Finite-Element Method completely implemented for graphic processor units using parallel algorithm libraries

Franz Pichler[*]; Gundolf Haase[†]

July 3, 2015

Abstract

A finite element code is developed in which all computational expensive steps are performed on a graphics processing unit (GPU) via the THRUST and the PARALUTION library. The code is focused on simulation of transient problems where the repeated computations per time step create the computational cost. It is applied to solve partial and ordinary differential equations as they arise in thermal-runaway simulations of automotive batteries. The speedup obtained by utilizing the GPU for every critical step is compared against the single core and the multi-threading solution which is also supported by the chosen libraries. This way a high total speedup on the GPU is achieved without the need for programming a single classical Compute Unified Device Architecture (CUDA) kernel.

[*]Virtual Vehicle Research Center, Graz, Austria

[†]Institut für wissenschaftliches Rechnen, Karl-Franzens University of Graz, Austria

1 Introduction

The simulation of thermal abuse scenarios of automotive battery systems is an important task during concept and design phase in electric vehicles. Experimental data for such models is obtained by thermal runaway experiments[8, 9, 7]. This data is used to parametrize thermal-chemical material models of battery chemistries. These material models can then be used in large scale simulations of complete battery systems. Such simulations need a high spatial resolution because of the complex structure of battery modules. These computational grids call for high performance computing implementations.

Furthermore, in transient problems the cost of simulation arises from repetition of evaluation of non-linearities, assembly of matrices, transfer of these matrices to a proper format, solution of the linear system and iteration of these steps for many time steps.

The finite element method is a good candidate for the solution of such problems because of its capability to handle unstructured and complex geometries and its high level of parallelism. Therefore this method has been implemented on many parallel computing target architectures [5, 1, 12]. There have been several papers published on the implementation of the finite element method on GPUs. In [23, 4] very sophisticated analysis of kernel design and other architecture related challenges are given. In [4] the focus of the community on the linear solver step is emphasized. Opposite to this focus the data assembly phase, which can form a performance bottle neck too, is taken into the focus of [4] and related papers. The presented work has the same focus but the approach is different in that programming libraries are used, that reduce the need for expert knowledge. This allows for faster prototyping of high performance applications. Most of the GPU programming is done via the THRUST[11] library that has a similar application programming interface (API) to the C++ standard template library (STL) [17]. So not a single classic CUDA[15] kernel had to be programmed for this application. The data layout needs some expertise in order to allow the THRUST algorithms for data coalescing which speeds up the computations dramatically. Furthermore, the complete THRUST library can easily be used for other hardware backends as OpenMP or Threading Building Blocks (TBB). This allows for a high portability of the whole program.

The library PARALUTION[16] is used for the solution of the arising linear systems. This program also supports a simple API and the portability to OpenMP backend.

2 Generic model formulation

In this section the model equations covered in this work and the used solution methods are explained. For better understanding of such partial equations intensive study of e.g. [6] is recommended.

Partial differential equation The partial differential equations solved in this work are of the form

$$
\begin{aligned}
c(u,x)\dot{u} - \nabla \cdot (k(u,x)\nabla u) &= f(u,x), \qquad (x,t) \in \Omega \times [0,T) && (1)\\
u(x,0) &= u_0 && (2)\\
u|_{\Gamma_D} &= g_D(u,x,t) && (3)\\
k(u,x)\nabla u|_{\Gamma_N} &= g_N(u,x,t) && (4)\\
k(u,x)\nabla u|_{\Gamma_R} &= \alpha(g_R(u,x,t) - u) && (5)
\end{aligned}
$$

where u is the unknown, and the coefficients $c \in \mathbb{R}$ and $k \in \mathbb{R}^{3\times 3}$ are assumed to be positive. Physically equation (1) can be interpreted as a diffusion equation on the domain $\Omega \subset \mathbb{R}^3$ where u is a diffusing species, c is the volumetric capacity, k is the tensor conductivity and f is a volumetric source term.

Equations (3) - (5) form three standard boundary conditions for such PDEs. The Dirichlet condition (3) fixes the unknown at boundary Γ_D. and will be realized here via the penalty form[2]

$$k(u,x)\nabla u|_{\Gamma_R} = \frac{1}{\varepsilon}(\mathbf{g}_D(u,x,t) - u) \qquad (3')$$

The Neumann condition (4), fixes the physical flow at the part Γ_N of the boundary. And the last boundary condition (5) referred to as a Robin boundary, relates the flow at the boundary to the unknown itself and a given function at the boundary Γ_R.

With equations of this form physical transport phenomena for various quantities as for example heat and chemical species can be modeled. For the ease of presentation the dependencies on the u, x and t are notated by bold letters from here on e.g.

$$\mathbf{k} := k(u,x,t).$$

A weak formulation[6] of the system (1) - (5) is achieved by multiplying equation (1) with test functions $\phi_j \in W^{1,0}(\Omega)$ and applying the divergence theorem to achieve

$$\int\limits_{\Omega} (\mathbf{c}\dot{u}\phi + \mathbf{k}\nabla u \cdot \nabla\phi)\, dx = \int\limits_{\Omega} \mathbf{f}\phi dx + \int\limits_{\partial\Omega} \mathbf{k}\nabla u \phi dS(x) \qquad (6)$$

where $W^{1,0}(\Omega)$ denotes the standard Sobolev space for this application. At this point boundary conditions (3'),(4) and (5) can be put in the surface integral on the right hand side of (6) by

$$
\begin{aligned}
\int\limits_{\partial\Omega} \mathbf{k}\nabla u\phi dS(x) &= \int\limits_{\Gamma_D} \mathbf{k}\nabla u\phi dS(x) + \int\limits_{\Gamma_N} \mathbf{k}\nabla u\phi dS(x) + \int\limits_{\Gamma_R} \mathbf{k}\nabla u\phi dS(x) = \\
&\int\limits_{\Gamma_D} \frac{1}{\varepsilon}(\mathbf{g}_D - u)\,\phi dS(x) + \int\limits_{\Gamma_N} \mathbf{g}_N\phi dS(x) + \int\limits_{\Gamma_R} \alpha(\mathbf{g}_R - u)\phi dS(x)
\end{aligned}
$$

which yields the final weak formulation

$$\int\limits_{\Omega} \left(\mathbf{c}\dot{u}\phi + \mathbf{k}\nabla u \cdot \nabla\phi\right) dx + \int\limits_{\Gamma_D} \frac{1}{\varepsilon} u\phi dS(x) + \int\limits_{\Gamma_R} \alpha u\phi dS(x) =$$

$$\int\limits_{\Omega} \mathbf{f}\phi dx + \int\limits_{\Gamma_N} \mathbf{g}_N\phi dS(x) + \int\limits_{\Gamma_D} \frac{1}{\varepsilon}\mathbf{g}_D\phi dS(x) + \int\limits_{\Gamma_R} \alpha\mathbf{g}_R\phi dS(x) \quad (7)$$

Finite element method The finite element method is a standard to solve (7) in a numerical approximative way. LITERATURE TO FEM. The basic idea here is to approximate the unknown u by linear combinations of a set of base functions $\phi \in W^{1,0}(\Omega)$ which have finite support regions combined from finite elements. The set of all finite elements and their nodes form the computational grid or mesh. The approximation is done by linear combinations of these base functions as

$$u \approx u^h = \sum_{i=1}^{n_p} u_i\phi_i \quad (8)$$

where n_p is the number of degrees of freedom that a specific finite element formulation has. For simplicity, linear tetrahedrons are assumed in the bulk of the domain Ω which yields triangles as boundary elements. Furthermore linear basis functions are assumed, for which the number of degrees of freedom n_p is just the number of mesh nodes. All the presented ideas are easily adapted to higher order elements.

Putting approximation (8) into (7) and choosing the test functions to be the same as the base functions yields

$$\sum_{i=1}^{n_p} \int\limits_{\Omega} \mathbf{c}\phi_i\phi_j dx\dot{u}_i +$$

$$\sum_{i=1}^{n_p} \left(\int\limits_{\Omega} \mathbf{k}\nabla\phi_i \cdot \nabla\phi_j dx + \int\limits_{\Gamma_D} \frac{1}{\varepsilon}\phi_i\phi_j dS(x) + \int\limits_{\Gamma_R} \alpha\phi_i\phi_j dS(x) \right) u_i =$$

$$\int\limits_{\Omega} \mathbf{f}\phi_j dx + \int\limits_{\Gamma_N} \mathbf{g}_N\phi_j dS(x) + \int\limits_{\Gamma_D} \frac{1}{\varepsilon}\mathbf{g}_D\phi dS(x) + \int\limits_{\Gamma_R} \alpha\mathbf{g}_R\phi_j dS(x) \quad (9)$$

for $j = 1, \ldots, n_p$.

Altogether (9) describes a system of (possibly non-linear) ordinary differential equations (ODES) in the unknowns $u_i^h, i = 1, \ldots, n_p$. For simplicity these ODES are solved by implicit Euler's method [10], a stable one step time discretization scheme. Here the time derivative is approximated by

$$\dot{u_i}(t_k) \approx \frac{1}{dt_k}\left(u_i(t_k) - u_i(t_{k-1})\right)$$

where t_k are the discretization points in time and dt_k are the time step size between them.

In order to solve the nonlinear problem a Picard iteration scheme is applied in which $u_i(t_k)$ is approximated by iterations $u_i^l(t_k)$ with iteration index l. In every iteration the nonlinearities are evaluated in dependency of the last iteration e.g.

$$\mathbf{f}_k^l := f(\sum_{i=1}^{n_p} u_i^l(t_k)\phi_i(x), x, t_k)$$

Introducing vector notation and matrix notations

$$\begin{aligned}
\bar{u}_k^l &= \{u_i^l(t_k)\}_{i=1}^{n_p}, && \text{(Vector of Unknowns)}\\
\bar{\bar{M}}_k^l &= \{\int\limits_\Omega \mathbf{c}\phi_i\phi_j dx\}_{i,j=1}^{n_p}, && \text{(Mass Matrix)}\\
\bar{\bar{K}}_k^l &= \{\int\limits_\Omega \mathbf{k}\nabla\phi_i \cdot \nabla\phi_j dx\}_{i,j=1}^{n_p}, && \text{(Stiffness Matrix)}\\
\bar{F}_k^l &= \{\int\limits_\Omega \mathbf{f}\phi_j dx\}_{j=1}^{n}, && \text{(Source or Load Vector)}\\
\bar{N}_k^l &= \{\int\limits_{\Gamma_N} \mathbf{g}_N\phi_j dS(x)\}_{j=1}^{n}, && \text{(Neumann Vector)}\\
\bar{\bar{D}}_k^l &= \{\int\limits_\Omega \frac{1}{\varepsilon}\phi_i\phi_j dx\}_{i,j=1}^{n_p}, && \text{(Dirichlet Matrix)}\\
\bar{D}_k^l &= \{\int\limits_\Omega \frac{1}{\varepsilon}\mathbf{g}_D\phi_j dx\}_{j=1}^{n_p}, && \text{(Dirichlet Vector)}\\
\bar{\bar{R}}_k^l &= \{\int\limits_\Omega \alpha\phi_i\phi_j dx\}_{i,j=1}^{n_p}, && \text{(Robin Matrix)}\\
\bar{R}_k^l &= \{\int\limits_\Omega \alpha\mathbf{g}_R\phi_j dx\}_{j=1}^{n_p} && \text{(Robin Vector)}
\end{aligned}$$

the finite element approximation can be rewritten as the linear system

$$\left(\frac{1}{dt}\bar{\bar{M}}_k^l + \bar{\bar{K}}_k^l + \bar{\bar{D}}_k^l + \bar{\bar{R}}_k^l\right)\bar{u}_k^l = \bar{F}_k^l + \bar{N}_k^l + \bar{D}_k^l + \bar{R}_k^l + \frac{1}{dt}\bar{\bar{M}}_k^l u_{k-1} \tag{10}$$

that has to be solved for every time-step k and for every iteration l until some convergence criteria is reached. Here $\bar{u}_{k-1}$ denotes the accepted solution from the last time step.

Ordinary differential equation In addition to the partial differential equations also ordinary differential equations (ODEs) of the form

$$\dot{u}(x,t) = f(u(x,t), x, t)$$

are considered in this work. Here the unknowns can be spread over a spatial domain Ω_{ode} but there is no transport of the quantity u in space. Example quantities for such a set of equations would be chemical reactions for which the involved species are fixed to their position. The solution of this equations is achieved by an implicit Euler method as

$$\dot{u}(x,t) \approx \frac{1}{dt}(u(x,t_k) - u(x,t_{k-1})) = f(u(x,t_k),x,t_k)$$

which yields

$$u(x,t_k) = u(x,t_{k-1}) + dt f(u(x,t_k),x,t_k). \tag{11}$$

The spatial discretization of the ODE unknowns is the same as in the finite element mesh which eases implementation.

Adaptive time-step strategy The implicit Euler method that is used for the discretization in time, is implemented with an adaptive time step method[14]. Here two steps of size $dt * 0.5$ are calculated starting from the last accepted solution. Then one step of size dt is calculated again starting from the last accepted solution. The time-step size is then categorized by the criteria

$$c_{time} = \frac{\|u_{h2} - u_f\|}{0.5(\|u_h\| + \|u_f\|)}$$

Depending on the size of this criteria the time-step is either accepted and enlarged, accepted and kept or denied and calculated again with a smaller size.

Coupled systems of equations With the PDEs and ODEs described above, coupled systems of equations can be used to simulate cross effects between physical quantities. For such cross effects to occur some of the terms arising in the equations have to be dependent on the solution of another equation. For example a coupled system could look like

$$\begin{aligned} \dot{u}_1 - \nabla \cdot \nabla u_1 &= f_1(u_1, u_2, x, t) \\ \dot{u}_2 &= f_2(u_2) \end{aligned}$$

where the term f_1 is showing such a cross dependency. These cross effects will be solved in an explicit manner here which means that for the non-linear iteration process of one equation the unknowns of the other equations are treated as constants, i.e. with

$$f_{\text{nonliniter}}(u_1, x, t) = f_1(u_1, \tilde{u}_2, x, t)$$

where $\tilde{u}_2$ is the last solution of the second equation the equations are solved as

$$\begin{aligned} \dot{u}_1 - \nabla \cdot \nabla u_1 &= f_{\text{nonliniter}}(u_1, x, t), \\ \dot{u}_2 &= f_2(u_2). \end{aligned}$$

When the non-linear iteration of one equation is finished, the next equation is solved where the solution of the first equation is again treated as a constant. This coupled iterations are continued until convergence is reached.

3 Implementation

In this chapter the implementation of the above described methods is summarized. The implementation is done in C++ and CUDA with the help of the CUDA library THRUST.

Finite element method The scheme described in chapter 2 is a standard finite element method that was implemented in many codes and software packages in academic and commercial applications.

Many of these codes would loop over the elements and load the data for one element, calculate the element matrix, which is its contribution to the global matrix, and add this entries to the global matrix. Once the global linear system is assembled it is solved and the whole process is iterated. The data structure is often based on the elements in such codes. All the data for one element form a structure, which is saved element by element in machine memory (array-of-structures).

The principle of assembling, solving and iterating is naturally still applied here but it was found that the data structure should be changed according to the data coalescing rules. Only with these changes to the data structure a speed up could be achieved using the GPU.

The data structure was designed by the structure-of-arrays (SoA) approach in opposite to the above described array-of-structures (AoS) approach [22]. These two approaches are sketched for the data structure used to store a finite element mesh consisting of tetrahedrons, in Figure 1. SoA usually allows for better data coalescing because the threads of the GPU can all access the needed data for the element that they are treating at the moment with one joint memory access.

The implementation presented here has its focus on speed and not memory so there where decisions made where memory usage may easily be optimized by the cost of additional computations.

Problem Initialization The first step in solving the model equations is to load the mesh and all user functions. The mesh is constraint to linear tetrahedra in this work but can easily be adapted to more sophisticated elements. Its bulk elements are saved as sketched in Figure 1. The whole mesh can be grouped into different sub-domains in order to distinguish physically different parts of the model domain. Every part is then an object in memory saved by the SoA approach. Every part has its own set of material properties that are described by a set of user functions describing the functions c, k and f in (1). The same idea is applied to the boundaries of the domain. The whole boundary is split into different boundary parts of triangles that each have a set of user functions associated with, describing the functions $\mathbf{g}_N, \mathbf{g}_D$ or $\mathbf{g}_R$ and α in (3)-(5). For all elements the transformation matrices from a reference element and their determinant are calculated and also stored in a SoA approach.

Knowing which terms are applied in the model domain allows to pre-calculate

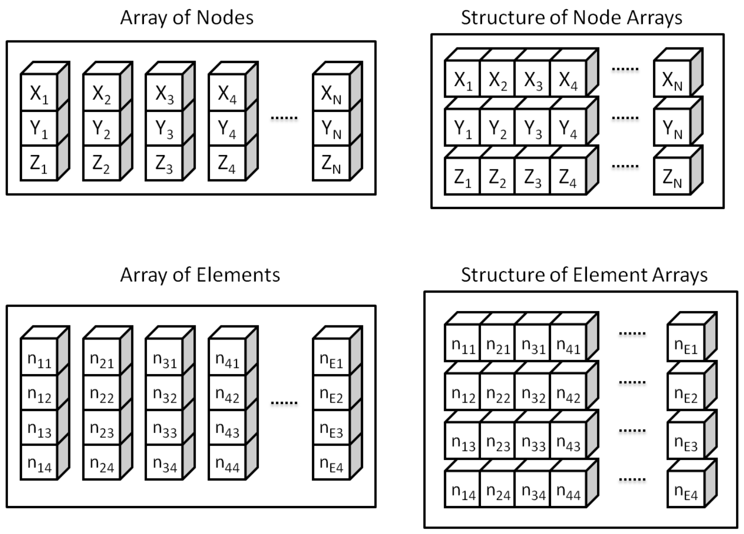

Figure 1: The idea of AoS (left) and SoA (right) by the example of saving a three dimensional mesh of tetrahedra. The xi, yi and zi values denote the coordinates of the mesh's i-th node. The nij integers denote the j-th node of the i-th element in the mesh.

the structure of the linear system (10). The linear system will be calculated in two steps that are actually implemented continuously as described below. First coordinate matrices (COO) will be established that describe the element matrices of every element in a SoA approach. Afterwards this matrices are transformed to the system matrix of (10) in compressed row storage format (CRS), which is one of the standard formats for sparse linear systems [3], or to the vector representing the right hand side of (10).

The structure of the COO matrices is described by a set of row and column indices for every element matrix entry. These row and column indices are assembled once in the initial phase of the simulation. Parallel to these indices the entries of the matrix can be assembled in the same structure, as shown in figure 2. The structures of rows and columns indices are each stored as one long vector. From them a permutation vector P can be calculated that gives the index in the CRS matrix in which an entry in the element matrix belongs to. This permutation vector is actually stored on the GPU and is used later on for the assembly of the CRS matrix. The vector has duplicate entries by the nature of the finite element method, meaning multiple element entries can belong to the same CRS entry.

In addition to the permutation vector the CRS displacement and the column index vector of the CRS format are stored on the GPU. These are uniquely defined and calculated from the row and column index vectors of the COO format. Finally a GPU vector storing the CRS elements has to be allocated. The COO entries do not have to be stored because they are only calculated locally and then immediately added to the CRS entries.

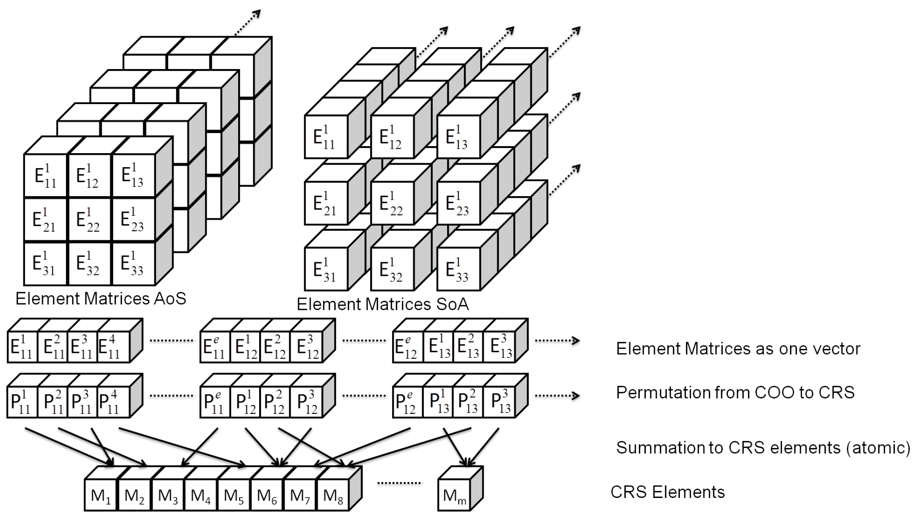
Element Matrices AoS
Element Matrices SoA
Element Matrices as one vector
Permutation from COO to CRS
Summation to CRS elements (atomic)
CRS Elements

Figure 2: The element matrices are shown for triangle elments in the Aos and SoA approach (top). Here the upper index denotes the finite element from which the element matrix E is generated. The lower indices ij denote i-th local row and the j-th local column. Below the COO entries assembled as one vector and the permutation vector (P) are shown. This permutation gives the global index in the CRS matrix (M) to which the corresponding entry in the COO matrix (E) has to be added. This summation has to be performed by atomic operations in parallel because of the duplicate entries in the P vector.

Evaluate nonlinearities In the beginning of the iteration the functions $\mathbf{c}$, $\mathbf{k}$, $\mathbf{f}$, $\mathbf{g}_D$, $\mathbf{g}_R$, $\mathbf{g}_N$ and α are evaluated for every node of the mesh. The resulting values are then stored in device vectors which permanently use working memory. In all further iterations only non constant functions are evaluated at the beginning of every iteration. Depending on the nature of the functions the computational effort can only be estimated knowing them. There are several versions of this call implemented depending on the nature of the user function.

If the function is simply a constant than `thrust::fill` is used to evaluate. If the function is depending on time, coordinates or the solution than a functor with the right amount of arguments is called in `thrust::for_each` that is loading a header file filled with the user specific operators. This allows the user to change the user functions by minimal compilation effort.

As an example the conductivity $\mathbf{k}$ is assumed to be nonlinear, depending on the solution u. This dependency is calculated and stored to a vector $\bar{k}$ where the n-th value is the value at the n-th node of the mesh

$$\bar{k}_n = k(\bar{u}_n)$$

Assemble COO Matrix The matrix assembly is done by a call to `thrust::for_each` where the element matrix structure of the virtual COO matrix is generated locally in an assembling functor. The output argument for this functor are the zipped element matrices, that are immediately permuted by the permutation vector P, on which the entries are added with atomic operations. That way every thread creates the element matrix of one element and then adds them directly into the CRS matrix which renders the storage of the COO entries unnecessary. This allows a single call to `thrust::for_each` to assemble the contribution of an equation term (10)) to the CRS matrix at once for all elements of a sub-domain. The input arguments which are the element transformation matrices and the determinant of these as well as the user function values and the Gauss weights are zipped separately in the same manner. A more detailed explanation of the numerical integration is given in appendix A.

Finalize Equation As a final step the mass matrix, that is also assembled separately, by the methods described above, has to be multiplied by the old solution and divided by the factor dt and finally be added to the right hand side of the problem. This can be done by the linear algebra functionality of most linear solvers or also trivially implemented with THRUST, which was done in this work.

Linear Solution For the solution of the linear system (10) the package PARALUTION [16] has been used. It has a free single GPU node version that gives some choices of preconditioners and solvers. The implemented Jacobi preconditioner and the conjugate gradient method are applied to solve the symmetric linear system of equations in every iteration. It is not the aim of this work to apply an optimal solver and so the choice of the preconditioner and solver have rather been based on simplicity in use and robustness than optimal speed or memory usage.

Ordinary differential equations The solution of the ODEs as described in 2 is straight forward to implement, and only a summary of it will be given here. The function evaluation is mechanism is the same asd for the PDEs and the solution of (11) is realized by element-wise division of one vector by another and an element-wise addition to the solution of the last time-step, i.e.

$$\bar{u}_k^l = dt\bar{F}_k^l/\bar{M}_k^l + u_{k-1} \tag{12}$$

where / denotes the element-wise division and the two vectors $\bar{F}_k^l$ and $\bar{M}_k^l$ denote the l-th iteration in the $k-th$ time step of the process. The entries of the vectors are the nodal evaluations of the ODE terms, i.e.

$$F_i = f(u_i, x_i, t_k), \tag{13}$$
$$M_i = c(u_i, x_i, t_k). \tag{14}$$

Calculate Convergence Norm In order to compute the norm of the solution and the norm of the change of the solution the following relation is used

$$\int (u^h)^2 dx = \int (\sum_{i=1}^{n_p} \phi_i u_i)^2 = \int \sum_{i=1}^{n_p} \sum_{j=1}^{n_p} \phi_i u_i \phi_j u_j dx = \bar{u}^T \bar{\bar{N}} \bar{u} \qquad (15)$$

where $\bar{\bar{N}}$ is a matrix that has similar form to the mass matrix $\bar{\bar{M}}$ and is one assembled and transformed to CRS format in the beginning of the simulation. This way calculation of the norm is done by a vector-matrix-vector product which again can be realized by the linear algebra capabilities of most GPU solvers or implemented directly.

4 Example Application

The presented finite element framework is applied to the simulation of thermal runaway experiments [9, 8, 7] of automotive batteries. The model describes the relation of the temperature distribution in a battery module (shown in Figure 3 and an exothermic reaction. This scenario is of great importance in safety related fields in the automotive and other sectors.

This model will be used to characterize different chemical materials and scenarios in the field of automotive batteries. For such a characterization many experiments in the lab and numerical simulations are necessary. Therefore a fast implementation of such a model is of high importance.

The thermal runaway process There are many candidates for the exothermic reactions that can occur in a battery [13] and cause thermal runaway. In the presented work the exothermic reaction is generically described by a progress quantity α[13] describing the amount of reactant that is still present. The battery will be heated up from the outside as a possibly abuse scenario. Once a certain heat is reached in the jelly roll the reaction will start and the battery will heat up even faster due to the exothermic nature of the process. The progress quantity will monotonically decrease until it reaches zero which indicates that all potential reactants of the reaction are used and the reaction will stop once again which is equivalent to a completely burnt out battery.

The equation domain A generic battery module was constructed, consisting of different parts as indicated in Figure 3 where most of the parts are just passive in the sense that they only act as heat conductors in the model. The only active parts in the module are the jelly rolls, which are the parts of a battery that contain the electro chemical active materials used to transform electrical into chemical energy and vice versa. This part consists of thin wounded layers.The layers form the cell sandwich (metal/cathode/separator/anode/metal) in which exothermic reactions can occur if the battery is mistreated. These exothermic reactions can lead to a thermal runaway process in which the battery is destroyed in the end and can cause serious damage to its surroundings.

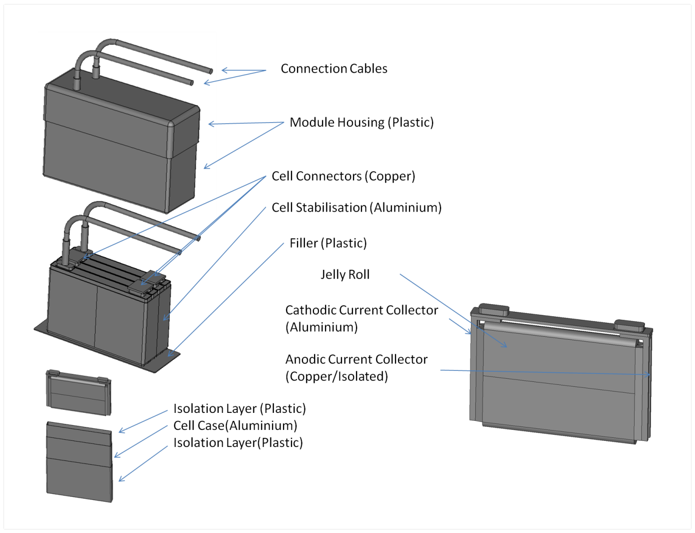
Connection Cables
Module Housing (Plastic)
Cell Connectors (Copper)
Cell Stabilisation (Aluminium)
Filler (Plastic)
Jelly Roll
Cathodic Current Collector (Aluminium)
Anodic Current Collector (Copper/Isolated)
Isolation Layer (Plastic)
Cell Case(Aluminium)
Isolation Layer(Plastic)

Figure 3: The geometry ofthe battery module: a) shows the whole battery module that is housed in a plastic casing outof which the two main contacts are connected with copper cables. b) shows the module without the case where the 6 cells are visible surroundeed by an aluminium cage and a bottom plastic plate. c) Shows the a single cell. In the left side the cell is exploded into its three surrounding layers that are a inner plastic isolation, a aluminium case and an outer plastic isolation. The right side shows the cell in more detail where the connection from the active jelly roll to the outside can be seen (The voids in the cell and the module case are another subdomain not shown that is forming the air in the module)

The chemical active parts are denoted by Ω_c and the whole domain is denoted by Ω.

Furthermore the same governing equations will be solved on a simple cube of edge length 100mm for which different mesh sizes are compared. With this cubes a relation of mesh size to speed up is established.

The computational mesh The geometrical discretization or mesh was created using the meshing tool SALOME [20] and a CAD geometry drawn in HYPREWORKS REFto hypreworks. In addition to the parts shown in Figure 3 the empty spaces inside the module and the cells have been meshed and represent the air inside the module. The cube was completely generated by SALOME. Four different mesh sizes where used where the cube edges where split into 10,20,40 and 80 segments. The resulting number of nodes and elements for all the meshes are summarized in table 4.

Mesh	Edge Element size	Nodes	Elements
Cube A	10 mm	2220	9166
Cube B	5 mm	12394	56401
Cube C	2.5 mm	109327	567476
Cube D	1.25 mm	543936	2920774
Battery Module	3 mm	132697	745950

Table 1: The meshes and their sizes used for simulation. The edge element length in the battery module is 3 mm wherever possible and smaller where necessary.

The governing equations The temperature distribution is governed by the heat equation

$$c_p(x)\dot{T}(x,t) - \nabla \cdot (\lambda(x)\nabla T(x,t)) = \Delta H \dot{\alpha}(x,t), \tag{16}$$

$$(x,t) \in \Omega \times [0, t_{\text{end}}], \tag{17}$$

where T is the temperature, c_p is the volumetric heat capacity, λ is the thermal conductivity and ΔH is the heat enthalpy of the exothermic reaction f. The chemical reaction progress is modeled by the ODE

$$\dot{\alpha}(x,t) = -f(\alpha(x,t), T(x,T)) \quad (x,t) \in \Omega_c \times [0, t_{\text{end}}] \tag{18}$$

where Ω_c denotes the sub-domain in the battery that describes the chemical active part of the battery module. The exothermic reaction is of Arrhenius type

$$f(c,t) = kc\exp(\frac{E_a}{RT}) \tag{19}$$

where k is the reaction frequency, E_a is the activation energy of the reaction and R is the universal gas constant.

Remark that in these equations the source terms are the nonlinearities that have to be assembled in every iteration of the solution process. In order to analyze also the extreme case, where every occurring term would be nonlinear and would have to be assembled for every iteration, a second set of numerical simulations is performed in which the terms are all assembled even though they are constant.

The initial conditions In the beginning the temperature is set to 420 degree Kelvin, which is just below the temperature where the thermal runaway is started, and the reaction progress is set to 1, which implies that all reactants are still existing.

The boundary condition The ambient temperature is raised linearly from 420 degree Kelvin, by 2 degree Kelvin per minute, for 1000 seconds and is then held constant. An estimated heat transfer coefficient of $\alpha = 10\frac{W}{Km^2}$ is used in the Robin boundary condition.

The simulation is executed for 2400 seconds of simulation time, after which all reactions have been finished and the temperature has reached a steady state.

The convergence criteria The applied tolerances for the simulation scenario discussed above are summarized in table 4. The criteria for the reaction progress solved for in the ODE (18) is taken relative to the its domain measure, because the range of the unknown goes down to 0, which forms a numerical problem when taking a relative criteria. The three tolerances for the adaptive time-stepping algorithm give the limits for either accepting the time-step and enlarging it for the next step, accepting it and keeping the same size, or not accepting it an decreasing it for the next try.

	Linear Solver	Nonlinear Iteration	Coupled Iteration	Adaptive Timestep
Temperature	$\|Ax - b\|_L^2$	$\frac{\|T_i - T_{i-1}\|}{0.5(\|T_i\| + \|T_{i-1}\|)}$	$\frac{\|T_j - T_{j-1}\|}{0.5(\|T_j\| + \|T_{j-1}\|)}$	$\frac{\|T_{2h} - T_f\|}{0.5(\|T_{2h}\| + \|T_f\|)}$
Reaction	not applicable	$\frac{\|\alpha_i - \alpha_{i-1}\|}{\|\Omega\|}$	$\frac{\|\alpha_j - \alpha_{j-1}\|}{\|\Omega\|}$	$\frac{[0.5(\|\alpha_{2h}\| + \|\alpha_f\|)]}{\|\Omega\|}$
tol	1.e-7	1.e-6	1.e-6	1.e-3,1.e-4,1.e-5

5 Results

In order to compare the simulation speed of the GPU architecture to a CPU architecture the OMP backends of THRUST and PARALUTION were used. So the whole program can be switched from GPU to CPU architecture by changing the according compiler flags. The only point where the code is different for these two architectures are the atomic additions that have to be implemented via the OMP pragma `atomic` for the OMP backend and via CUDAs `atomicAdd` for the GPU backend.

The calculations have been performed on a NVIDIA Titan Black and on 1 to 6 threads of a Intel(R) Xeon(R) CPU E5-1620 v2 with 3.70GHz. It was not the aim of this publication to optimize the OMP implementation but still the portability should be underlined. A further run has been performed on the GPU where the three dimensional mesh and the solutions for every time-step have been written to the hard drive using the C++ VTK API [21]. This is in general not necessary in scientific numerical simulations, but sometimes it gives a more intuitive understanding of "what is going on". The result is shown in Figure 6 where the toolbox PARAVIEW [24] has been used for visualization. It should be remarked that writing the whole solution with the mesh for every time-step is easily the slowest step in such a simulation, especially when this involves data transfer from the GPU-RAM to the CPU-RAM.

Cube Simulations The number of time-steps and iterations produced for the examples are summarized in table 5. These numbers are given for the GPU implementation and slightly vary for the OMP implementations and the different number of threads due to rounding differences. This variation is less than one percent for all the listed iteration numbers and therefore not shown here. The

Mesh	Timesteps	PDE It.	ODE It.	Linear It.
Cube A(single)	204/8	4509	5422	84298
Cube A(double)	204/8	4472	5399	49072
Cube B(single)	203/8	4460	5397	133800
Cube B(double)	205/8	4466	5374	78269
Cube C(single)	205/8	4452	5373	238086
Cube C(double)	204/8	4472	5400	121589
Cube D(single)	202/8	4330	5347	397624
Cube D(double)	204/8	4472	5400	214816
Battery Module(single)	137/144	2783	3414	1035449
Battery Module(double)	137/144	2779	3356	602278

Table 2: Summary of the time-steps and iterations needed to solve the example applications. The second number in the time-step column gives the denied number of time-steps. Every timesteps consists of the calculation of 3 actual timesteps (half, second half and full step) as described by the time-stepping algorithm.

number of time-steps is the same for all implementations. The most interesting fact shown in this table is that the number of linear iterations needed in the single precision mode is often nearly twice as big as for the double precision case. This is due to the tolerances that are the same for both implementations. By the nature of the less precise numbers, they need more iterations to achieve the same tolerance, than the more precise numbers. This explains why in total the double precision implementation is sometimes faster than the single precision one.

The two most expensive steps, namely assembling the CRS matrix and the solution of the arising linear systems, and their total execution time are summarized in Figure 5 for the cube domain and in Figure 5 for the battery module. As expected the advantages of the GPU are only showing for problems of a certain size. On the coarsest cube the problem is actually solved faster by the CPU implementations, because the linear solver is slower on the GPU than on the CPU. For a higher number of discretization points the GPU pays off more and more, where for the finest cube a total speed up of XXX is achieved. The presented results make it obvious that usage of the GPU overcomes the use of more and more threads in a multi-threaded environment. The results show that the multi-threading solutions stagnate in their speed-up using more than 3 threads which is probably due to the limited bandwidth they share. Using distributed threads could improve the OMP implementation, because that way a higher memory bandwidth can be achieved. However, up to 3 threads the solution scales very nice. The GPU has a way higher bandwidth than the CPU threads, in theory and as shown in praxis. Especially the assembling step profited here. A speed up of a factor of 33 was reached for the assembly in Cube 3 for single precision. For double precision the pseudo atomic addition on the GPU shows its cost. Where the OMP solution still scale similar to the single precision case , the GPU loses half of it speed up. Still it is nearly ten times faster than the CPU.

For the module simulation the linear solver step takes up way more of the total simulation time than for the cube simulations. This is due to the aspect ratios of the elements that is very bad for thin elements as they arise in the isolation layers in the module. Problems like these arise in moset real life apliucations and that is why it is important to study the simulation times not only on simlpe geometries like the cube. The extreme case where not only the source term but all terms are treated nonlinear is shown in the bottom plots in Figure 5. It shows that both, the linear solver and the assembly, can form the bottle neck of the simulation, depending on the number of nonlinear terms in the PDE. For al cases the OMP solution scales good up to 3 threads as for the cube studies. Also the GPU solution performs very well for all cases even though the highest speed up was established for the single precision case with forced nonlinearities. For this case the CPU is slower for single than for double precision because of the higher number of linear solver iterations needed for convergence. Opposed to that the GPU solution is faster in the single precision case because of the actual atomic operations that it can perform for single precision.

6 Conclusion

Using the THRUST library, a fast implementation of GPU code was possible without the need for kernel programming. This allows a beginner in this field to achieve results quickly. The data design was the only part where some knowledge about the GPU architecture was used in order to allow for data coalescing. A wide range of coupled ordinary and partial differential equations as they occur in battery modeling and in other fields can be handled quite efficiently by the presented solution. By having all the involved steps implemented on the GPU a good basis is created for keeping the cost of all possible bottle necks low. This is also due to the fact that no data transfer from CPU to GPU memory is needed except for single numbers in the whole simulation.

For the cases studied in this work speed ups of a factor up to 30 where achieved in the assembling phase for single precision numbers and speed ups of up to 9 for the double precision case. If there would be real atomic summations available for double precision numbers in the future this result could easily be improved.

Further improvements of the code are possible and planned. This can involve the implementation of further elements and also the implementation of further equation terms in the partial differential equations. Furthermore a Newton-Raphson method can generally be implemented for this system where the derivatives of the occurring nonlinear functions have to be given by the user. Depending on the nature of these further improvements the assembly or the solving step will get more expensive.

7 Acknowledgements

This work was accomplished at the VIRTUAL VEHICLE Research Center in Graz, Austria. The authors would like to acknowledge the financial support of the "COMET - Competence Centers for Excellent Technologies Programme of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Science, Research and Economy (bmwfw), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG).

They would furthermore like to express their thanks to their supporting industrial and scientific project partners namely Samsung SDI Battery Systems GmbH and AVL List.

The Titan Black used for this research was donated by the NVIDIA Corporation.

Appendices

A Numerical Integration

The numerical integration implemented in this work is a gauss quadrature scheme combined with the use of a reference element Ω_{ref} that is approximating the integral over an element Ω_e by

$$\int\limits_{\Omega_e} f(x)dx \approx \sum_{g=1}^{n_g} \omega_g f(T_e(x_g))|det\left(J_{T_e}(x_g)\right)|. \tag{20}$$

This needs a set of weights ω_g, integration points x_g on the reference element and the transformation T_e and its spatial derivative J_{T_e}. The gauss weights and points be taken from literature [18, 19] and the transformation T_e, that maps the reference tetrahedron Ω_{ref} to the element Ω_e, is given by a linear function

$$T_e : \Omega_{ref} -> \Omega_e \tag{21}$$

$$\tilde{x} -> \bar{\bar{T}}_e \tilde{x} + x_0 \tag{22}$$

which yields the Jacobi matrix

$$J_{T_e} = \bar{\bar{T}}_e. \tag{23}$$

The linear operator for a tetrahedron that has the four corner points $\bar{x}_1, \bar{x}_2, \bar{x}_3$ and $\bar{x}_4$ is given by

$$\bar{\bar{T}}_e = \left[\bar{x}_2 - \bar{x}_1 | \quad \bar{x}_4 - \bar{x}_1 | \quad \bar{x}_4 - \bar{x}_1.\right] \tag{24}$$

An analogous matrix is set up for the triangle boundary elements.

In order to keep the number of user function evaluations low the functions at the Gauss points are again approximated by the finite element method. For the example of the stiffness matrix $\bar{\bar{K}}$ involving the term $\mathbf{k}$. This means that the actual evaluation of the integral is given by

$$\int\limits_{\Omega} \nabla\phi_i \cdot k(u')\nabla\phi_j dx = \sum_{g=1}^{n_g} (\bar{\bar{T}}_e \tilde{\nabla}\tilde{\phi}_i(x_g)) \cdot \sum_{k=1}^{n_n} \bar{\bar{k}}_{p(k)} \tilde{\phi}_k(x_g)(\bar{\bar{T}}_e \tilde{\nabla}\phi_j(x_g))\omega_g |det(\bar{\bar{T}}_e)| \tag{25}$$

where $p_e(\cdot)$ denotes a permutation from the local node index k of element e to the global node index $p_e(k)$. This can be rewritten for an element matrix $\bar{\bar{K}}_e$ as

$$\bar{\bar{K}}_e = \sum_{g=1}^{n_g} (\bar{\bar{T}}_e \bar{\bar{G}}_g)^T (\bar{\bar{\bar{k}}}_e \bar{B}_g)(\bar{\bar{T}}_e \bar{\bar{G}}_g)\omega_g |det(\bar{\bar{T}}_e)| \tag{26}$$

where $\bar{\bar{G}}_g$ denotes a matrix which columns are the basis function gradients evaluated on the reference element, i.e

$$\bar{\bar{G}}_g = \begin{bmatrix} \tilde{\nabla}\tilde{\phi}_1(\tilde{x}_g) & \tilde{\nabla}\tilde{\phi}_2(\tilde{x}_g) & \tilde{\nabla}\tilde{\phi}_3(\tilde{x}_g) & \tilde{\nabla}\tilde{\phi}_4(\tilde{x}_g) \end{bmatrix}, \tag{27}$$

$\bar{B}_g$ denotes the basis function values on the reference element evaluated at the gauss point $\tilde{x}_g$, i.e

$$\bar{B}_g = \begin{bmatrix} \tilde{\phi}_1(\tilde{x}_g) & \tilde{\phi}_2(\tilde{x}_g) & \tilde{\phi}_3(\tilde{x}_g) & \tilde{\phi}_4(\tilde{x}_g) \end{bmatrix}, \tag{28}$$

$\bar{\bar{\bar{K}}}$ denotes the evaluations of the tensor function k at the nodes of element e at time t_k, forming a three dimensional tensor, i.e

$$\bar{K}(\cdot,\cdot,i) = \begin{bmatrix} \bar{\bar{k}}(u_{p_e(i)}, x_{p_e(i)}, t_k) \end{bmatrix}, \quad i = 1, \ldots, 4 \tag{29}$$

The Stiffness matrix $\bar{\bar{K}}$ is the most complex example of all the terms listed in 2 because it involves the transformation of the gradient of the basis functions. Analogously to the just described method the form of the other terms can be derived as

$$\bar{\bar{M}}_e = \sum_{g=1}^{n_g} \sum_{k=1}^{n_n} \bar{c}_{p(k)} \tilde{\phi}_k(x_g)(B)^T(B)\omega_g |det(J_T)| \tag{30}$$

$$\bar{F}_e = \sum_{g=1}^{n_g} \sum_{k=1}^{n_n} \bar{c}_{p(k)} \tilde{\phi}_k(x_g)(B)\omega_g |det(J_T)| \tag{31}$$

so on for other terms

References

[1] Finite element applications on a shared-memory multiprocessor: Algorithms and experimental results. *Journal of Computational Physics*, 94(2):352 – 381, 1991.

[2] Ivo Babuška. The finite element method with penalty. *Mathematics of computation*, 27(122):221–228, 1973.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[4] Cris Cecka, Adrian J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5):640–669, 2011.

[5] Sergey Choporov. Parallel computing technologies in the finite element method. *Radio Electronics, Computer Science, Control*, (2), 2013.

[6] L.C. Evans. *Partial Differential Equations.* Graduate studies in mathematics. American Mathematical Society, 1998.

[7] Andrey W Golubkov and David Fuchs. Thermal runaway: Causes and consequences on cell level. In *Automotive Battery Technology*, pages 37–51. Springer, 2014.

[8] Andrey W Golubkov, David Fuchs, Julian Wagner, Helmar Wiltsche, Christoph Stangl, Gisela Fauler, Gernot Voitic, Alexander Thaler, and Viktor Hacker. Thermal runaway behavior of commercial 18650 li-ion batteries. *Environmental Science*, 5(1):5271, 2012.

[9] Andrey W Golubkov, David Fuchs, Julian Wagner, Helmar Wiltsche, Christoph Stangl, Gisela Fauler, Gernot Voitic, Alexander Thaler, and Viktor Hacker. Thermal-runaway experiments on consumer li-ion batteries with metal-oxide and olivin-type cathodes. *RSC Advances*, 4(7):3633–3642, 2014.

[10] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[11] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[12] PK Jimack and N Touheed. Developing parallel finite element software using mpi. *High Performance Computing for Computational Mechanics*, pages 15–38, 2000.

[13] D. D. MacNeil and J. R. Dahn. Test of reaction kinetics using both differential scanning and accelerating rate calorimetries as applied to the reaction of lixcoo2 in non-aqueous electrolyte. *The Journal of Physical Chemistry A*, 105(18):4430–4439, 2001.

[14] Susan E Minkoff and Nicholas M Kridler. A comparison of adaptive time stepping methods for coupled flow and deformation modeling. *Applied mathematical modelling*, 30(9):993–1009, 2006.

[15] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[16] PARALUTION Labs. PARALUTION v1.0.0 , 2015. http://www.paralution.com/.

[17] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[18] H.T. Rathod, K.V. Nagaraja, and B. Venkatesudu. Symmetric gauss legendre quadrature formulas for composite numerical integration over a triangular surface. *Applied Mathematics and Computation*, 188(1):865 – 876, 2007.

[19] H.T. Rathod, B. Venkatesudu, K.V. Nagaraja, and Md. Shafiqul Islam. Gauss legendregauss jacobi quadrature rules over a tetrahedral region. *Applied Mathematics and Computation*, 190(1):186 – 194, 2007.

[20] Andre Ribes and Christian Caremoli. Salome platform component model for numerical simulation. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 553–564. IEEE, 2007.

[21] Will Schroeder. *The visualization toolkit : an object-oriented approach to 3D graphics*. Kitware, Clifton Park, N.Y, 2006.

[22] Robert Strzodka. Abstraction for AoS and SoA layout in C++. In Wenmei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 253–269. Morgan Kaufmann, Waltham, MA, 2011.

[23] Li Tang, X. Sharon Hu, Danny Z. Chen, Michael Niemier, Richard F. Barrett, Simon D. Hammond, and Genie Hsieh. Gpu acceleration of data assembly in finite element methods and its energy implications. *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 0:321–328, 2013.

[24] Ayachit Utkarsh. The paraview guide: A parallel visualization application, 2015.

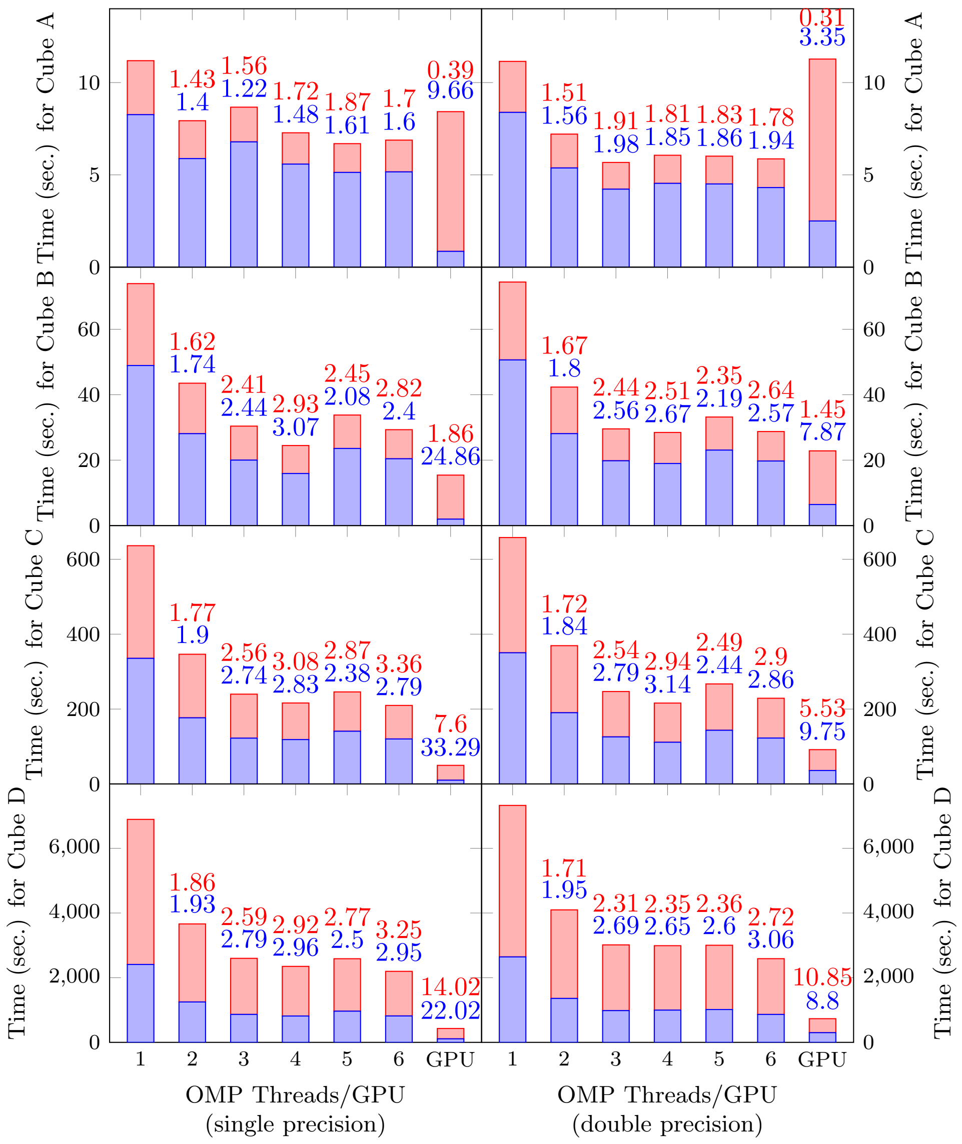

Figure 4: Computation time for the most expensive steps (in seconds) compared for the multi-threaded and the GPU solution. The left and right panels show the timings for single and double precision solution respectively. The lower (blue) bar shows the time for assembling the linear system and the upper (red) bar shows the time for the solution of the system. The speed up compared to the CPU (1 Thread) solution is indicated by the respective number above the bars.

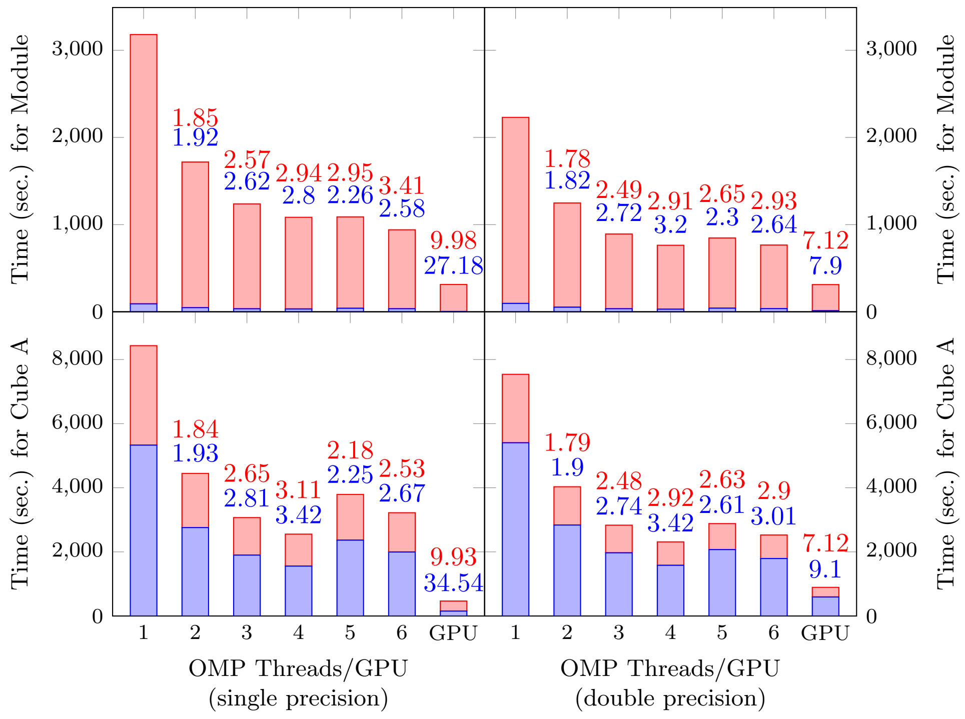

Figure 5: Computation time for the module geometry. Left and right column show single and double precision computation times. Top row shows times where only the nonlinear r.h.s. is assembled in every iteration and bottom row shows times when all terms are treated nonlinear and are assembled in every iteration. The top bar (red) shows time for the linear solver, the lower bar (blue) shows time for assembly and the respective numbers above show seep up against CPU implementation.

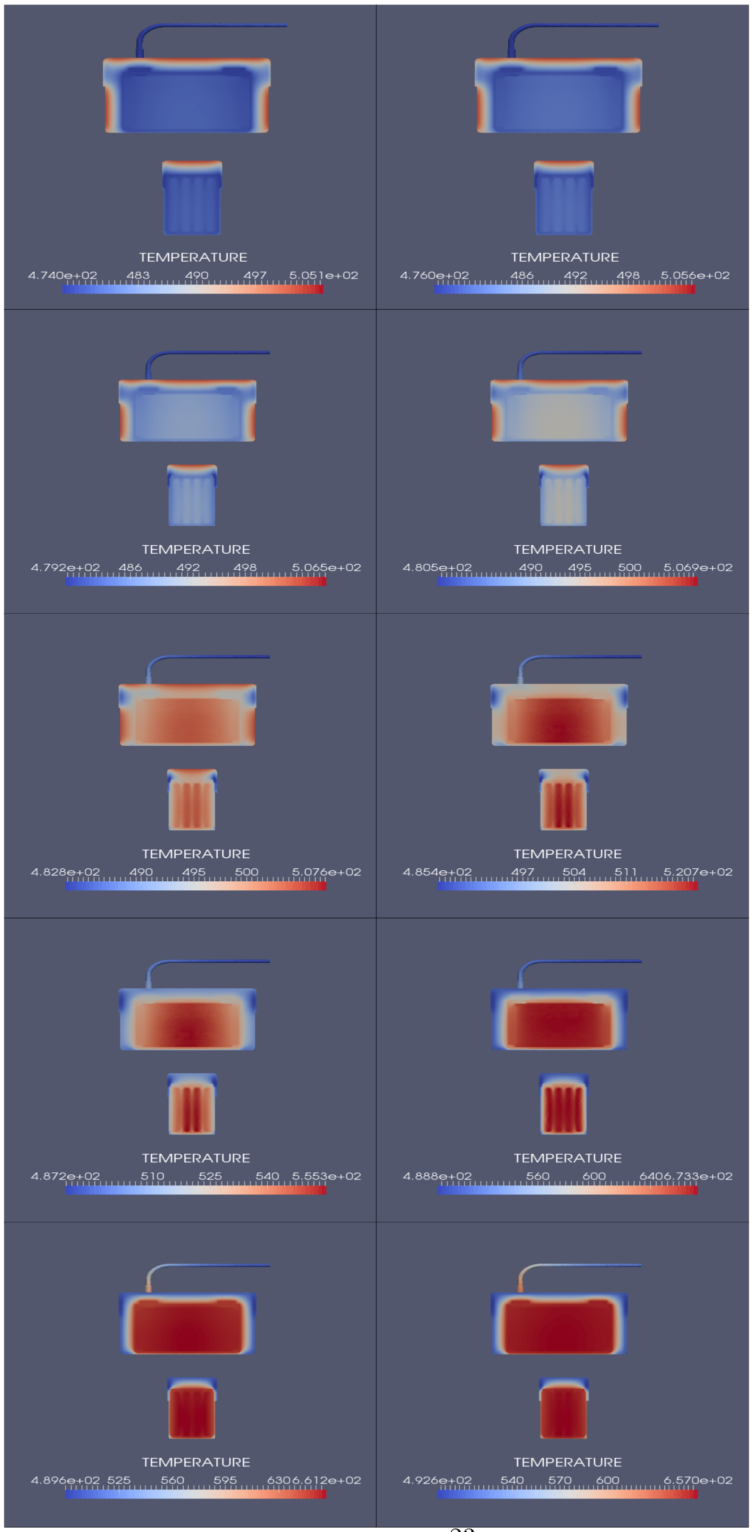

Figure 6: Cut through the module in xz and yz plane for ten time-steps starting at time t=1300 with $\Delta t = 22.sec$