Non-linear system solving using Thrust and cuSOLVER

Mario Rohrhofer

January 28, 2016

Introduction

Implementation

Problems

Benchmarks

Dynamic simulation with IPSEpro

- Within the ECO-COOL-Project the TU Graz together with partners in industry develops dynamic models of household refrigerators. The goal is to achieve highly detailed models and find possibilities to decrease the total energy consumption.

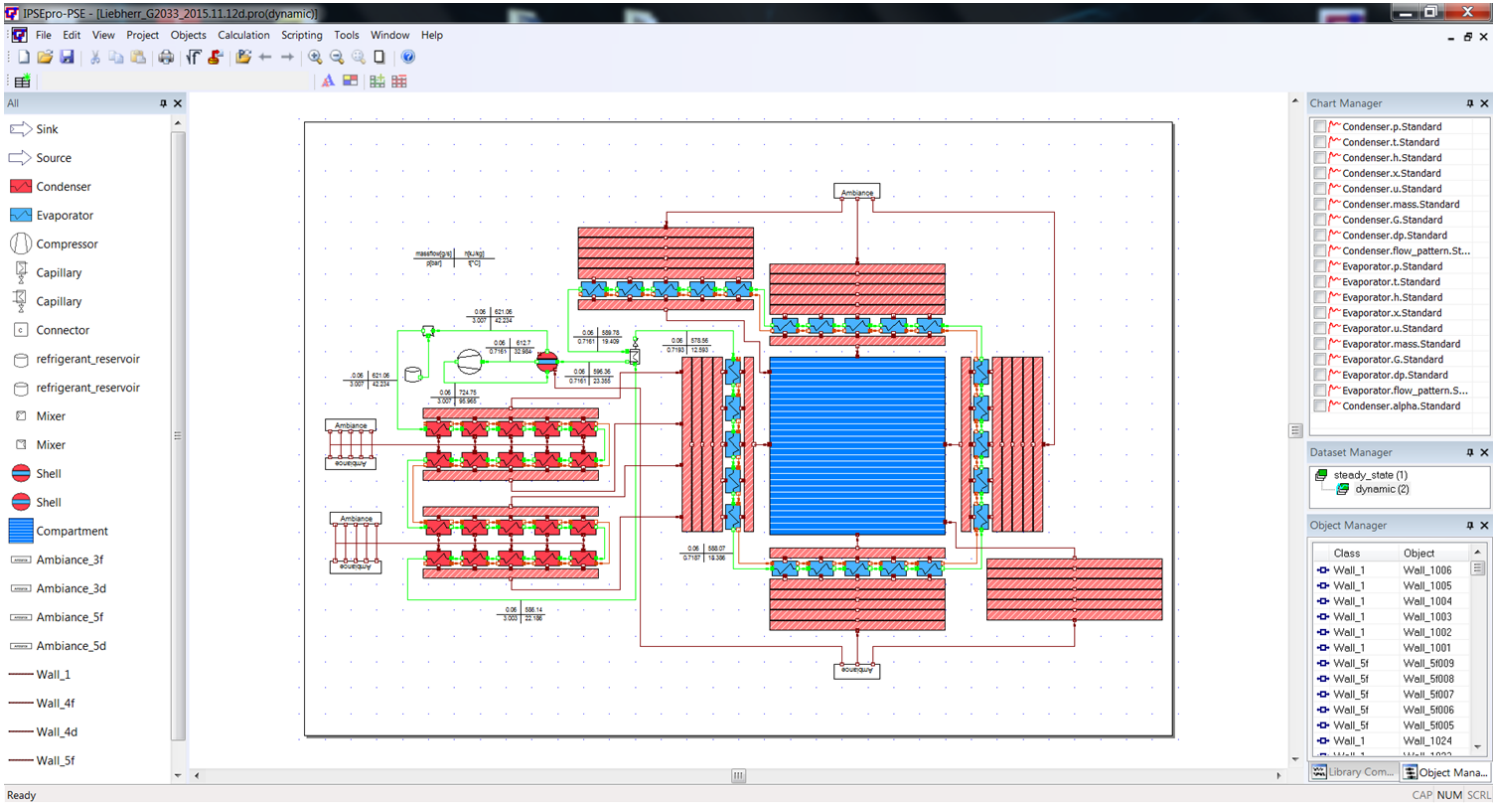

Figure: Flowsheet of a houshold refrigerator.

Dynamic simulation with IPSEpro

- The software IPSEpro by SimTech GmbH is used to solve the arising differential-algebraic systems.
- We consider the initial value problem given by

$$\begin{aligned} F(t, x, \dot{x}) &= 0 \\ x(t_0) &= x_0, \end{aligned}$$

  with $F : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ and $x_0 \in \mathbb{R}^n$.
- The system is solved by the implicit BDF-method. Hence a system of nonlinear equations has to be solved in each time step which is done via Newton's method.
- The Jacobian has the special form

$$J(t, x, \dot{x}) = \frac{1}{\Delta t} \frac{\partial F}{\partial \dot{x}}(t, x, \dot{x}) + \frac{\partial F}{\partial x}(t, x, \dot{x}).$$

- The goal is to solve $J(t, x, \dot{x})\Delta x = -F(t, x, \dot{x})$ on the GPU.

Thrust

- Thrust is a C++ template library similar to the STL.
- Arrays on host and device memory are available with similar functionality as std::vector, *e.g.*.

```
1  std::vector<double> sVec;
2  thrust::host_vector <double> hVec;
3  thrust::device_vector<double> dVec, dVec1;
```

- Thrust also provides algorithms for copying memory from host to device, sorting, reductions and transformations (which are similar to the for_each functionality in the STL). Given a host and device vector copying of memory can simply be done by the assignment operator.

```
1  dVec = hVec;
2  thrust::transform(dVec.begin(), dVec.end(), dVec1.begin(), thrust::negate<double>());
3  thrust::copy(dVec1.begin(), dVec1.end(), sVec.begin()); // -> sVec = -hVec
```

- On the device raw pointers can be extracted from the thrust-vectors.

```
1  double* raw_ptr = thrust::raw_pointer_cast(dVec.data());
```

cuSOLVER

- The cuSOLVER library is based on cuBLAS and cuSPARSE and is available since cuda 7.0.
- The library is (not only) capable of solving linear systems on the CPU and GPU, for dense and sparse matrices.
- Before we can use *e.g.* the QR decomposition several structures have to be allocated.

```
1  cusolverSpCreate(cusolverSpHandle_t *handle);
2  cudaStreamCreate(cudaStream_t *streamId);
3  cusolverSpSetStream(cusolverSpHandle_t handle, cudaStream_t streamId);
4  cusparseCreateMatDescr(cusparseMatDescr_t *descrA);
```

  These variables have to be deleted with respective functions after the calculation.
- The matrix has to be given in CSR format.
- Solving a linear system with QR decomposition can be written in one line, *i.e.*

```
1  cusolverStatus_t cusolverSpDcsrlsvqr( cusolverSpHandle_t handle, int nSize, int nElements,
2                                        const cusparseMatDescr_t descrA,
3                                        const double* csrVal, const int* csrRowPtr,
4                                        const int* csrColInd, const double* b,
5                                        double tol, int reorder, double* x, int* sing);
```

Preliminaries

- All calculations are implemented in a separate DLL.
- Definitions and includes

```
1  #include <vector>
2  #include <thrust\device_vector.h>
3  #include <thrust\host_vector.h>
4
5  #include <cusolverSp.h>
6
7  // helper classes
8  #include "CuMemory.h"  // device specific memory
9  #include "Parameter.h" // decision variables
10 #include "Triplet.h"   // storage class for (i,j,a_ij)
11
12 typedef std::vector<Triplet>          STLTripletArray;
13 typedef thrust::host_vector  <double> HostVec_d;
14 typedef thrust::host_vector  <int>    HostVec_n;
15 typedef thrust::device_vector<double> DeviceVec_d;
16 typedef thrust::device_vector<int>    DeviceVec_n;
```

- The class *Parameter* contains all decision variables, such as the chosen device or the applied solver.
- The class *cuMemory* contains variables needed for the solver such as cusolverSpHandle_t, cudaStream_t or cusparseMatDescr_t.

The Solver Class

```
1  class CuSolve
2  {
3  public:
4      CuSolve ();
5      ~CuSolve();
6
7      void Initialize    (const int nSize);
8      void SetParameters (const Parameter& Par);
9      void AddTriplet    (const Triplet& Tri);
10     void AddTriplet    (const int i, const int j, const double val);
11     void ResetTriplets ();
12     void SetCSREntry   (const int i, const double val);
13     void SetRhsEntry   (const int i, const double val);
14     int  Solve         (double* const pSolution);
15
16 protected:
17     void GenerateCSR   ();
18     void SolveOnGPU    (double* const pSolution);
19     void SolveOnCPU    (double* const pSolution);
20
21 private:
22     int m_nSize;       // system size
23     int m_nElements;   // # elements in sparse matrix
24     int m_singularity; // indicator of singular row
25
26     STLTripletArray m_Triplets; // sparse matrix A in COO format
27     Parameter m_Parameter;      // parameter for solving
28
29     // host memory
30     // sparse matrix A in CSR format and rhs
31     HostVec_d m_HVals; // array of entries
32     HostVec_n m_HRows; // array of row indices
33     HostVec_n m_HCols; // array of column indices
34     HostVec_d m_HRhs;  // array of right hand side
35 };
```

CSR-Format

```
1  void CuSolve::GenerateCSR()
2  {
3      m_nElements = m_Triplets.size();
4
5      // allocate host memory
6      m_HRows.clear();
7      m_HRows.resize(m_nSize+1,0);
8      m_HCols.resize(m_nElements);
9      m_HVals.resize(m_nElements);
10
11     int nEntry    = 0;
12     int nRowEntry = 0;
13     int nLastRow  = 0;
14     m_HRows[nRowEntry++] = nEntry;
15
16     for (int i = 0; i < m_nElements; i++)
17     {
18         if ( nLastRow < m_Triplets[i].Row() )
19         {
20             m_HRows[nRowEntry++] = nEntry;
21             nLastRow++;
22         }
23
24         m_HVals[nEntry]   = m_Triplets[i].Val();
25         m_HCols[nEntry++] = m_Triplets[i].Col();
26     }
27
28     // add the final position
29     m_HRows[nRowEntry] = nEntry;
30 }
```

Solving the linear System on the GPU

```
1  void CuSolve::SolveOnGPU (double* const pSolution)
2  {
3      CuMemory theCuMem;
4      cusolverStatus_t csStatus;
5
6      // generate the appropriate matrix structure
7      if ( m_Parameter.CSRGen() == GENERATE_CSR )
8          GenerateCSR();
9
10     // copy memory from host to device
11     DeviceVec_d d_Vals = m_HVals; // matrix entries
12     DeviceVec_n d_Rows = m_HRows; // row indices
13     DeviceVec_n d_Cols = m_HCols; // column indices
14     DeviceVec_d d_Rhs  = m_HRhs;  // right hand side
15     DeviceVec_d d_Sol(m_nSize);   // solution
16
17     // get raw pointers
18     double* pd_Vals = thrust::raw_pointer_cast(d_Vals.data());
19     int*    pd_Rows = thrust::raw_pointer_cast(d_Rows.data());
20     int*    pd_Cols = thrust::raw_pointer_cast(d_Cols.data());
21     double* pd_Rhs  = thrust::raw_pointer_cast(d_Rhs.data());
22     double* pd_Sol  = thrust::raw_pointer_cast(d_Sol.data());
23
24     if ( m_Parameter.Solver() == LU )
25     {
26         printf("\nLU is not available on the device. QR is used instead\n");
27         m_Parameter.SetSolver(QR);
28     }
29     ...
```

Solving the linear System on the GPU

```
1   ...
2   // Solve A*x = b
3   if ( m_Parameter.Solver() == CHOLESKY )
4   {
5       printf("\nCholesky is used \n");
6       csStatus = cusolverSpDcsrlsvchol(*theCuMem.SolverHandle(), m_nSize, m_nElements,
7                                        *theCuMem.SpMatDescr(), pd_Vals, pd_Rows, pd_Cols,
8                                        pd_Rhs, m_Parameter.Tol(), m_Parameter.Reorder(),
9                                        pd_Sol, &m_singularity);
10  }
11  else // if ( m_Parameter.Solver() == QR )
12  {
13      printf("\nQR is used \n");
14      csStatus = cusolverSpDcsrlsvqr(*theCuMem.SolverHandle(), m_nSize, m_nElements,
15                                     *theCuMem.SpMatDescr(), pd_Vals, pd_Rows, pd_Cols,
16                                     pd_Rhs, m_Parameter.Tol(), m_Parameter.Reorder(),
17                                     pd_Sol, &m_singularity);
18  }
19
20  if ( 0 <= m_singularity )
21  {
22      printf("WARNING: the matrix is singular at row %d under tol (%E)\n", m_singularity,
            m_Parameter.Tol());
23  }
24  else
25  {
26      // copy device memory to solution vector
27      thrust::copy(d_Sol.begin(), d_Sol.end(), pSolution);
28  }
29 }
```

Compatibility

- Several problems of compatibility arose during the implementation process.
  1. The source code of the company could only be compiled with 32bit and is written in Visual Studio 2010.
  2. Since Cuda 7.x, libraries such as cuSPARSE, cuBLAS, cuFFT or cuSOLVER are only available in 64bit.
  3. Cuda 6.5 has 32bit libraries but cuSOLVER is not yet implemented.
  4. With MPI and the MPI_Comm_spawn-function we tried to enable communication between 32 and 64bit processes but the function is not implemented in MSMPI v6. A board entry said in HP-MPI the function is available but Windows blocks it.
  5. The final attempt was made with Intel Parallel Studio XE 2016 which contains Intel MPI and the Intel compiler. Unfortunately, all MPI implementations are only available in 64bit since Intel MPI Library 5.0.

Low scale test data

- With Matlab's sprand-function 3 regular sparse matrices of size $n = 100, 500, 1000$ were generated.
- One sparse system of a real model with size $n = 724$ is used.
- To obtain larger systems the matrices are repeated along the diagonal, *i.e.* via the Kronecker tensor product

$$I \otimes A = \begin{pmatrix} A & & \\ & \ddots & \\ & & A \end{pmatrix}.$$

- A vector filled with ones is used as right hand side.
- All tests were performed on a HP ZBook with Intel Core i7-4700MQ CPU @2.4GHz, 4+4 cores, 8 GB Ram and Quadro K1100M (GPU).

Results

- CPU and GPU Tests were performed with the CuSolverSp Library using QR decomposition. Each calculation is repeated 30 times and the average times are shown in the table below. The GPU times include the copying of memory.

Size	Rep	GPU [s]	CPU [s]
100	50	0.228267	0.089033
100	100	0.4063	0.152567
100	500	1.877333	0.633167
100	1000	$*$	1.233667
500	2	0.228633	0.2377
500	5	0.5241	0.5732
500	10	1.100133	1.111533
500	20	2.207133	2.2143
1000	1	0.580433	0.5814
1000	2	1.181567	1.21
1000	3	1.8177	1.925667
724	1	0.059767	0.032667

$*$ a Thrust-error occurred in the destructor of the device memory.