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 $u : \Omega \to \mathbb{R}^3$

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

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

$\varepsilon : \text{Strain}$

$\sigma : \text{Stress}$

$f : \text{External forces}$

GTC 2010    Christian Dick, dick@tum.de
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 $i$
$\varphi_i$ : Basis function for Node $i$

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

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

$u^e$ : Linearized $u_i$
$f^e$ : Linearized $f_i$

\[ B^T D B \, u^e = f^e \]

$K^e$ : Element stiffness matrix
$D$ : Material law
Elasticity & FEM

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

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

![Diagram showing target voxel grid, k-Buffer capturing fragments, and depth k-Buffer with numerical representation.](image)
Per-Element Equations (static)

- Per-element equations

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

\[ u_i : \text{Displacements} \]
\[ f_i : \text{Forces} \]

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

---

Forces:  
\[ u_1, f_1 \]
\[ u_2, f_2 \]
\[ u_3, f_3 \]
\[ u_4, f_4 \]
Per-Element Equations (co-rotated, static)

• Co-rotated strain

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

\[
\sum_{j=1}^{8} RK_{ij} R^T u_j = f_i - \sum_{j=1}^{8} RK_{ij} (R^T p_j^0 - p_j^0)
\]

• \( R \) is obtained by polar decomposition of the element’s average deformation gradient (5 iterations)

\[
R_0 = l_3 + \frac{1}{4} \sum_{i=1}^{8} u_{iold}^T \begin{pmatrix} \pm \frac{1}{a}, \pm \frac{1}{b}, \pm \frac{1}{c} \end{pmatrix}
\]

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

\( a, b, c \): Edge lengths of hexahedral element

\( \text{Signs} : \) \((-,+), (+,+), (-,-), (+,-)\)

\( R : \) Element rotation

\( p_j^0 : \) Undeformed vertex positions
Per-Element Equations (co-rotated, dynamic)

- **Dynamics**
  \[
  \sum_{j=1}^{8} (M_{ij} \ddot{u}_j + C_{ij} \dot{u}_j + \hat{A}_{ij} u_j) = \hat{b}_i \quad , \quad i = 1, \ldots, 8
  \]
  \(M:\) Mass matrix  
  \(C:\) Damping matrix

- **Newmark time integration**
  \[
  \ddot{u}_i = \frac{2}{\Delta t^2} (u_i - u_i^{old}) - \dot{u}_i^{old}, \quad \dot{u}_i = \frac{4}{\Delta t^2} (u_i - u_i^{old} - \dot{u}_i^{old} \Delta t) - \dot{u}_i^{old}
  \]
  \(\Delta t:\) Time step length

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

- **Mass proportional damping:** \(C_{ij} = \alpha M_{ij}\)
- **Mass lumping:** \(M_{ii} = m_i l_3\), \(M_{ij} = 0\) 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

$$\sum_{i=(-1,-1,-1)^T}^{(1,1,1)^T} A_i^x u_{x+i} = b_x$$

- Dirichlet boundary conditions: Replace equation by $l_3 u_x = 0$ and in the other equations set the respective coefficient $A_i$ corresponding to this vertex to 0
Geometric Multigrid

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

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

$\Omega^h$
Geometric Multigrid

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

  \[
  \text{Relax } A^h v^h \approx b^h \quad \text{Pre-smoothing}
  \]
  \[
  \text{Residual } r^h = b^h - A^h v^h
  \]

  Restrict
  \[
  r^{2h} = R^h_r r^h
  \]

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

  \[
  \text{Interpolate } e^h = I^{2h}_r e^{2h}
  \]

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

  Post-smoothing

  Coarse grid correction

\[\]
Geometric Multigrid

- Solve $A^h u^h = b^h$, current approximate solution $v^h$
  
  Relax $A^h v^h \approx b^h$
  
  Residual $r^h = b^h - A^h v^h$
  
  Restrict $r^{2h} = R^h r^h$
  
  Multigrid V-Cycle
  
  Interpolate $e^h = I^{2h} e^{2h}$
  
  Coarse Grid Solver

- Post-smoothing
  
  Correct $v^h \leftarrow v^h + e^h$

- Linear time complexity in the number of unknowns
Variational Properties of Multigrid

• Selection of transfer operators and coarse grid operators:

\[ I_{2h}^h : \text{Tri-linear interpolation} \]
\[ R_{2h}^h = (I_{2h}^h)^T \quad \text{Restriction is transpose of interpolation} \]
\[ A_{2h}^h = R_{2h}^h A^h I_{2h}^h \quad \text{Galerkin-based coarsening} \]

1D Example:
Fine Grid

Coarse Grid

Interpolation

Restriction

• 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

GTC 2010 Christian Dick, dick@tum.de
Matrix-Free Multigrid Formulation

- Construction of coarse grid equations $A^{2h} = R^{2h}_h A^h_{2h}$
  - 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

GTC 2010 Christian Dick, dick@tum.de
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 $\rho$

• 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 $u^{old}$, $\dot{u}^{old}$

• 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 \((v_0^i, v_1^i, v_2^i, \ldots)\)

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

GTC 2010    Christian Dick, dick@tum.de
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) (Forces and dynamics)
  - Assembly of coarse grid equations

- (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

```c
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 $l_3u = 0$
  - 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

GTC 2010  Christian Dick, dick@tum.de
Assembly of Simulation Level Equations (2)

• 1 CUDA thread per simulation level vertex

• Algorithm:
  – If the vertex is not fixed:
    • Fetch $u, u^{old}, \ddot{u}^{old}, \dddot{u}^{old}$
    • Update $\ddot{u}^{old}, \dddot{u}^{old}, u^{old}$ and store in global memory
    • Fetch force vector $f$
    • Fetch indices of incident elements
    • Fetch densities $\rho$ 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

- 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.
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

\( E = 10^6 \text{Pa}, \ \nu = 0.3, \ \rho = 10^5 \text{kg} / \text{m}^3, \ dt = 50\text{ms}, \ \text{hexahedra edge length} = 2.8\text{mm} \) and \( 1.4\text{mm} \)
Performance – Solver Convergence

Timings were obtained on the CPU using 1 core

\[ E = 10^6 \text{ Pa}, \quad \nu = 0.3, \quad \rho = 10^3 \text{ kg/m}^3, \quad dt = 50\text{ms}, \quad \text{hexahedra edge length} = 2.8\text{mm and 1.4mm} \]
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

GTC 2010    Christian Dick, dick@tum.de
# Performance – Costs per Finite Element

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

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

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

GTC 2010        Christian Dick, dick@tum.de
Performance – GFLOPS (*sustained*)

Model Size (Number of Finite Elements) and FP Precision

- GPU
- 1 Core
- 2 Cores
- 4 Cores
- 8 Cores

GTC 2010    Christian Dick, dick@tum.de
Performance – Memory Throughput (sustained)

![Bar chart showing memory throughput for different model sizes and precision levels for GPU, 1, 2, 4, and 8 cores.](chart.png)

Model Size (Number of Finite Elements) and FP Precision

- **Single**: 12K, 33K, 94K, 269K
- **Double**: 12K, 33K, 94K, 269K

**Axes**:
- **Y-axis**: GB/s
- **X-axis**: Model Size (Number of Finite Elements) and FP Precision

**Legend**:
- **GPU**
- **1 Core**
- **2 Cores**
- **4 Cores**
- **8 Cores**

**Note**: The chart illustrates the sustained memory throughput for various model sizes and precision levels, comparing different core configurations for GPUs.
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://www.cg.in.tum.de/Research/Publications/CompMechanics