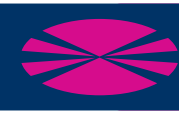
Optimization of Multibody Real-Time Simulator

Alberto Luaces Fernández

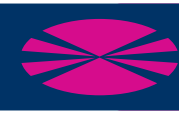
August 8th, 2013, Madison, Wisconsin



Multibody simulator with more than 70 bodies and generic contact model:

- Matrix size 702×702.
- High sparsity rate, around 2%.
- Stored in *coordinate* sparse format (i, j, val).
- If considered a banded matrix, $kl = 17$, $ku = 27$.
- Time to beat is 0.6 milliseconds (on Euler).

COO format



The *COO* format is not the most efficient, but has some nice properties:

- Straightforward to interpret and parse.
- There is no fixed sparse structure.
- Allows to add unordered elements at any time.
- Several matrices can be concatenated, and then selected with the beginning and end of the vectors.

The final form of the problem is

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda}^{*} + \mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\mathbf{\Phi} = \mathbf{Q}(\mathbf{q}, \dot{\mathbf{q}}) \tag{1}$$

$$\boldsymbol{\lambda}_{i+1}^{*} = \boldsymbol{\lambda}_{i}^{*} + \alpha\mathbf{\Phi}_{i+1},\ i = 0, 1, 2, ... \tag{2}$$

$$\left[\frac{\partial f(\mathbf{q})}{\partial \mathbf{q}}\right]_{i} \Delta\mathbf{q}_{i+1} = -[f(\mathbf{q})]_{i},\ \boldsymbol{\lambda}_{i+1}^{*} = \boldsymbol{\lambda}_{i}^{*} + \alpha\mathbf{\Phi}_{i+1} \tag{3}$$

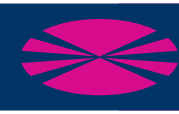
$$\left[\frac{\partial f(\mathbf{q})}{\partial \mathbf{q}}\right] \cong (1-\delta_m)\,\mathbf{M} + (1-\delta_f)\,\gamma h\mathbf{C}_{n+1} + (1-\delta_f)\,\beta h^2 \left(\mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\mathbf{\Phi}_{\mathbf{q}} + \mathbf{K}\right)_{n+1} \tag{4}$$

The tangent matrix is composed of $\mathbf{M}$, $\mathbf{C}$, $\mathbf{K}$ and $\mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\mathbf{\Phi}_{\mathbf{q}}$

- The choosen formulation results in a constant $\mathbf{M}$ during all the simulation.
- Any element of the tangent matrix can be the result of the addition of the different matrices.
- At any iteration or time step, $\mathbf{C}$, $\mathbf{K}$ and $\mathbf{\Phi}_{\mathbf{q}}^{\mathrm{T}}\alpha\mathbf{\Phi}_{\mathbf{q}}$ can be updated individually or deactivated by setting some of its elements to zero.

Problem: in order to use the final matrix for solving or multiplying, all the elements must be read and assembled in their final position.

Solve as a dense system



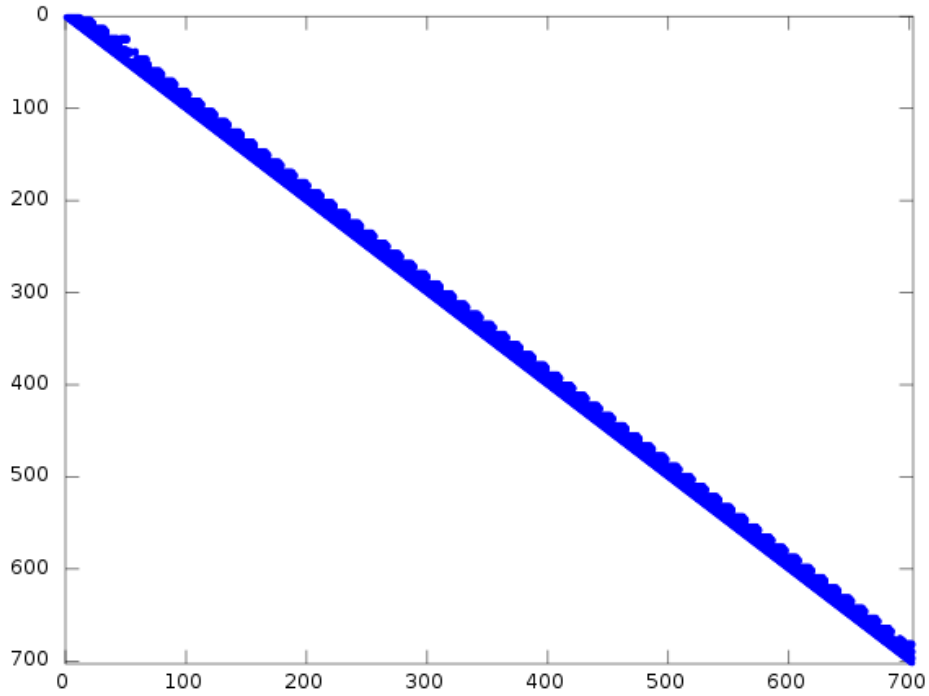
Raw computing power of the GPU can be used in order to solve the problem: decompress the matrix into a dense representation.

- Two approaches regarding bandwitdh: decompress and transmit or transmit and decompress.
- Transfer bandwidth not so important (no noticeable difference between host and device function).
- Using LAPACK parallelized library, CULA.
- Selected algorithm DGESV from LAPACK.
- Solving time is 24 milliseconds for host memory.
- Solving time is 19 milliseconds for host memory.
- Reference LAPACK, not MKL: 45 milliseconds.

```
// Host
   culaDgesv(n, 1, &A[0], n, &pivots[0], &b[0], n);

// Device
  culaDeviceDgesv(n, 1, (culaDeviceDouble *) Ad, n,
    (culaDeviceInt *) pivotsd, (culaDeviceDouble *) bd, n);

// LAPACK
  dgesv_(&n, &n_lhs, &A[0], &n, &pivots[0], &b[0], &n, &info);
```

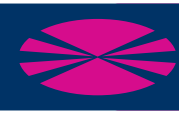
Solve as a banded system

Exploit the banded form of the matrix.



$$\begin{pmatrix} * & * & * & * & \cdots & * \\ * & * & * & * & \cdots & * \\ * & u_{1,2} & u_{2,3} & u_{3,4} & \cdots & u_{n,m-1} \\ d_{1,1} & d_{2,2} & d_{3,3} & d_{4,4} & \cdots & d_{n,m} \\ l_{2,1} & l_{3,2} & l_{4,3} & l_{5,4} & \cdots & * \\ l_{3,1} & l_{4,2} & l_{5,3} & l_{6,4} & \cdots & * \end{pmatrix}$$

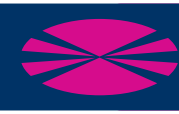
Solve as a banded system



- The complexity grows with the bandwith of the matrix, not with the matrix size.
- Banded matrix format has to be filled from *COO*
- Matrix factorization with DGBTRF in CULA v17.
- There is no (CUDA) accelerated (yet) counterpart for solving $\mathbf{LUx} = \mathbf{b}$
- Resorting to the CPU solution solver, DGBTRS.
- Spends 1.8 milliseconds (factorization itself, 1.4 ms).
- The banded matrix is still very sparse.

```
// Factorize

culaDeviceDgbtrf(rows, cols, kl, ku, (culaDeviceDouble *) bandd,
   stride, (culaDeviceInt *) pivotsd);

// Copy memory back to CPU...

(...)

// Solve (LAPACK)

dgbtrs_(&type, &cols, &kl, &ku, &nhrs, &band[0], &stride,
    &pivots[0], &b[0], &rows, &error);
```

Sparse solver (cuSparse) I

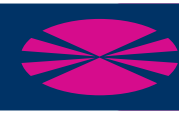


Resorting to sparse iterative solver.

- Strict ordering; partial support for *COO*, everything else *CSR*.
- *COO* format is also strict: ordered-by-row, non-repeated elements.
- The $13,571$ elements (keys) are reduced to $8,124$ elements.
- *Thrust*'s `sort_by_key` and `reduce_by_key` are helpful here:

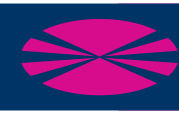
```
// Sort by rows, then by column

  thrust::sort_by_key(
        thrust::make_zip_iterator(thrust::make_tuple(j.begin(), i.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(j.end(), i.end())),
        v.begin());

// Add elements with same indices

 auto end = thrust::reduce_by_key(
        thrust::make_zip_iterator(thrust::make_tuple(j.begin(), i.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(j.end(), i.end())),
        v.begin(),
        thrust::make_zip_iterator(thrust::make_tuple(oj.begin(), oi.begin())),
        ov.begin());
```

Sparse solver (cuSparse) II



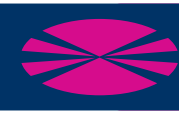
The solver has two steps:

- Analyze structure: `cusparseDcsrsm_analysis`.
- Actually solve system: `cusparseDcsrsv_solve`.
- Fast analysis task, however it has to be carried always (depends on val).
- Best time for the system is 1.80 milliseconds (solve time $50\mu s$!).



Summary of the results:

Solver	DP time (ms)	SP time (ms)
CULA Dense	19	18
LAPACK Dense	45	-
CULA banded	1.8	1.31
cuSparse CSR	1.8	2.15



Conclusions:

- Data transfers seem appropriate for real-time purposes ($< 10\mu s$) even on cheap hardware.
- It is not possible to solve the problem just by brute force (transfer and computation limitations).
- Banded scheme could be a solution, but it is not fully implemented,
- Sparse solvers behave better than expected; real bottleneck is data ordering.

Possible improvements:

- Try not to transfer redundant data (only the neccesary to build the matrix on the GPU side).
- Use single precision. Computation speedup, works on cheaper hardware. Precision problems.
- Study the timing of even smaller problems, to see how they scale.