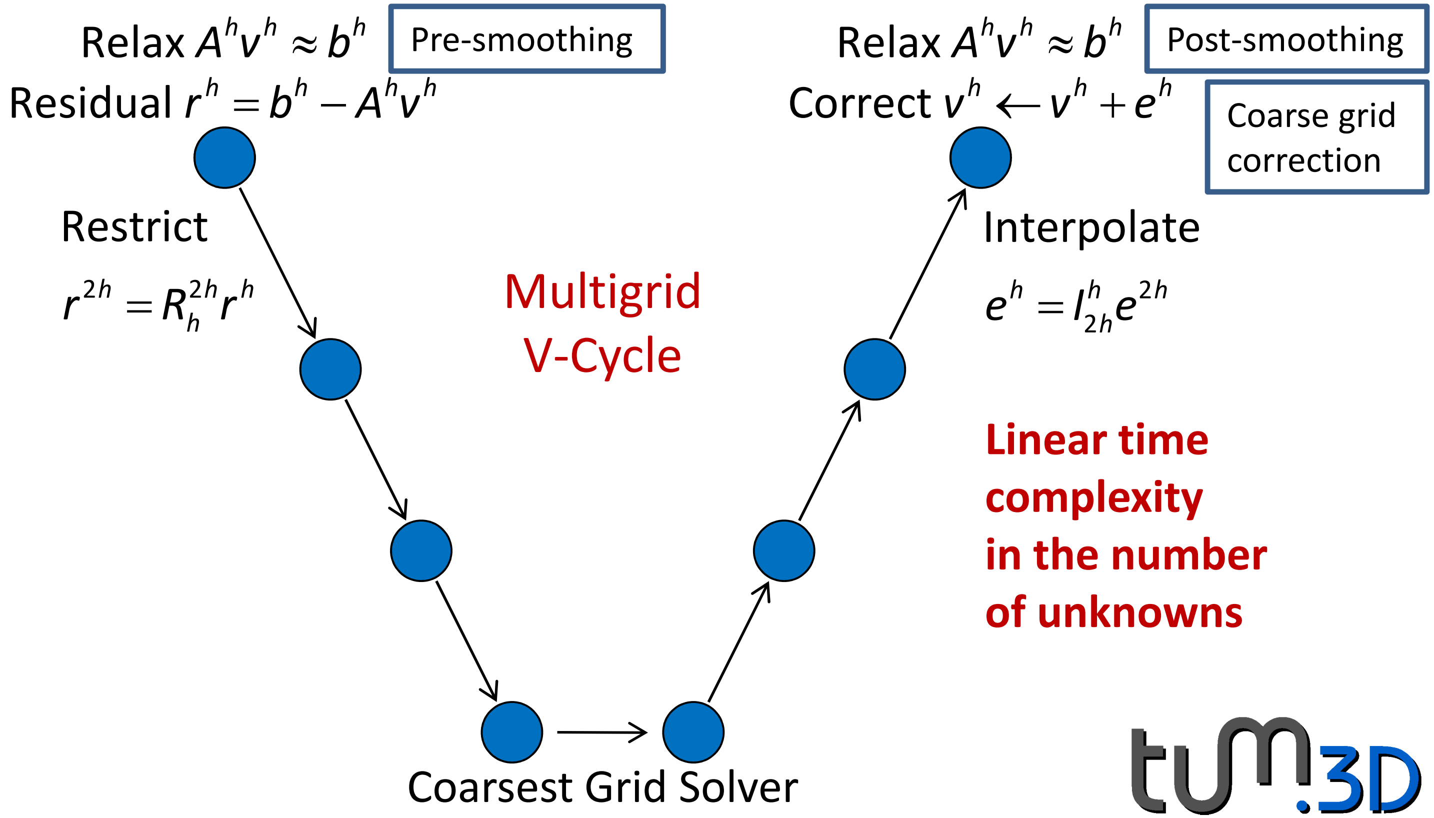
Correct $v^h \leftarrow v^h + e^h$ Coarse grid correction

Relax $A^h v^h \approx b^h$ Post-smoothing

Geometric Multigrid

- Solve $A^h u^h = b^h$, current approximate solution v^h





Relax $A^h v^h \approx b^h$ Pre-smoothing

Residual $r^h = b^h - A^h v^h$

Restrict

$r^{2h} = R_h^{2h} r^h$

Multigrid V-Cycle

Coarsest Grid Solver

Relax $A^h v^h \approx b^h$ Post-smoothing

Correct $v^h \leftarrow v^h + e^h$ Coarse grid correction

Interpolate

$e^h = I_{2h}^h e^{2h}$

Linear time complexity in the number of unknowns

Variational Properties of Multigrid

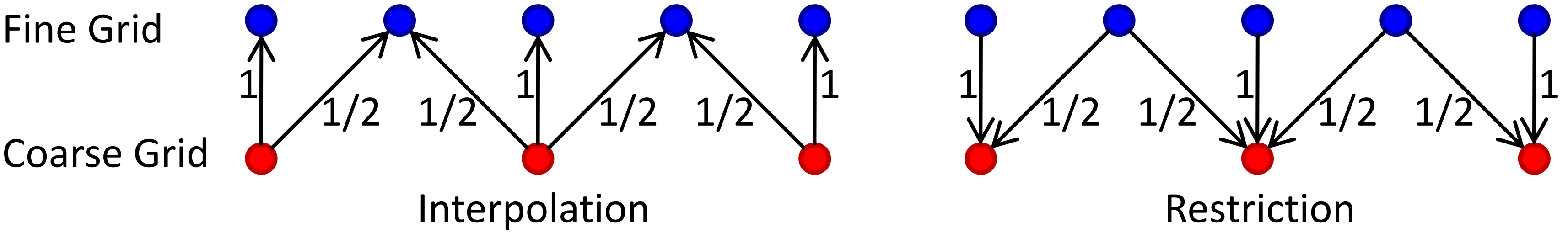
- Selection of transfer operators and coarse grid operators:

  I_{2h}^{h} : Tri-linear interpolation

  $R_h^{2h} = \left(I_{2h}^{h}\right)^T$ ← Restriction is transpose of interpolation

  $A^{2h} = R_h^{2h} A^h I_{2h}^{h}$ ← Galerkin-based coarsening

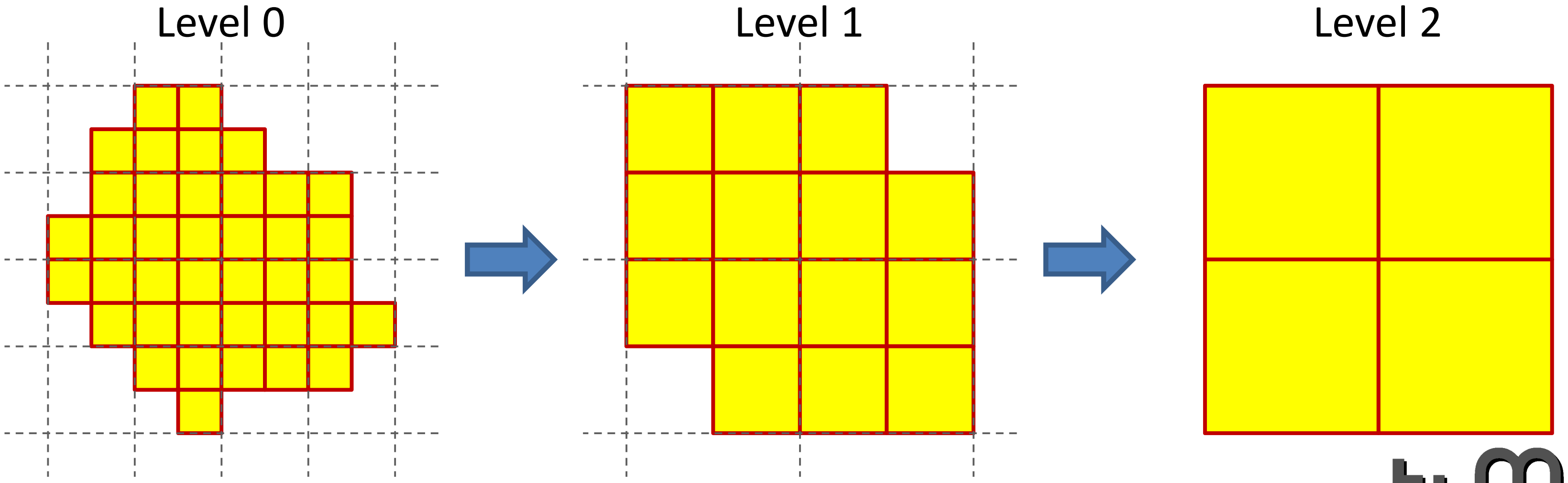
1D Example:



- In the finite element context, these choices naturally arise from the variational principle
  - Coarse grid correction minimizes the functional

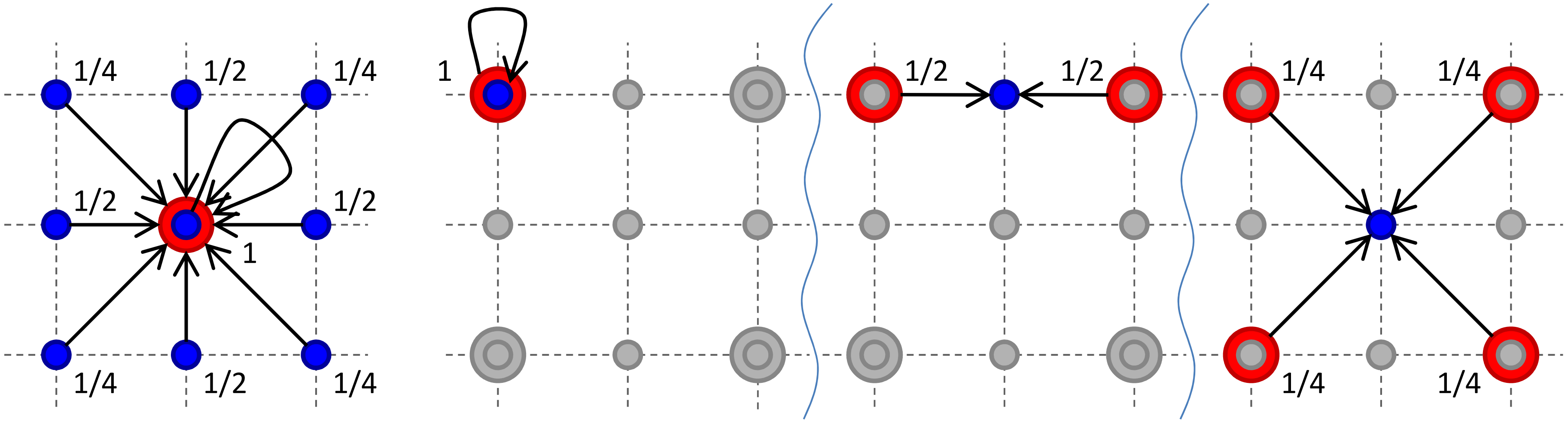
Multigrid Hierarchy

- The Cartesian grid enables to efficiently construct a nested multigrid hierarchy
  - Grids are successively built from fine to coarse levels
  - With each coarser level the cell size is doubled
  - A cell is created if it covers at least one cell on the previous finer level



Matrix-Free Multigrid Formulation

- Restriction and Interpolation
  - Weights are implicitly determined by the location of the vertices



Restriction

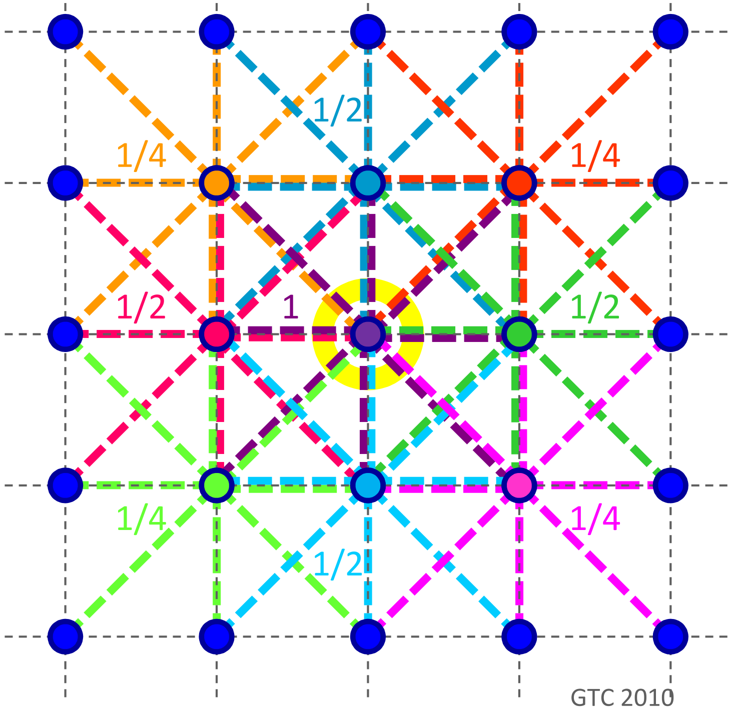
Interpolation

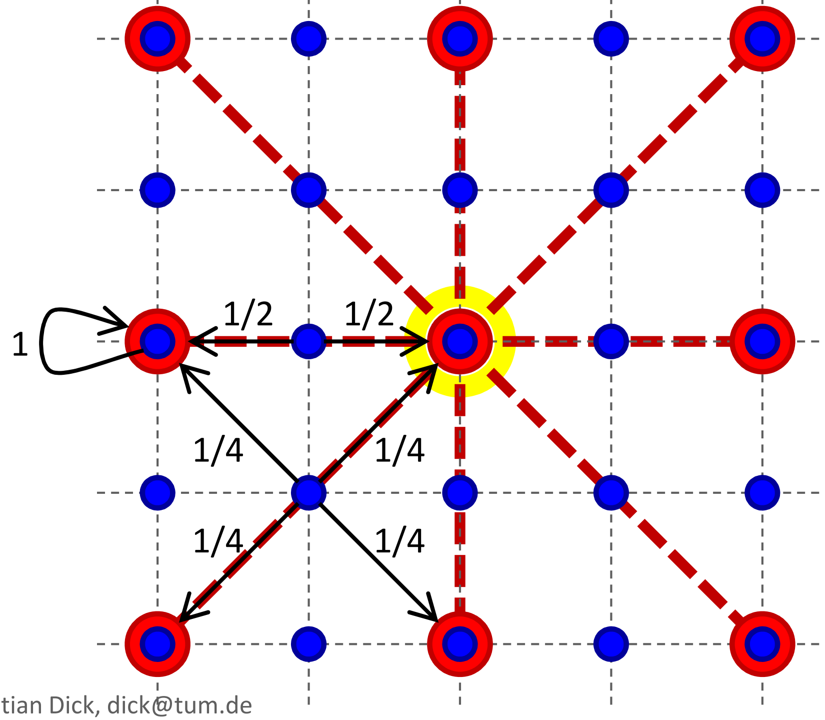
🔵 Fine grid vertices

🔴 Coarse grid vertices

Matrix-Free Multigrid Formulation

- Construction of coarse grid equations $A^{2h} = R_h^{2h} A^h I_{2h}^h$
  - Restriction: Linear combination of 3^3 fine grid equations leads to a 5^3 stencil
  - Interpolation: Substitution of fine grid unknowns reduces the stencil to a 3^3 domain



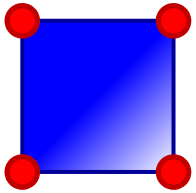


Data Structures – FE Model Topology

- Index-based representation of FE model and multigrid hierarchy
  - Finite elements and vertices are enumerated and accessed via indices
  - Typically requires significantly less memory than an index-free representation using a rectangular domain with implicit neighborhood relationships
  - Indices are represented by 32-bit integers
  - A special index value of -1 indicates “void” elements/vertices

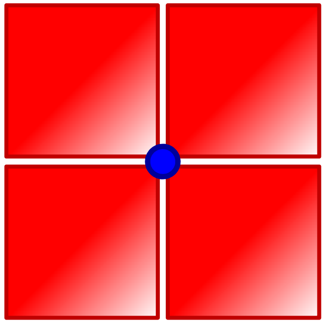
We store:

- For each finite element:



8 indices of incident vertices

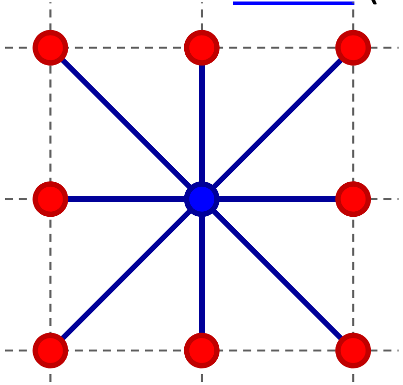
- For each simulation level vertex:



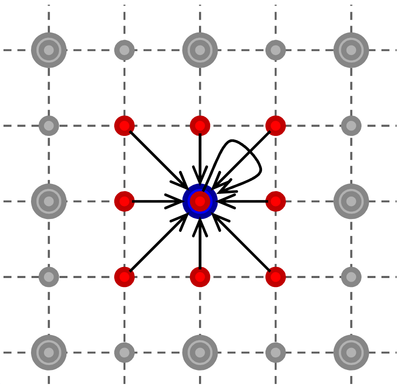
8 indices of incident elements

Data Structures – FE Model Topology

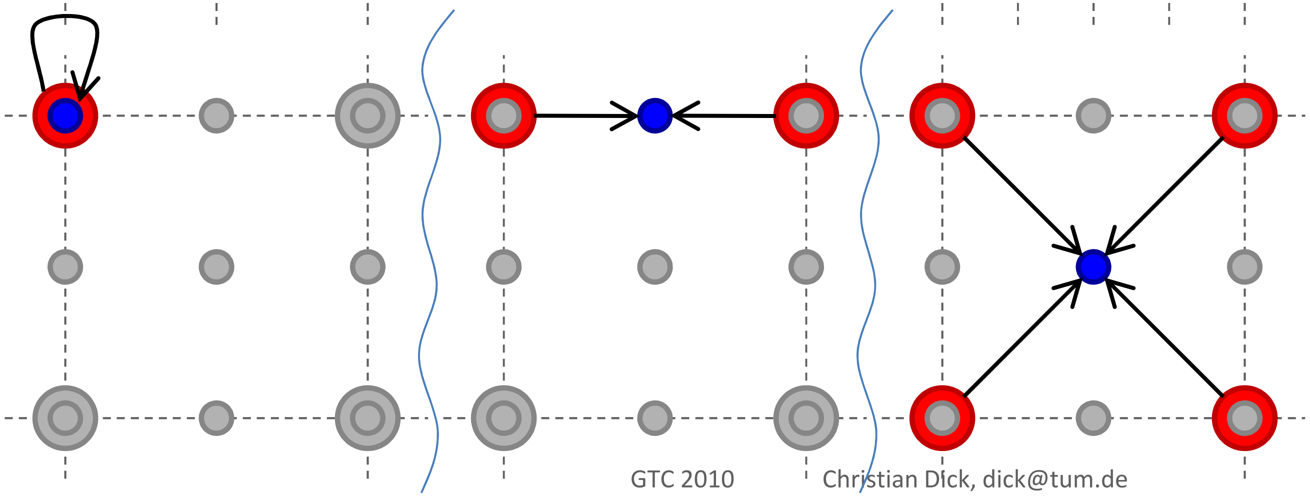
- For each vertex (all multigrid levels):



27 indices of neighbor vertices (on the same level), i.e. the 3^3 domain of the numerical stencil



27 indices of vertices which restrict to the considered vertex (on the next finer level)



Up to 8 indices of vertices which the considered vertex interpolates from (on the next coarser level)

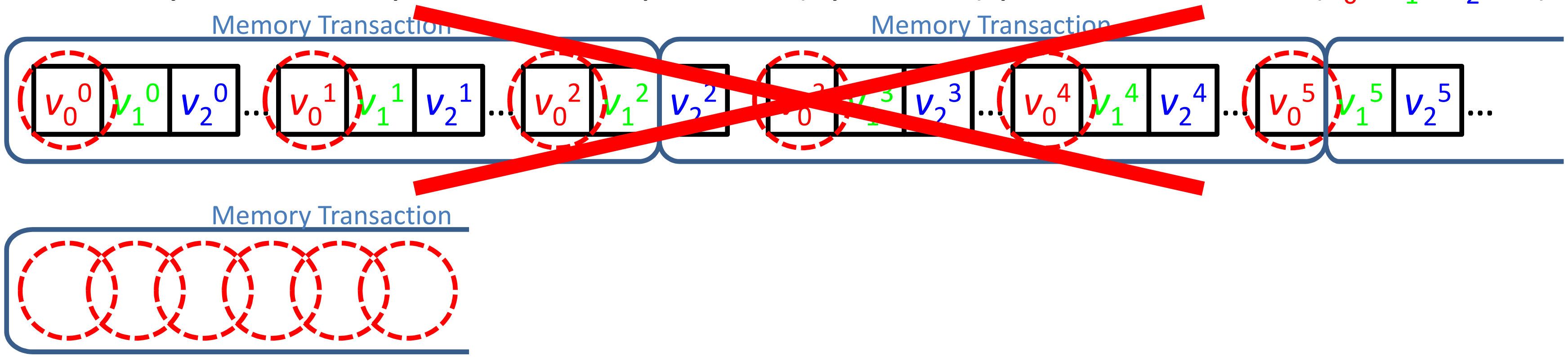
Data Structures – Simulation

- For each finite element:
  - Elastic modulus E
  - Element rotation R
  - Density ρ
- For each simulation level vertex:
  - Vertex position p^0 in undeformed state
  - Is the vertex fixed?
  - Force vector f
  - Displacement vector u^{old} of previous time step and its first and second derivatives $\dot{u}^{old}, \ddot{u}^{old}$
- For each vertex (all multigrid levels):
  - 27 3x3 matrix coefficients A_i
  - Right hand side vector b
  - Displacement vector u
  - Residual vector r

GPU-friendly Memory Layout

- Coalescing of device memory accesses
  - Fermi GPU fetches contiguous blocks of 128 bytes aligned at 128-byte boundaries
  - Threads in a warp should access successive memory addresses
- General parallelization strategy: One CUDA thread per vertex/finite element

Arrays with multiple scalar components (up to 243) per data element (v_0^i, v_1^i, v_2^i, …)

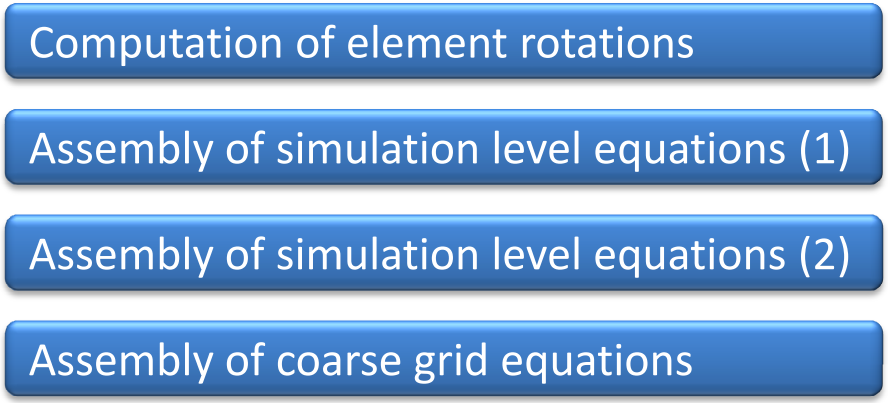

Memory Transaction
Memory Transaction
v_0^0 v_1^0 v_2^0 … v_0^1 v_1^1 v_2^1 … v_0^2 v_1^2 v_2^2 … v_0^3 v_1^3 v_2^3 … v_0^4 v_1^4 v_2^4 … v_0^5 v_1^5 v_2^5 …
Memory Transaction

- Store arrays such that their scalar components are grouped into separate memory blocks

CUDA Kernels (1)

In each time step:

- (1) Re-assemble system of equations to consider current element rotations



Computation of element rotations

Assembly of simulation level equations (1)

Assembly of simulation level equations (2)

Assembly of coarse grid equations

(Forces and dynamics)

- (2) Solve system of equations by performing multigrid V-cycles (next slide)

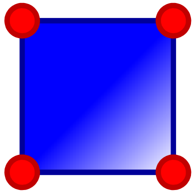
CUDA Kernels (2)

- (2) Solve system of equations by performing multigrid V-cycles

```
for (int c = 0; c < numVCycles; c++) // Perform numVCycles V-cycles per time step
{
    for (int $\ell$ = 0; $\ell$ < numLevels-1; $\ell$++) // Going down in the V-cycle from fine to coarse grids (level 0 is the finest)
    {
        for (int i = 0; i < numPreSmoothSteps; i++) { Gauss-Seidel relaxation on level $\ell$ }
        Computation of residual on level $\ell$
        Restriction of residual from level $\ell$ to level $\ell$+1
    }
    CG solver for coarsest level (i.e., level numLevels-1)
    for (int $\ell$ = numLevels-2; $\ell$ >= 0; $\ell$--) // Going up in the V-cycle from coarse to fine grids
    {
        Interpolation of error from level $\ell$+1 to level $\ell$ and coarse grid correction
        for (int i = 0; i < numPostSmoothSteps; i++) { Gauss-Seidel relaxation on level $\ell$ }
    }
}
```

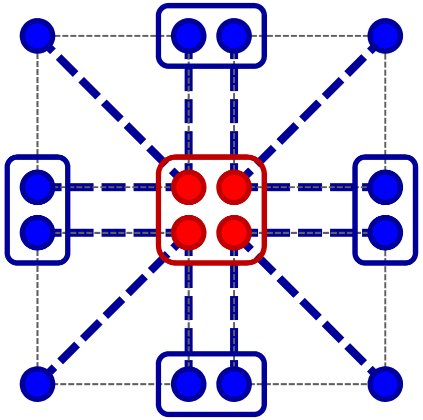
Computation of Element Rotations

- 1 CUDA thread per finite element
- *Algorithm:*
  - Fetch indices of element’s vertices
  - Fetch displacement vectors *u* at these vertices
  - Compute deformation gradient
  - Compute element rotation *R* by iterative polar decomposition
  - Store element rotation *R*



Assembly of Simulation Level Equations (1)

- 1 CUDA thread per simulation level vertex
- A single generic element stiffness matrix K^0 is stored in constant memory
  - A specific element stiffness matrix K is obtained by scaling with the element's elastic modulus E: $K = EK^0$
- *Algorithm:*
  - Host: Initialize global memory with 0
  - If vertex is fixed: Store equation $I_3 u = 0$
  - Else: Iterate over the incident elements of the vertex
    - Fetch element index
    - Fetch element rotation R and elastic modulus E
    - Iterate over element's vertices
      - Accumulate the per-vertex equation in global memory; Skip LHS part of a vertex's contribution if the vertex is fixed

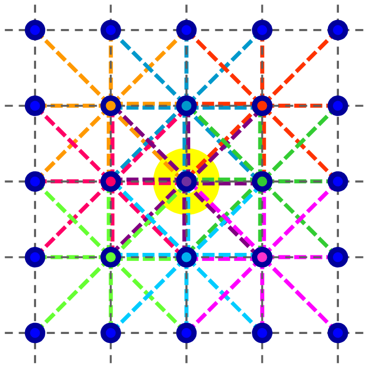


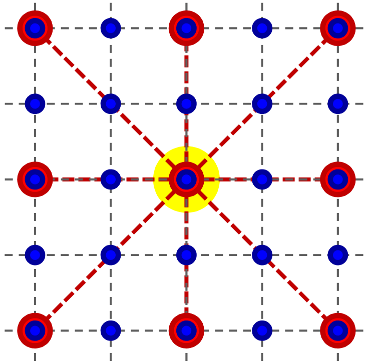
Assembly of Simulation Level Equations (2)

- 1 CUDA thread per simulation level vertex
- *Algorithm:*
  - If the vertex is not fixed:
    - Fetch $u, u^{old}, \dot{u}^{old}, \ddot{u}^{old}$
    - Update $\ddot{u}^{old}, \dot{u}^{old}, u^{old}$ and store in global memory
    - Fetch force vector *f*
    - Fetch indices of incident elements
    - Fetch densities ρ of incident elements and compute vertex mass
    - Add the contributions of the external force and of the dynamics to the equation in global memory

Assembly of Coarse Grid Equations

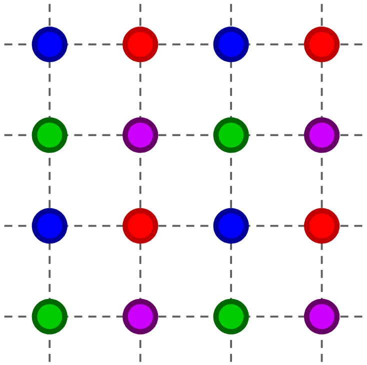
- 9 CUDA threads per vertex (each thread computes one of the nine components of all 3x3 matrix coefficients of the stencil)
- Stencil is accumulated in 27 registers and finally written to global memory
- *Algorithm:*
  - Iterate over 5^3 domain
    - Accumulate contributions of the 27 stencils at current position (restriction)
    - Distribute result to 3^3 stencil (interpolation)
- Code had to be manually unrolled to achieve optimal performance

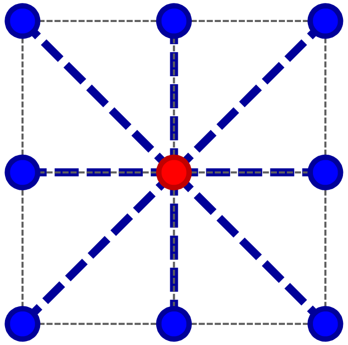




Gauss-Seidel Relaxation

- Multi-color Gauss-Seidel relaxation by partitioning the vertices into 8 subsets
  - Vertices in each subset can be relaxed in parallel, but subsets must be processed sequentially: The kernel is called once for each subset
- 1 CUDA thread per vertex of the respective subset
- *Algorithm:*
  - Fetch RHS vector *b*
  - Iterate over the 3^3 domain of neighbor vertices
      - Fetch index of neighbor vertex
      - Fetch displacement vector *u* at this vertex
      - Fetch 3x3 matrix coefficient A_i
      - Accumulate contributions
    - Relax equation and store new displacement vector *u* in global memory



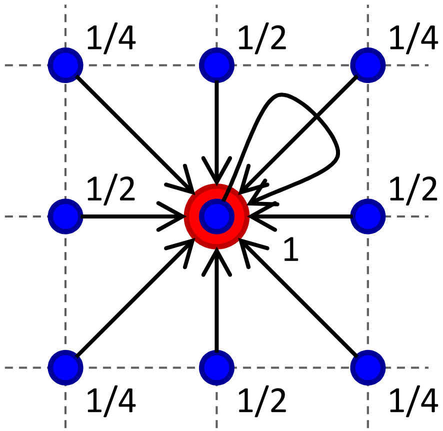


Computation of Residual

- 1 CUDA thread per vertex
- Similar to Gauss-Seidel relaxation, however now all vertices can be processed in parallel with a single kernel call

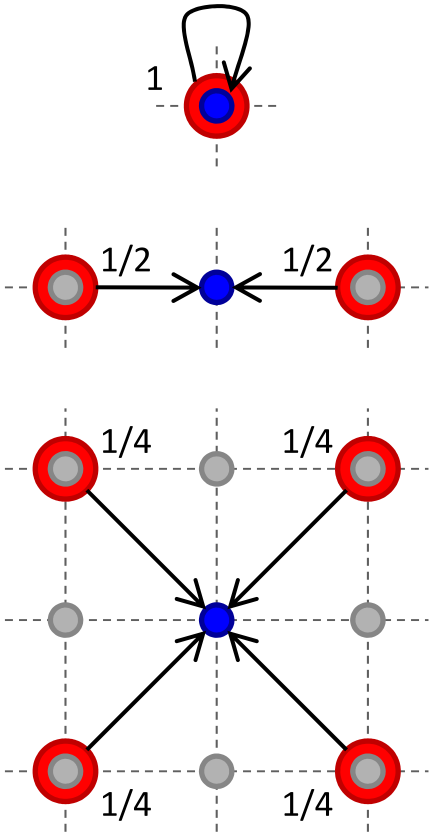
Restriction of Residual

- 1 CUDA thread per vertex
- *Algorithm:*
  - Iterate over the 27 vertices which restrict to this vertex
    - Fetch index of vertex
    - Fetch residual *r* at this vertex
    - Accumulate weighted residual
  - Store accumulated residual as right hand side vector *b* in global memory
  - Initialize displacement vector *u* with 0



Interpolation of Error and Coarse Grid Corr.

- 1 CUDA thread per vertex
- *Algorithm:*
  - Iterate over the up to 8 vertices which this vertex interpolates from
    - Fetch index of vertex
    - Fetch displacement vector *u* at this vertex
    - Accumulate weighted displacement vector
  - Add accumulated displacement vector to the displacement vector *u* in global memory (coarse grid correction)

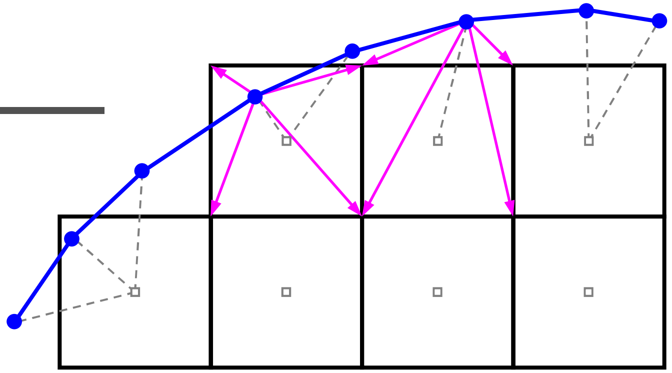


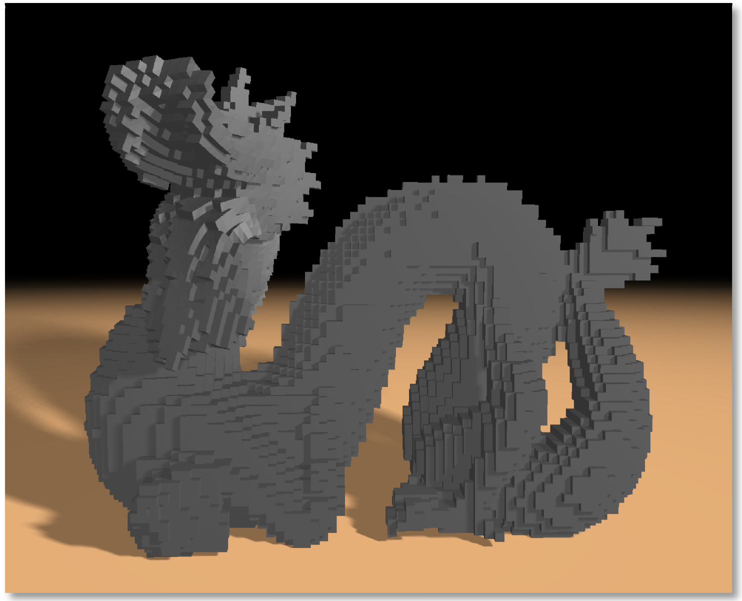
Solver for the Coarsest Level

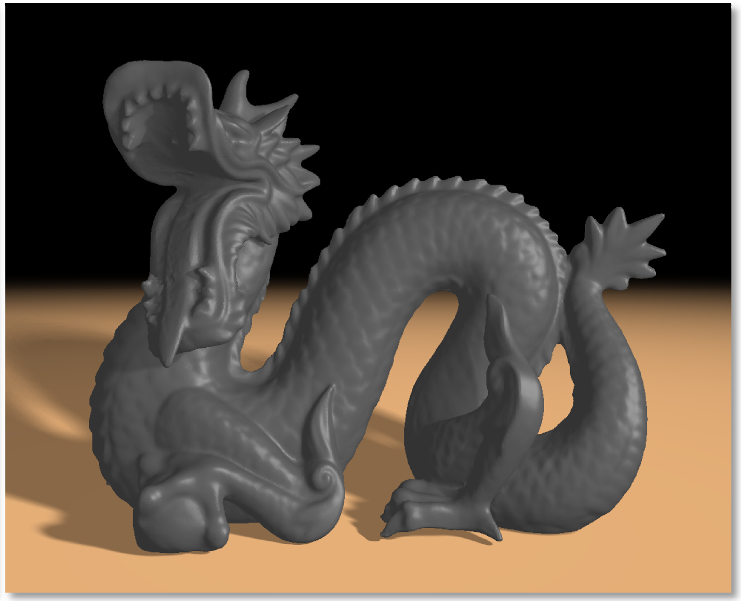
- Conjugate gradient (CG) solver with Jacobi pre-conditioner (inverse diagonal)
- Runs on a single multiprocessor (using a single thread block) to avoid global synchronization via separate kernel calls
- 1 CUDA thread per vertex
- Number of unknowns is limited by the maximum number of threads per block and by the size of the shared memory
  - Number of multigrid levels is chosen such that the number of vertices on the coarsest grid is $\leq$ 512

Rendering

- High resolution render surfaces
  - Each vertex is bound to nearest finite element
  - Deformed vertex positions are determined by tri-linear interpolation/extrapolation



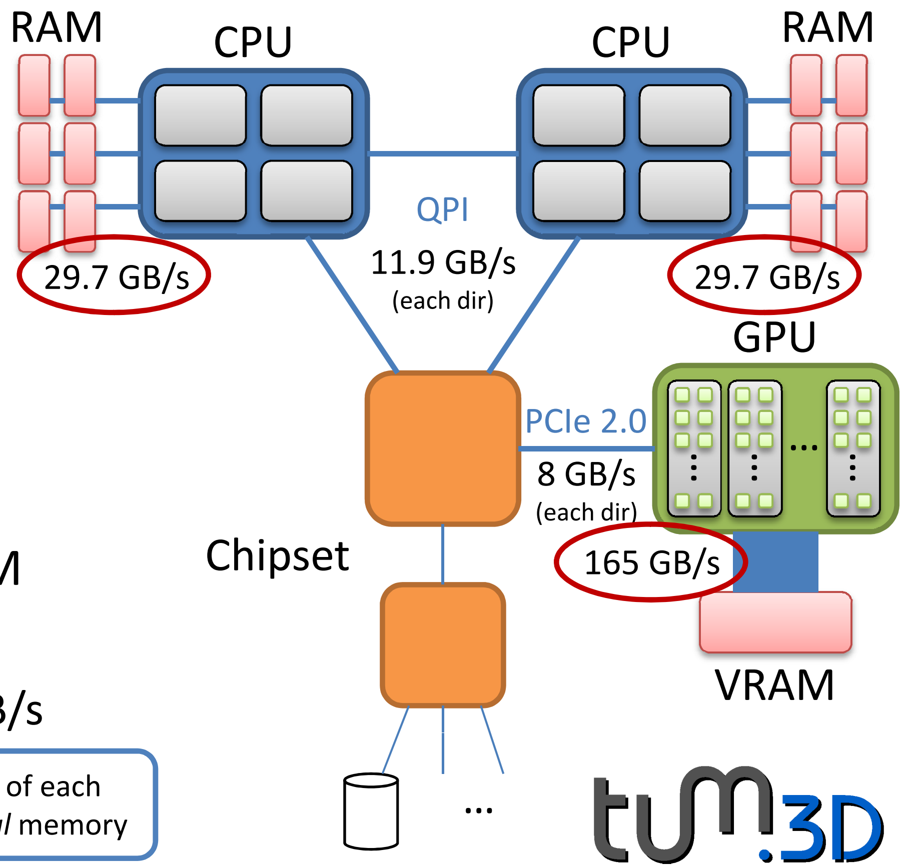




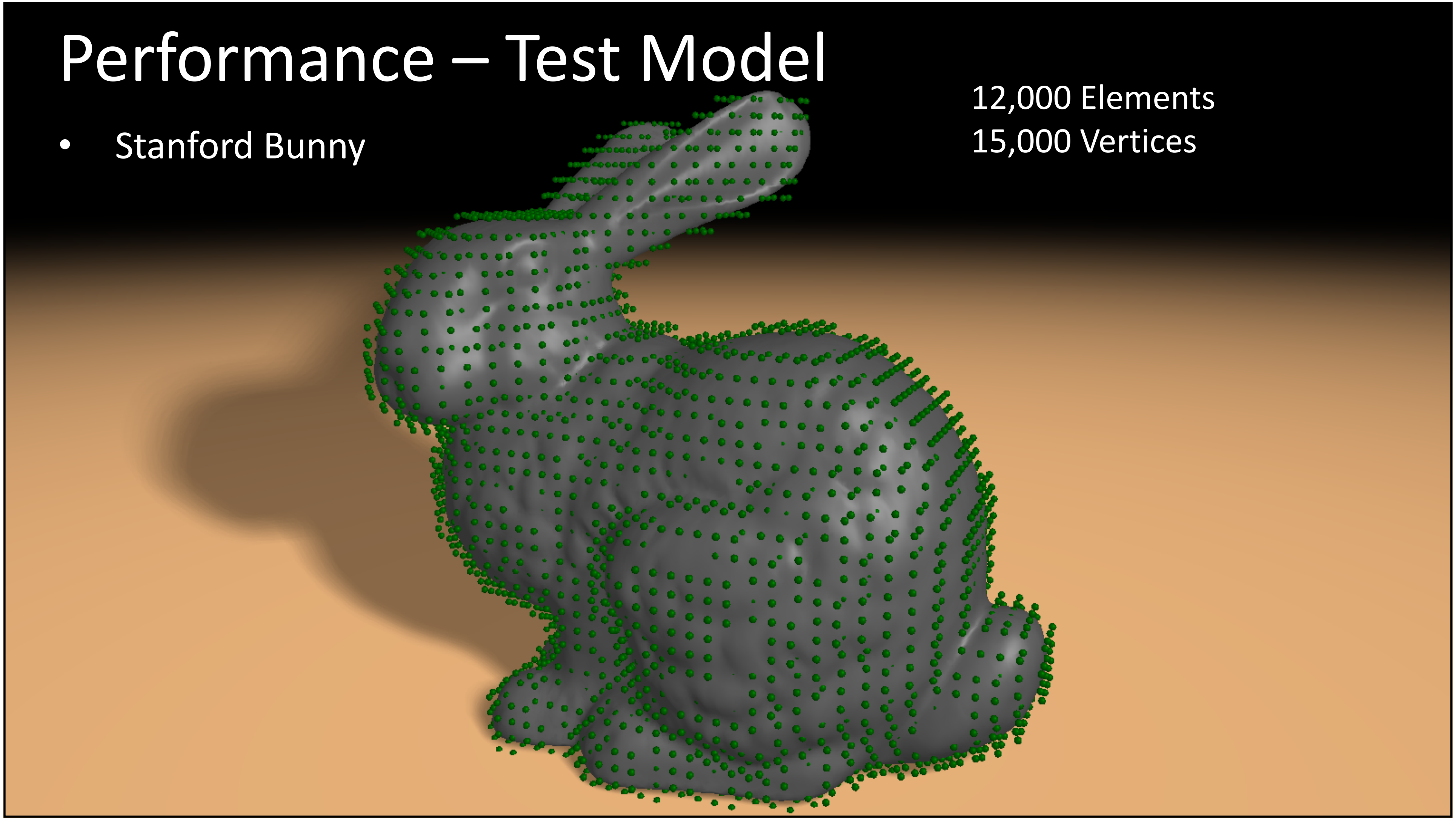
Performance – Test System

- 2x Intel Xeon X5560 2.8 GHz (3.2 GHz)
  - 4 cores per CPU
- 48 GB DDR3 1333 MHz, triple channel
  - NUMA architecture*
  - Theoretical memory bandwidth: 29.7 GB/s (per CPU)
  - Theoretical QPI bandwidth: 11.9 GB/s (each direction)
- NVIDIA GTX 480 (“Fermi”), 1.5 GB VRAM
  - 15 multiprocessors, 480 CUDA cores
  - Theoretical memory bandwidth: 165 GB/s

*For the parallelized CPU implementation, the coefficients of each per-vertex equation are stored in the respective CPU’s *local* memory



tum.3D computer graphics & visualization



Performance – Test Model

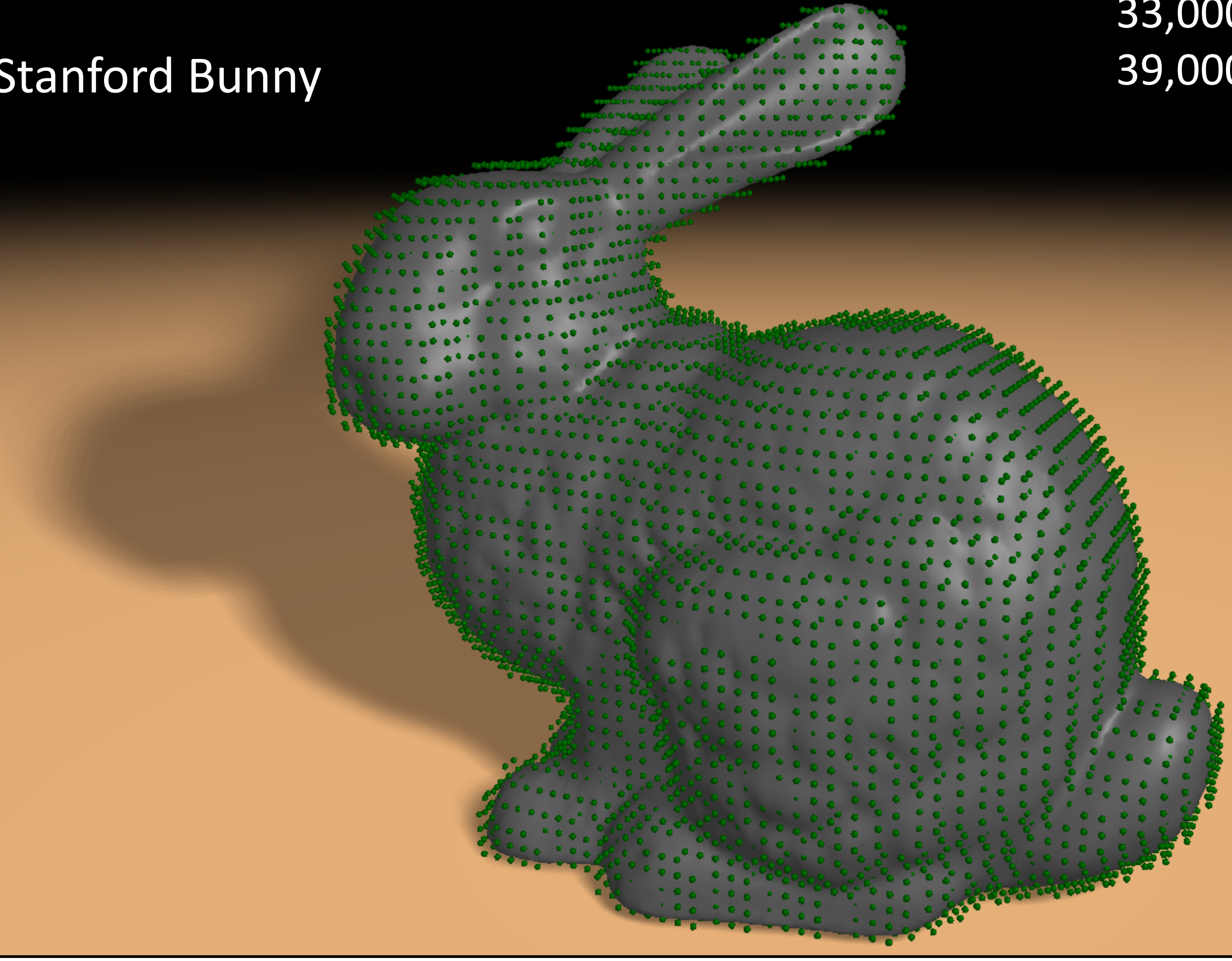
- Stanford Bunny

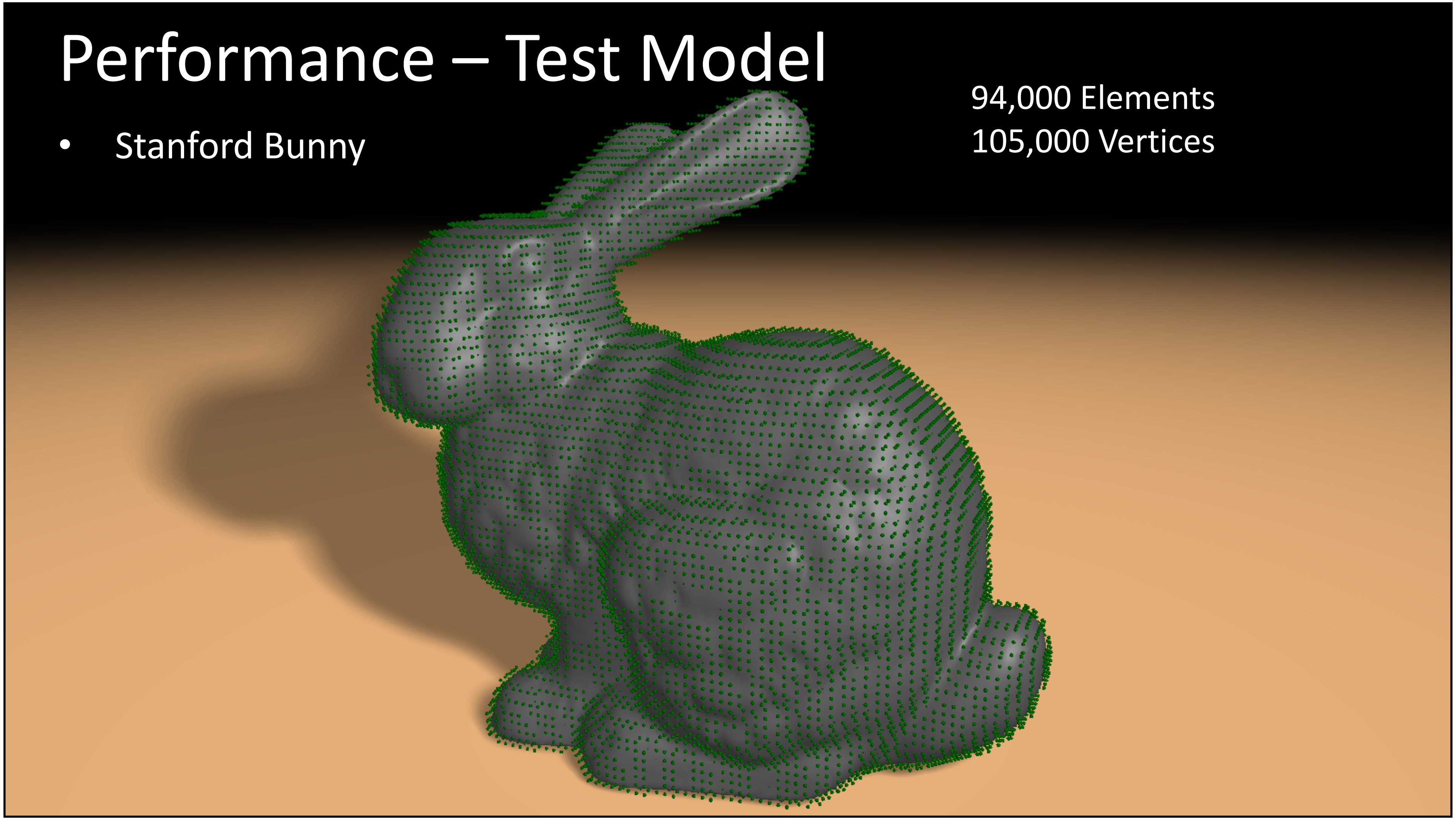
12,000 Elements
15,000 Vertices

Performance – Test Model

- Stanford Bunny

33,000 Elements
39,000 Vertices





Performance – Test Model

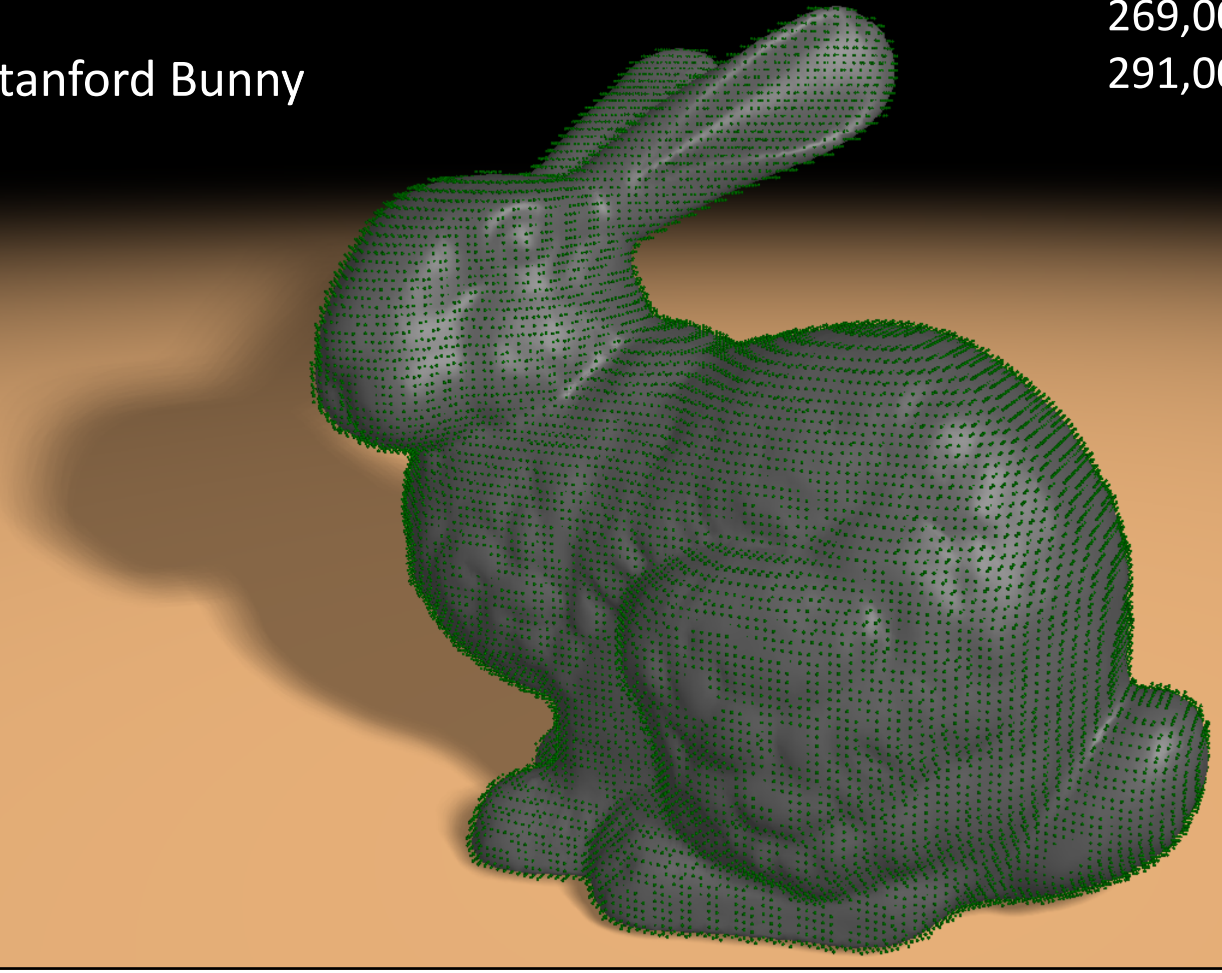
- Stanford Bunny

94,000 Elements
105,000 Vertices

Performance – Test Model

- Stanford Bunny

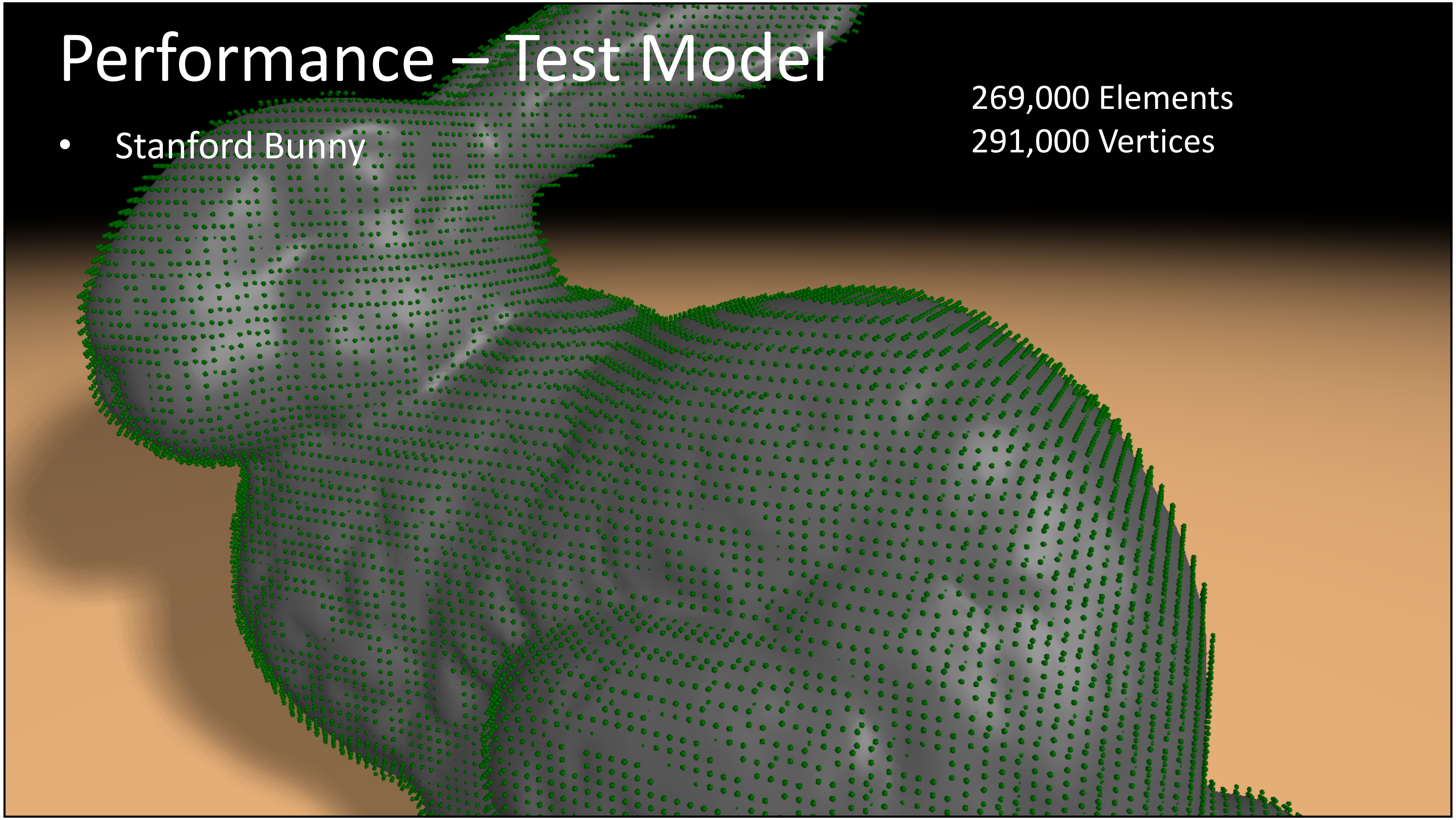
269,000 Elements
291,000 Vertices



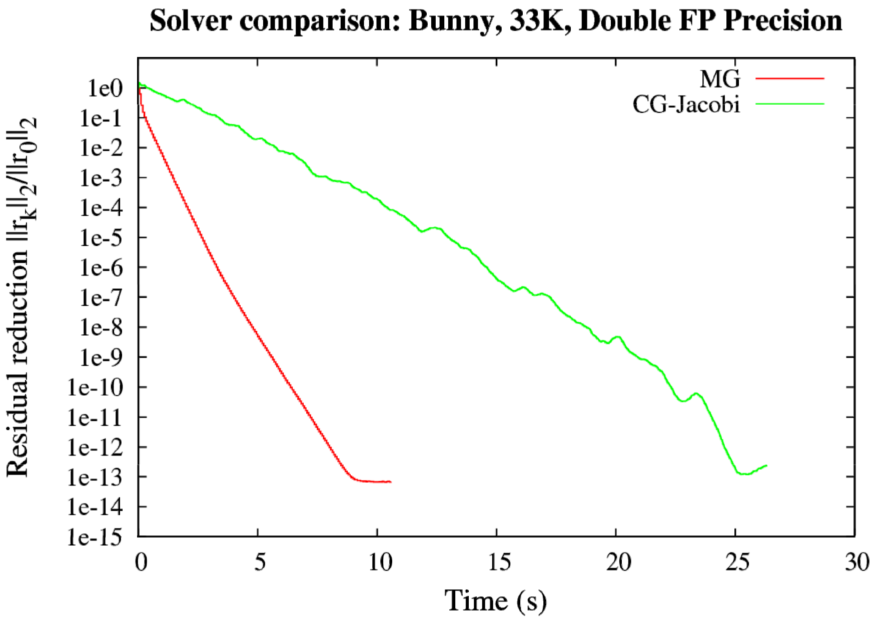
Performance – Test Model

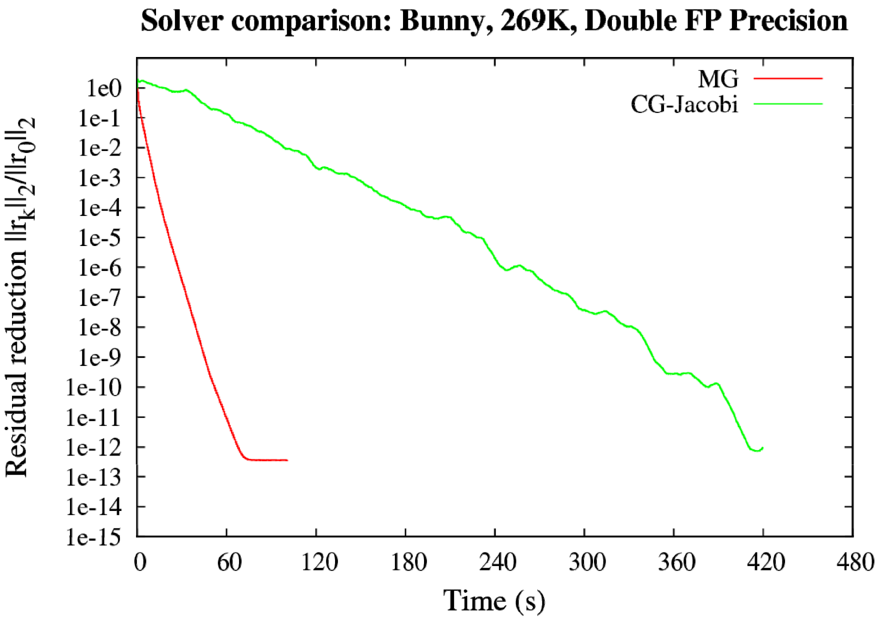
- Stanford Bunny

269,000 Elements
291,000 Vertices



Performance – Solver Convergence



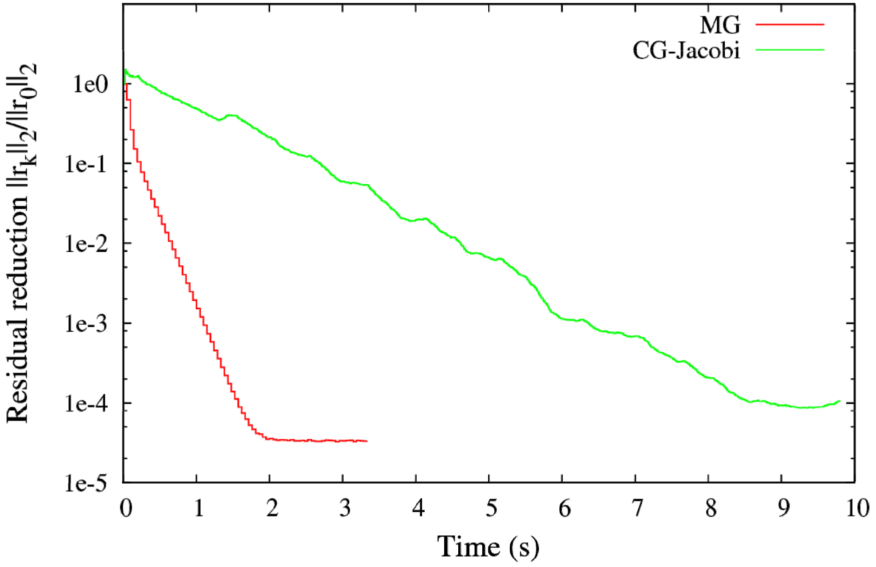


Timings were obtained on the CPU using 1 core

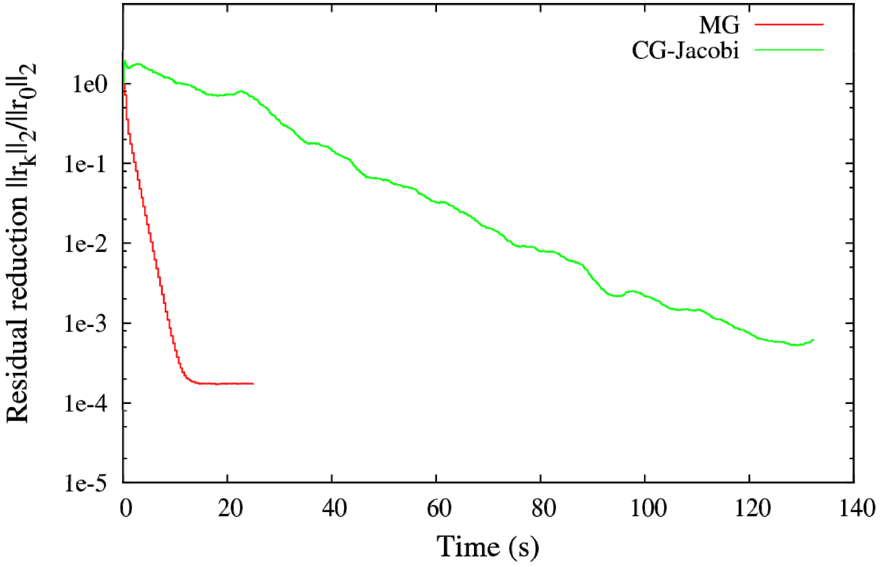
$E = 10^6 Pa$, $\nu = 0.3$, $\rho = 10^5 kg / m^3$, $dt = 50ms$, hexahedra edge length $= 2.8mm$ and $1.4mm$

Performance – Solver Convergence

Solver comparison: Bunny, 33K, Single FP Precision



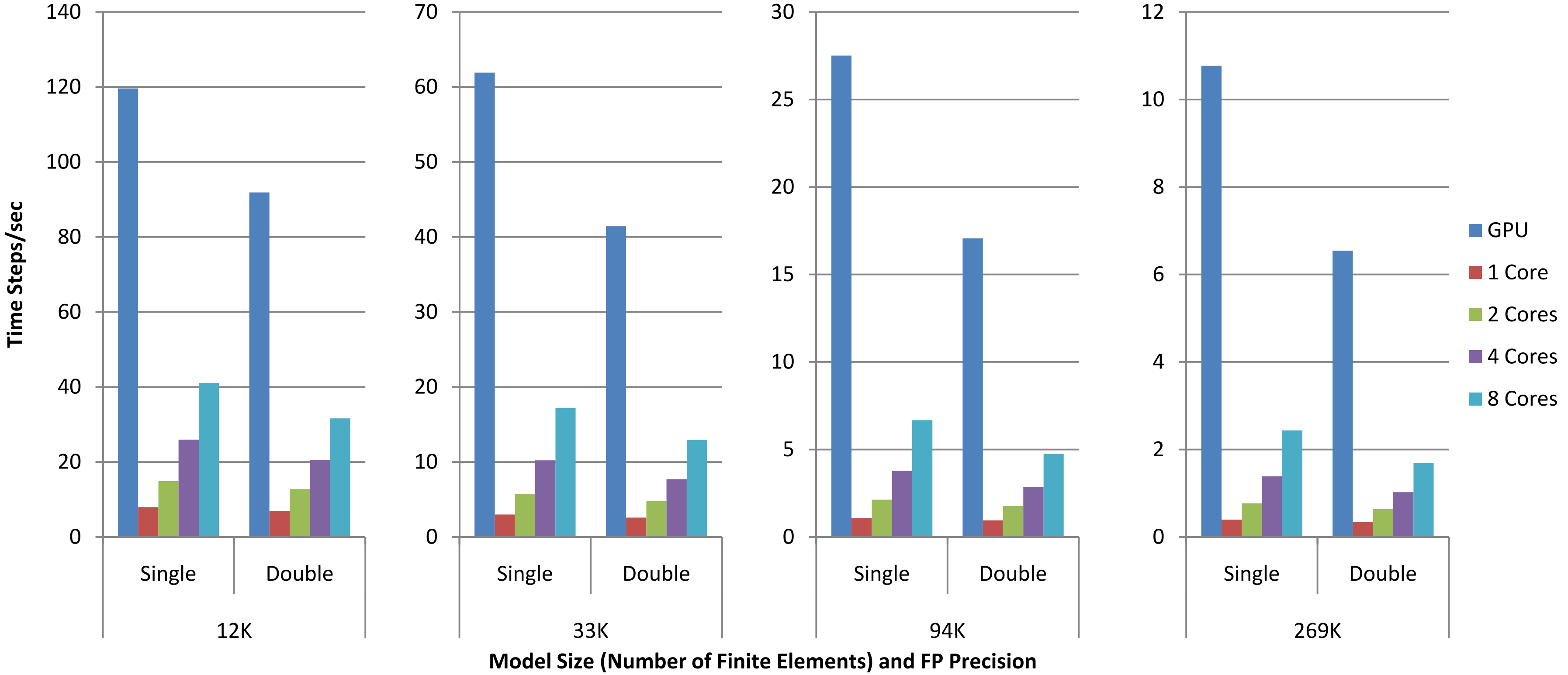
Solver comparison: Bunny, 269K, Single FP Precision



Timings were obtained on the CPU using 1 core

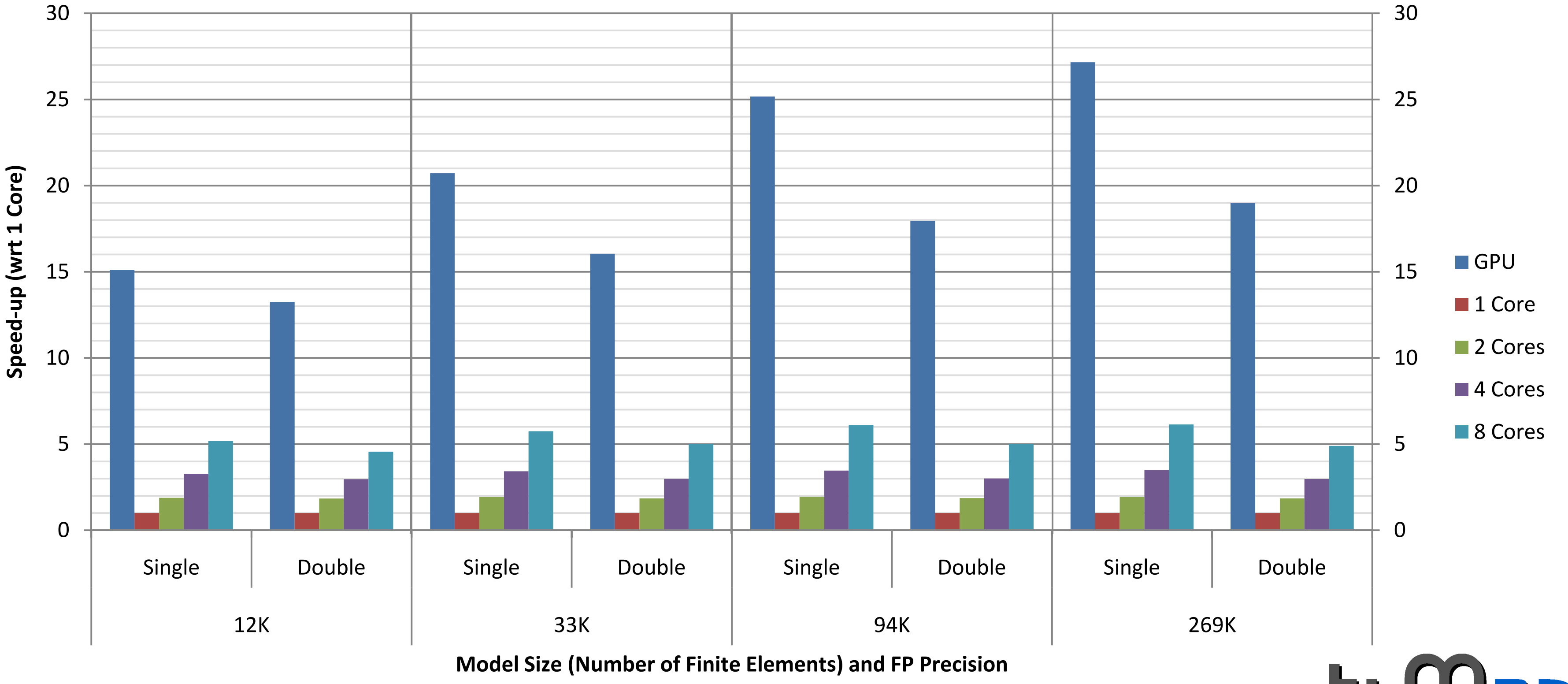
$E = 10^6 \, Pa$, $\nu = 0.3$, $\rho = 10^5 \, kg / m^3$, $dt = 50ms$, hexahedra edge length $= 2.8mm$ and $1.4mm$

Performance – Simulation Time Steps/sec



Each time step includes the re-assembly of the per-vertex equations (simulation level + coarse grids) for the co-rotational strain formulation and 2 multigrid V-cycles, each with 2 pre- and 1 post-smoothing Gauss-Seidel steps

Performance – Speed-up



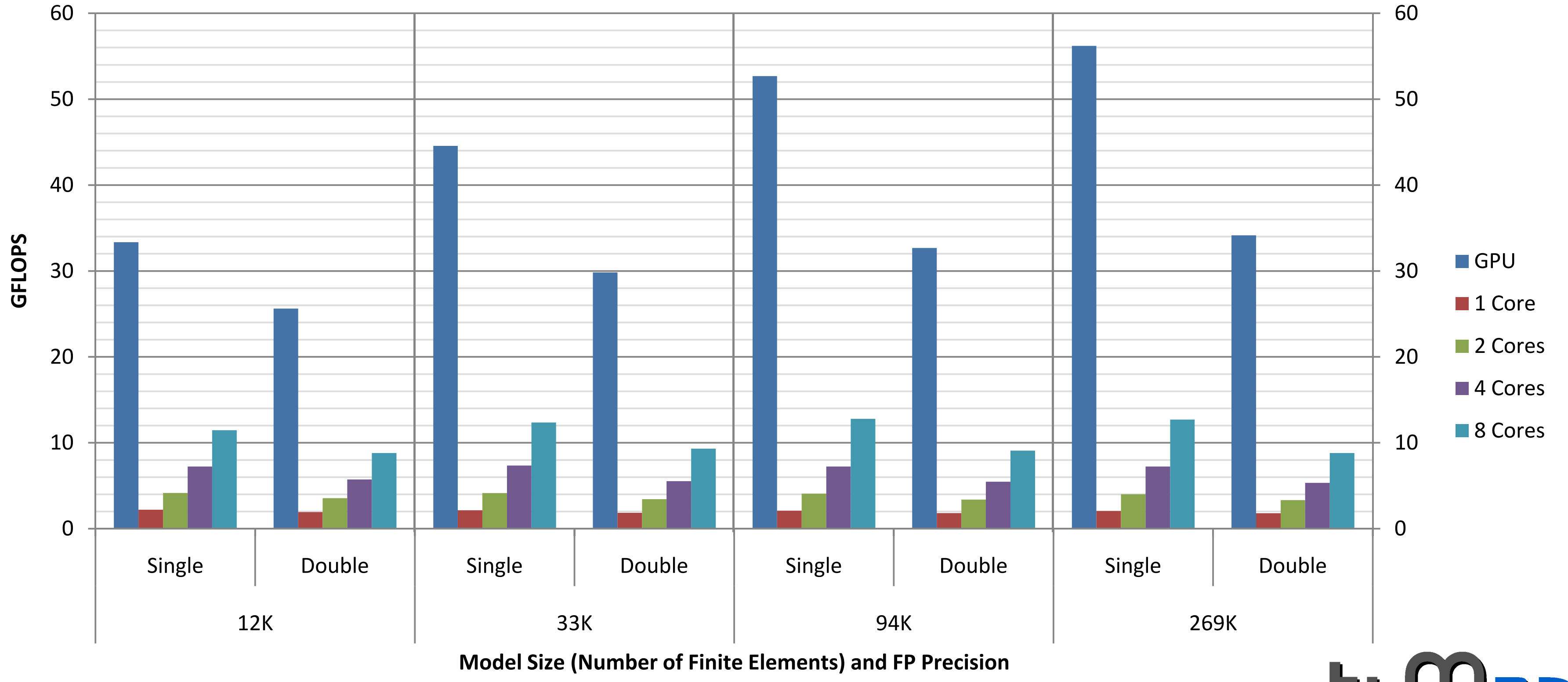
Performance – Costs per Finite Element

- Amortized costs per finite element per time step (Bunny 269K elements)

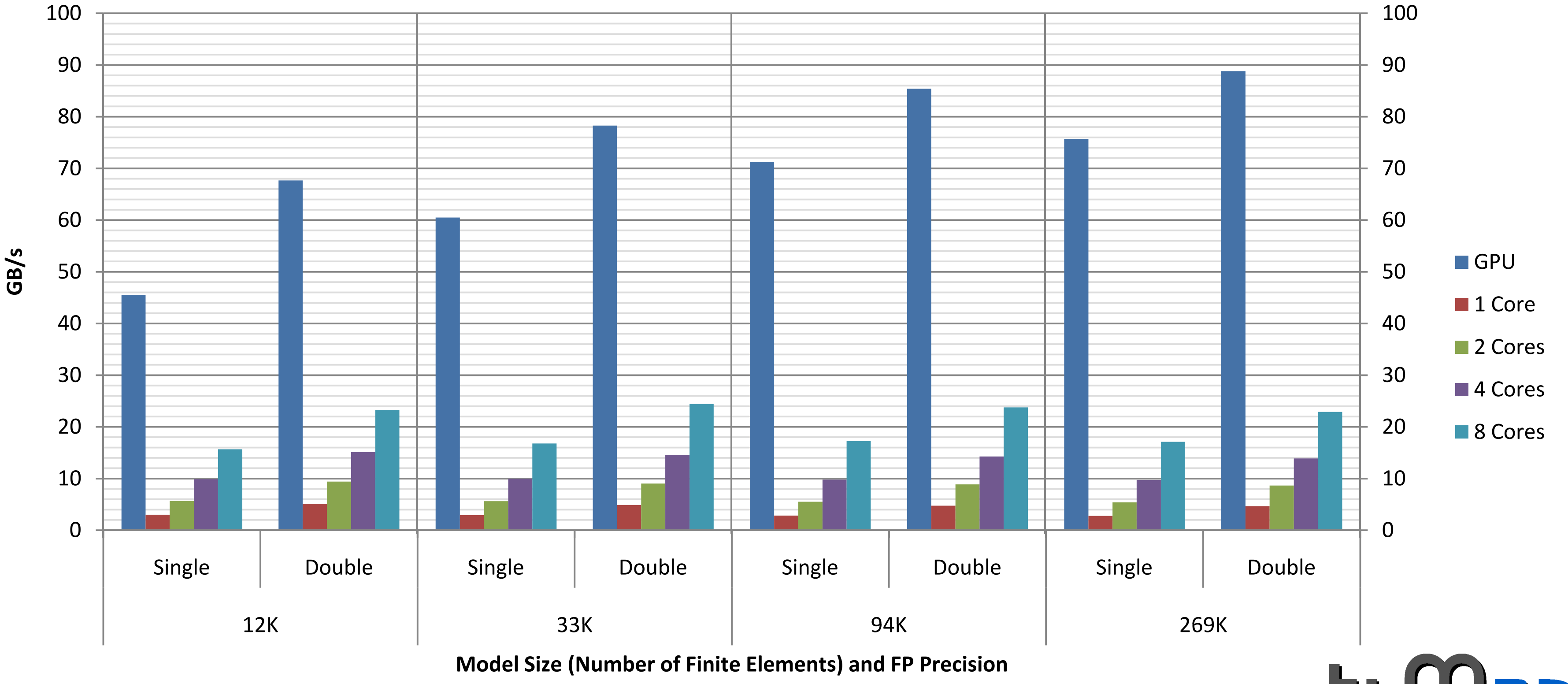
2 V(2,1)-cycles per time step

Kernel	FLOPs	%	Bytes R/W		%	GPU Timings %	
			Single	Double		Single	Double
Computation of element rotations	470	2	160	300	1	0	1
Assembly of simulation level equations (1)	10000	51	6600	13000	23	33	30
Assembly of simulation level equations (2)	32	0	140	270	0	0	0
Assembly of coarse grid equations	3200	17	5100	10000	18	24	21
Gauss-Seidel relaxation	6 x 660	20	6 x 1800	6 x 3500	39	27	33
Computation of residual	2 x 620	6	2 x 1800	2 x 3500	13	11	12
Restriction of residual	2 x 210	2	2 x 580	2 x 1000	4	1	1
Interpolation of error and coarse grid corr.	2 x 39	0	2 x 200	2 x 350	1	2	1
CG solver on coarsest level	2 x 3	0	2 x 1	2 x 1	0	1	1
Total (per finite element per time step)	19000		28000	54000			

Performance – GFLOPS *(sustained)*



Performance – Memory Throughput *(sustained)*



Conclusion and Future Work

- Real-time FEM simulation of deformable objects enabled by a fully GPU-based geometric multigrid solver
  - Hexahedral finite elements on a uniform Cartesian grid, co-rotational strain
  - Regular shape of stencil enables GPU-friendly parallelization and memory accesses
  - Performance is boosted by the GPU's compute power and memory bandwidth
    - Up to 27x faster than 1 CPU core / 4x faster than 8 CPU cores
    - Up to 56 GFLOPS (single) / 34 GFLOPS (double precision) *(sustained performance)*
    - Up to 88 GB/s memory throughput *(sustained performance)*
    - Real-time/interactive simulation rates:
120 time steps/sec for 12,000 elements, 11 time steps/sec for 269,000 elements
- Future work
  - GPU-based collision detection
  - Parallelization on multiple GPUs

Thanks for your attention!

- http://wwwcg.in.tum.de/Research/Publications/CompMechanics

C. Dick, J. Georgii, R. Westermann. A Real-Time Multigrid Finite Hexahedra Method for Elasticity Simulation using CUDA. Technical Report, July 2010.

Demo Executables