Parallelizing a Real-Time 3D Finite Element Algorithm using CUDA: Limitations, Challenges and Opportunities.

Vukasin Strbac
Biomechanics section
KU Leuven
The Finite Element Method and the GPU

Implicit

Basically two phases:

1. Assembly: computation of elemental stiffness matrices and scatter into large global stiffness matrix
   (a bit more interesting cfr. Luitjens GTC2012)

2. Solve the system to obtain displacements (well treated already)

Explicit

Loop:

1. Compute forces directly from a material model on a per-element basis

2. Update displacements explicitly (e.g. central differences)

3. Impose boundary conditions, incrementally

(similar to element stiffness matrix computation)
Real-time application of Finite Elements

- Pulling aside the renal vein
- Clamping the renal artery
- Inflation of an endoclamp balloon

Active constraints:
- Design a new, safer clamp?
- Monitor inflation pressure in the balloon?
- …
Real-time FE in the surgical theater

- ‘Limited’ number of DOFs is fine
  - Driven by pathological conditions of tissue which can emerge even on coarser meshes.

- Nonlinear, both in geometry and materials

- Quick response
  - Explicit solution gives more opportunity to check values

- Why not use a CPU cluster?
  - Benefit from a tightly-coupled system (especially explicit computation)
Real-time TLED, aortic expansion example

- Simulation at 17 fps
  ~70ms per solution, 1000 timesteps each
- Loading is 0-500mmHg sine wave
- Mimicking endoclamp balloon expansion
- Neo-Hookean material with $E=3000\text{Pa}$, $\nu=0.49$
### Total Lagrangian Explicit Dynamic Dynamic (TLED)

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute time-step;</td>
<td></td>
</tr>
<tr>
<td>Compute shape function derivatives;</td>
<td></td>
</tr>
<tr>
<td>for each time-step{</td>
<td></td>
</tr>
<tr>
<td>computeForces();</td>
<td></td>
</tr>
<tr>
<td>updateDisplacements();</td>
<td></td>
</tr>
<tr>
<td>enforceBoundaryConditions();</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- **3D low order elements:**
  - 8-node hexahedron
  - Trilinear shape functions
- **Low order integration:** single integration point

Avezedo 1999
**TLED forces computation**

```plaintext
computeForces(u, ∇N, El, V, matP)
{
    load(El); // element info
    load(u,(El)); // displacements of that element
    load(∇N); // shape function derivatives
    X=f(u, ∇N); // deformation gradient
    C=f(X); // cauchy-green deformation tensor
    Cinv = f(C); // its inverse
    S=f(Cinv, matP); // 2PK stress
    load(V); // volumes
    F=f(S,V,X,∇N); // final forces
    store(F, (El)); // store
}
```
TLED memory considerations

```c
computeForces(u, \nabla N, E1, V, matP) {
    load(E1); // coalesced read +8.
    load(u, (E1)); // uncoalesced read +24
    load(\nabla N); // coalesced read +24
    X = f(u, \nabla N); // +9
    C = f(X); // +9
    Cinv = f(C); // +9
    S = f(Cinv, matP); // +6
    load(V); // coalesced +1
    F = f(S, V, X, \nabla N); // +24
    store(F, (E1)); // Uncoalesced store
}
```

- coalesced read +8
- uncoalesced read +24
- coalesced read +24
- +9
- +9
- +9
- +6
- coalesced +1
- +24
- Uncoalesced store

= ~128+ registers for the simplest case
Memory analysis on C2075

<table>
<thead>
<tr>
<th>Memtype</th>
<th>32</th>
<th>64</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register words</td>
<td>63</td>
<td>63</td>
<td>42</td>
</tr>
<tr>
<td>L1 (48kB per SM)</td>
<td>192B</td>
<td>96B</td>
<td>64B</td>
</tr>
<tr>
<td>L2 (768kB total)</td>
<td>219B</td>
<td>109B</td>
<td>19B</td>
</tr>
<tr>
<td>L1 words</td>
<td>48</td>
<td>87</td>
<td>58</td>
</tr>
<tr>
<td>L2 words</td>
<td>54</td>
<td>114</td>
<td>76</td>
</tr>
</tbody>
</table>

--maxrregcount impose blockSize

- Registers -> L1 -> L2 -> global
  - Registers spilling everywhere!
  - Currently blockSize = 32
  - blockSize of 64 would spill all the way to global memory.

Keeping local memory away from gmem very beneficial

you are here! at ~129

- Registers
- L1 cache
- L2 cache
- Global memory

Vukasin Strbac
GTC2014
3/26/2014
9/21
Conclusion for c2075:

- Currently lots of execution dependency stalls due to “few” warps in flight (IPC: ~0.73, C2075)
- Increasing number of resident warps -> better latency hiding
  - But have to get data from global memory!
- Does not pay off.
computeForces(u, ∇N, El, V, matP) {
    load(El);
    load(u, (El));
    load(∇N);
    X = f(u, ∇N);
    C = f(X);
    Cinv = f(C);
    S = f(Cinv, matP);
    load(V);
    F = f(S, V, X, ∇N);
    store(F, (El));
}

Lifetime of some variables not friendly!
Pseudocode, computing forces

```c
computeForces(u, ∇N, El, V, matP) {
    load(El);
    load(u, (El));
    load(∇N);
    X = f(u, ∇N);
    C = f(X);
    Cinv = f(C);
    S = f(Cinv, matP);
    load(V);
    F = f(S, V, X, ∇N);
    store(F, (El));
}
```

Lifetime of variables not friendly!

Can we split the kernel?

---

Store here

Load here
Pseudocode, computing forces, splitting

```plaintext
computeFstore(u, ∇N, El, V, matP) {
    load(El);
    load(u, (El));
    load(∇N);
    X = f(u, ∇N);
    store(X);
    C = f(X);
    Cinv = f(C);
    S = f(Cinv, matP);
    store(S);
}
```

```plaintext
computeFread(S, X, ∇N, V, El) {
    load(X);
    load(S);
    load(∇N);
    F = f(S, V, X, ∇N);
    load(El);
    store(F, (El));
}
```

- Register req. go down
- Occupancy goes up
- But added +47/15 fetches/stores
- Doesn't pay off
  + maxrregcount = 41
- Register req. forced down
- Register spills
- Occupancy goes further up but doesn't hide latency
- Doesn't pay off

Single, large kernel still faster…
Challenges and limitations

- In explicit FE granularity is intuitive and parallelism trivial but limited
  - Element kernels
  - Nodal kernels

- Therefore lots of memory per element
  - Especially for Lagrangian methods
  - Even more so on higher order elements and elasticity problems.

- High register req. -> occupancy issues
  - Low occupancy -> difficult latency hiding

- High register req. -> register spilling
  - Additional latency, can be hidden only by ILP

- Splitting the kernel unlikely to improve performance
Simple compression problem on truth cube

Boundary conditions:
- Top node set goes down Z, rest is free
- Bottom node constrained in Z, rest is free

Models range from 5x5x5 to 45x45x45

Comparing against Abaqus
- Accuracy and speed

Model: Neo-Hookean solid

GPUs used in testing:
- K20c
- C2075
- GTX680
- GTX580
- GTX460

Testing modes
- Single/Double precision
- Using/not using texture memory
Testing and validation: 90k elements, 50k timesteps

- Single card solution
Testing and validation: per time-step view

But can we use the single precision solution?
Round-off error increases with:
- number of time-steps
- element density
Final comments

- Generally:
  - Decomposition at the element level is necessary
  - Problem driven by memory, bound by latency

- Practically:
  - Prefer more, lower order elements
  - Lower occupancy may provide better performance
  - Splitting a large force computation kernel unlikely to increase performance
  - Use atomics on Kepler, no need for mini-reduction of force contributions
Use left/right mouse button to rotate/zoom.
Use numbers 1-7 to switch/merge models.
Use 'C' to switch stresses.
Use 'A' to switch autonomous compression/tension.
Use 'T' to switch to free mode.
Use '+' / '-' to compress/extend in free mode.
Use 'L' to lock/unlock the top and bottom rows.
Use 'F' to switch between nonframe/fill rendering.
Use 'M' for colormap.
Use 'R' for colormap values.
Use 'S' for status.
Use 'H' for Help. Turn off TEXT for best performance.
Thank you.

Questions?
computeForces()

- Create 2 additional versions of the kernel, memory only and math only.
- Tricking the compiler can be tricky: mem-only version must contain “some” compute, and math-only must waste cycles on generating data instead of fetching.
- Poor overlap between mem and math.

CONCLUSION:
Memory and latency bound.