Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dielectric Breakdown Prediction with GPU-Accelerated BEM

Master Thesis

C. Münger

February 13, 2020

Supervisor: Prof. Dr. R. Hiptmair
Department of Mathematics, ETH Zürich

Abstract

This thesis shows the application of the boundary element method to dielectric breakdown prediction and its fast computation using graphics processing unit. First, the problem is formulated in the boundary element context. Then, based on the H2Lib library, the necessary kernels are developed in OpenCL to offload and distribute the computations onto multiple GPUs. The key features of the implementation are illustrated in detail. Last but not least the performance and accuracy of the newly implemented code are evaluated against an existing industrial code.

Acknowledgments

First of all I would like to thank Dr. Jörg Ostrowski at the ABB corporate research center in Dättwil for the opportunity to do this project in industrial environment and his continuous support and valuable input when I was stuck.

I also would like to thank Prof. Dr. Steffen Börm at the university of Kiel, who took a lot of time to discuss and explain the H2Lib Library and priniples of GPUs in person and/or per email.

And I would also like to thank Prof. Dr. Ralf Hiptmair for supervising this thesis.

Also would like to thank my friends from ETH for the last years. The lunch and the afternoon coffees were refreshing breaks form the daily studying. And a special thanks to Yannick and Dominik for proofreading this thesis.

Contents

Contents ii

1 Introduction 1

2 Theoretical Background 3
2.1 Formulation 4
2.1.1 Field lines 6
2.2 GPU Programming 6
2.2.1 CPU vs. GPU hardware 6
2.2.2 OpenCL 7

3 Implementation 9
3.1 Building blocks 9
3.1.1 FE-Space 9
3.1.2 2nd-order triangles 10
3.1.3 Quadrature for near-singular & singular integrals . . . 10
3.2 Matrix Assembly 12
3.3 Constraint Assembly 14
3.4 GMRES 15
3.5 Postprocessing 16
3.6 Distributed-memory parallelization 17

4 Numerical results 20
4.1 Comparison H2Lib vs. POLOPT 20
4.1.1 Test cases 20
4.1.2 Double vs. single precision 22
4.1.3 Time-to-solution comparison 22
4.2 Field line computation 23

5 Conclusion & Outlook 27

A Appendix **28**
A.1 OpenCL kernel 28

B Manual **38**
B.1 Prerequisites 38
B.1.1 Code structure 38
B.1.2 Compile time options 39
B.1.3 Running examples 39

Bibliography **40**

Chapter 1

Introduction

Every high voltage device has to pass dielectric type tests, in which a large voltage is applied to the device. The test is passed if no dielectric breakdown occurs. A breakdown usually starts from an electrode-surface with high dielectric stress, and then propagates through the volume towards the opposing electrode, see figure 1. This propagation stops if the electric field strength along the breakdown path is not strong enough [3].

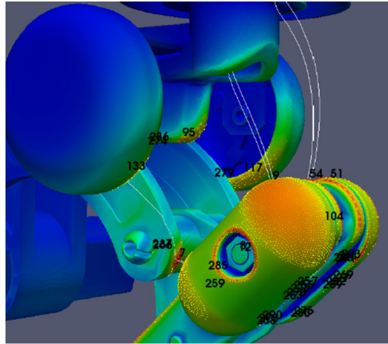


Figure 1.1: The electric field strength on the surface of a disconnector and possible breakdown paths along field lines.

An inception of a streamer, i.e. the initial state of a breakdown only occurs if the criterion (1.1) is fulfilled.

$$\int_{\gamma} \alpha_{eff}(|\mathbf{E}(\gamma(s)|)ds < K_{str} \tag{1.1}$$

Herein α_{eff} is the effective ionization function that depends on the strength of the electric field $\mathbf{E}$, and K_{str} is the (gas specific) streamer constant. Thus the prediction of a dielectric breakdown during a type test requires the com-

putation of the electric field at all surface points and along field lines in the volume.

Simulation-based dielectric design became a standard procedure because a user-friendly, i.e., fast, robust, reliable, and easy-to-use computational method was developed [2], [4]. In the following chapter we will first describe the boundary element method formulation and then introduce how general-purpose graphics processing units (GPGPUs) can be used to reduce computing times. In the third chapter key implementation features will be discussed and the distribution of the workload on to multiple GPUs. Last but not least results will be presented with a focus on application and the comparison to an exiting industrial code.

Chapter 2

Theoretical Background

The necessary equation can be derived directly from Maxwell's equations [2]. In the case of electrostatics all terms with a time derivative vanish. Thus we have

$$\nabla \times \mathbf{E} = 0, \tag{2.1}$$

$$\nabla \mathbf{D} = \rho. \tag{2.2}$$

The electric field is conservative and can be expressed as the gradient of the scalar potential φ

$$\mathbf{E} = -\nabla \varphi. \tag{2.3}$$

Assuming linear materials the constitutive relation has the following form

$$\mathbf{D} = \varepsilon \mathbf{E}, \tag{2.4}$$

where ε is the permittivity of the material. Combining the above equations, we have that the potential φ satisfies the Poisson equation

$$\nabla^2 \varphi = -\frac{\rho}{\varepsilon}. \tag{2.5}$$

In absence of charges in space this is simply the Laplace equation

$$-\nabla^2 \varphi = 0. \tag{2.6}$$

2.1 Formulation

We use an indirect formulation with a single layer potential

$$\varphi(\mathbf{x}) = \Psi_{SL}[\sigma](\mathbf{x}) = \int_{\Gamma} \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}\sigma(\mathbf{y})dS_y. \tag{2.7}$$

Herein, $\mathbf{x}$ is any point in the domain, $\mathbf{y}$ is a point on the surface and σ is the surface charge distribution.

Conductors A conductor, a well separated conducting part, is on a constant potential. If a conductor is connected to a voltage source U_0, then it holds for all points of the conductor surface Γ_C that

$$\varphi(\mathbf{x}) = U_0 \quad \forall \mathbf{x} \in \Gamma_C. \tag{2.8}$$

Floating Conductors For conductors that are not connected to a voltage source it holds that

$$\varphi(\mathbf{x}) = U \quad \forall \mathbf{x} \in \Gamma_F. \tag{2.9}$$

where Γ_F is the surface of the floating conductor and U is an additional unknown that can be determined by a charge neutrality condition [1]. It can be derived from the Gauss law

$$\int_{\Gamma_F} \mathbf{D} \cdot \mathbf{n} dS = Q. \tag{2.10}$$

The normal component of displacement field can be expressed as

$$\mathbf{D} \cdot \mathbf{n} = \varepsilon \mathbf{E} \cdot \mathbf{n} = -\varepsilon \nabla \varphi \cdot \mathbf{n} = -\varepsilon \nabla_n \Psi_{SL}[\sigma]. \tag{2.11}$$

The last term is the exterior Neumann trace of the single layer potential and can be expressed with help of the adjoint double layer

$$\nabla_n \Psi_{SL}[\sigma] = \frac{1}{2}\sigma + K'\sigma. \tag{2.12}$$

Combination of the equations from above yields the constraint

$$\int_{\Gamma_F} \frac{1}{2}\varepsilon\sigma + \varepsilon K'\sigma dS = 0 \tag{2.13}$$

This can be extended if the floating conductor is a thin sheet, i.e. modeled only by a single surface. Then the electric field from the other side needs also to be taken into account

$$\int_{\Gamma_F} \frac{1}{2}(\varepsilon_1 + \varepsilon_2)\sigma + (\varepsilon_1 - \varepsilon_2)K'\sigma dS = 0. \tag{2.14}$$

Dielectric Interfaces On dielectric surfaces the free surface charge σ_S has to vanish everywhere. The interface condition reads as

$$\mathbf{n} \cdot (\mathbf{D}_2 - \mathbf{D}_1) = 0. \tag{2.15}$$

Thus the condition that has to be fulfilled on the dielectric surface Γ_D is

$$\frac{1}{2}(\varepsilon_1 + \varepsilon_2)\sigma(\mathbf{x}) + (\varepsilon_1 - \varepsilon_2)(K'\sigma)(\mathbf{x}) = 0 \quad \text{for } \mathbf{x} \in \Gamma_D. \tag{2.16}$$

General problem In a general setting we have to solve the following set of integral equations.

$$(V\sigma)(\mathbf{x}) = U_0 \text{ for } \mathbf{x} \in \Gamma_C \tag{2.17}$$

$$(V\sigma)(\mathbf{x}) - U = 0 \text{ for } \mathbf{x} \in \Gamma_F \tag{2.18}$$

$$\frac{1}{2}(\varepsilon_1 + \varepsilon_2)\sigma(\mathbf{x}) + (\varepsilon_1 - \varepsilon_2)(K'\sigma)(\mathbf{x}) = 0 \text{ for } \mathbf{x} \in \Gamma_D \tag{2.19}$$

with the additional constraints for the floating potentials either of this form

$$\int_{\Gamma_F} \frac{1}{2}\varepsilon\sigma + \varepsilon K'\sigma dS = 0 \tag{2.20}$$

or in case of a floating sheet in the following form

$$\int_{\Gamma_F} \frac{1}{2}(\varepsilon_1 + \varepsilon_2)\sigma + (\varepsilon_1 - \varepsilon_2)K'\sigma dS = 0 \tag{2.21}$$

where the single layer potentials V is

$$(V\sigma)(\mathbf{x}) = \int_\Gamma \frac{1}{4\pi|\mathbf{x} - \mathbf{y}|}\sigma(\mathbf{y})dS_y \tag{2.22}$$

and the adjoint double layer potential K' is

$$(K'\sigma)(\mathbf{x}) = \int_\Gamma \frac{(\mathbf{x} - \mathbf{y}) \cdot \mathbf{n}(\mathbf{x})}{4\pi|\mathbf{x} - \mathbf{y}|}\sigma(\mathbf{y})dS_y. \tag{2.23}$$

2.1.1 Field lines

To compute field lines we have to solve the following ODE

$$\frac{d\gamma(s)}{ds} = \mathbf{E}(\gamma(s)) \tag{2.24}$$

where $\gamma(s)$ is the field line and $\mathbf{E}$ the electric field.

The electric field is the gradient of the potential (2.3). It can be therefore computed from the gradient of the single layer potential of the surface charge distribution.

$$\mathbf{E}(\mathbf{x}) = -\mathbf{grad}_{\mathbf{x}} \int_{\Gamma} \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}\sigma(\mathbf{y})dS_y = \int_{\Gamma} \frac{(\mathbf{x}-\mathbf{y})}{4\pi\|\mathbf{x}-\mathbf{y}\|^3}\sigma(\mathbf{y})dS_y \tag{2.25}$$

2.2 GPU Programming

In this section we will shortly describe the difference between a CPU and a GPU as well as explain the core concepts of OpenCL.

2.2.1 CPU vs. GPU hardware

Both CPU and GPU comprise of arithmetic logic units (ALU), control logic and a fast local cache and more but slower memory (DRAM). These components differ in size, complexity and number for CPUs and GPUs.

A CPU has a relative small number of ALU, a fairly complex control logic unit and a large cache. Thus is well suited for sequential instruction execution, using techniques like out-of-order execution and speculative execution and low latency of the cache to be fast.

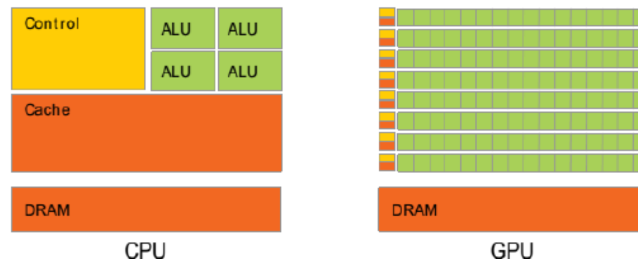


Figure 2.1: Difference of CPU and GPU hardware

A GPU mostly consists of hundreds of ALUs, simpler control logic with a small cache. A GPU is mainly built to do computationally intensive work. Its focus lies on high throughput. It uses different techniques to achieve that. First, it uses massive *vectorization* of the workload. The vector-registers

on a GPU have a typical width of 32. Thus 32 threads execute the same instruction at once. These threads on a GPU are bundled together in so called warps or wavefronts that fill this vector registers. Second, it uses so called *latency hiding*, where multiple warps are ready to be executed on the same compute unit. The scheduler picks one warp to execute. When its execution stalls, maybe waiting for a memory access, another warp, that is ready to be executed on the same compute unit, is executed in the mean time. Thus the latency of the memory access of the first warp is hidden behind the execution of another warp.

The challenge to use the full potential of GPU is to have enough parallelism to fill all instruction-registers, multiple warps per compute unit and to have the full width of the vectorized units do the same instruction.

2.2.2 OpenCL

OpenCL is an open standard for parallel programming of heterogeneous systems. It allows to write programs that can be executed in parallel on heterogeneous systems by providing an high-level abstraction over these systems. There exist in-depth guides like [8]. Here we will only give a short introduction to OpenCL and its core concepts.

The platform model gives a high-level description of the system. An OpenCL platform includes a single host, that interacts with the external environment. The host is connected to one or more OpenCL devices. An OpenCL device can be a CPU, a GPU or any other hardware that is supported by OpenCL. The compute devices are divided into compute units, which are further divided into one or more processing units.

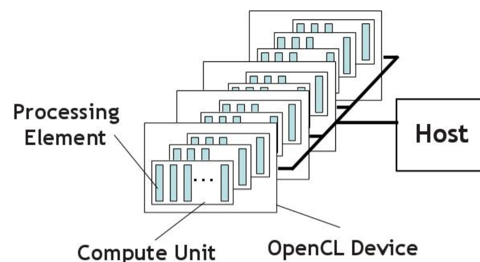


Figure 2.2: OpenCL platform model

The execution model defines how the application is mapped to the hardware. The central part is the kernel, the function we want to execute. When the host issues the command to execute a kernel, the OpenCL runtime creates an index space, the *NDRange*. For each point in the index space an instance of the kernel is executed. This instance is called *workitem*, which is identified by its global coordinates g_i in the index space. The workitems are organized

in equal-sized *workgroups*. Workgroups give a more coarse decomposition of the index space. Workitems in a workgroup are executed concurrently on a single compute unit. The workgroup itself has an index w_i and the workitems in a workgroup have unique local indices l_i. The global index g_i of a workitem can be expressed as combination of its workgroup index w_i, the workgroup size L_i and the local index l_i.

$$g_i = w_i \cdot L_i + l_i \tag{2.26}$$

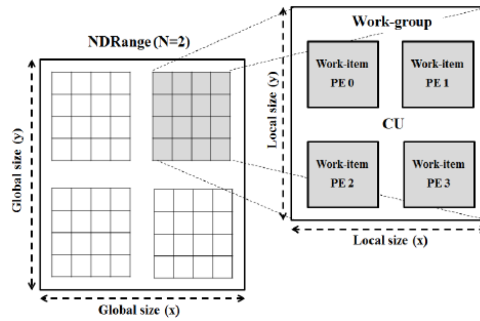


Figure 2.3: Example of 2-dimensional NDRange with global IDs and workgoup with local Ids

There is also a memory model that defines different levels of memory in OpenCL. That is not of importance for this work and will not be discussed here.

The programming model is equivalent to the execution model on the algorithmic level. Meaning it maps a parallel algorithm to OpenCL. There are two different programming models. The data-parallel programming model that uses a data-centered approach. It applies a sequence of instructions concurrently to the data. Sometimes this is also referred to as Single Instruction Multiple Thread (SIMT), similar to single instruction vectorization (SIMD). The other programming model is the task-parallel programming model, where the problem is split into independent tasks. These tasks are them submitted at the same time and it is up to the system to execute them in a balanced order. In this work we only use the data-parallel programming model.

More detailed explanations can be found in [8] and [7].

Chapter 3

Implementation

The implementation is based on the H2Lib Library [6]. H2Lib is an open source library for hierarchical matrices. The library also contains application modules for boundary element methods in 3D.

We use the collection method, that is we demand that the equations (2.17) to (2.19) are only exactly satisfied in the nodes of the mesh $\mathbf{x}_i$.

$$(V\sigma)(\mathbf{x}_i) = U_0 \quad \forall \mathbf{x}_i \in \Gamma_E \tag{3.1}$$

$$(V\sigma)(\mathbf{x}_i) - U = 0 \quad \forall \mathbf{x}_i \in \Gamma_F \tag{3.2}$$

$$\frac{1}{2}(\varepsilon_1 + \varepsilon_2)\sigma(\mathbf{x}_i) + (\varepsilon_1 - \varepsilon_2)(K'\sigma)(\mathbf{x}_i) = 0 \quad \forall \mathbf{x}_i \in \Gamma_D \tag{3.3}$$

3.1 Building blocks

In this section we describe the basic building blocks we need to assembly the collocation matrix.

3.1.1 FE-Space

We use a piece-wise linear approximation for the surface charge density σ. Thus, we will have contributions from multiple elements to one entry in the matrix. Or the other way around a triangle is part of the support of three basis functions. We need to keep track of the mapping between the local basis function on a triangle to the global basis function. In the H2Lib following element-centric representation is used.

We have a 2d array with the dimensions $N \times 4$ where for each of the N elements we store four entries. The first entry gives the index of the physical element in the mesh. Whereas the remaining three are the indices of the basis function associated with the first, second and third corner of the triangle.

The reserved value '∼0' (all bits set) is used to mark when an element has no contribution to a global basis function.

Furthermore, this format allows for efficient storage and transfer onto a GPU, because the memory handling is easier for continuous blocks of memory.

3.1.2 2nd-order triangles

We use the following parametrization for the curved triangles. We have the linear shape functions

$$\varphi_A(t,s) = 1 - s \quad \varphi_B(t,s) = s - t) \quad \varphi_C(t,s) = t$$

and add the edge bubble functions

$$\begin{aligned}
\varphi_{AB}(t,s) &= \varphi_A(t,s)\varphi_B(t,s) = (1-s)(s-t) \\
\varphi_{BC}(t,s) &= \varphi_B(t,s)\varphi_C(t,s) = (s-t)t \\
\varphi_{CA}(t,s) &= \varphi_C(t,s)\varphi_A(t,s) = t(1-s).
\end{aligned}$$

Since the edge bubble function are equal to $1/4$ in the midpoints of the respective edge, we can define the difference vectors

$$D_{AB} := 4M_{AB} - 2(A+B) \quad D_{BC} := 4M_{BC} - 2(B+C) \quad D_{CA} := 4M_{CA} - 2(C+A)$$

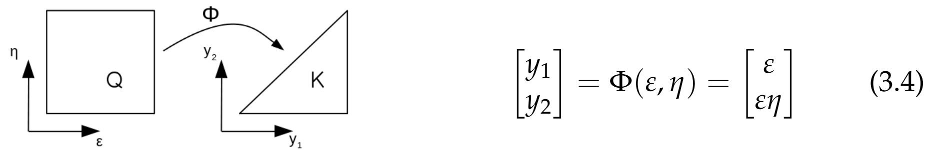
to obtain the quadratic parametrization

$$\begin{aligned}
\Phi(t,s) =& A\varphi_A(t,s) + B\varphi_B(t,s) + C\varphi_C(t,s) + \\
& D_{AB}\varphi_{AB}(t,s) + D_{BC}\varphi_{BC}(t,s) + D_{CA}\varphi_{CA}(t,s)
\end{aligned}$$

where A, B and C are the corner of the triangle and M_{AB}, M_{BC} and M_{CA} are the midpoints of the edges.

3.1.3 Quadrature for near-singular & singular integrals

The integrals for the single layer (2.22) and double layer potential (2.23) are singular if $\mathbf{x}$ is equal to $\mathbf{y}$. This means that standard quadrature does not well approximate the integral. In the case of our collocation method the integral can become singular in a corner of the triangle, i.e. the collocation point is the corner of the triangle. Thus we can use the Duffy-Transformation. It maps the unit square to the reference triangle, see figure 3.1 and equation (3.4).



$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \Phi(\varepsilon,\eta) = \begin{bmatrix} \varepsilon \\ \varepsilon\eta \end{bmatrix} \tag{3.4}$$

Figure 3.1: Mapping from square to triangle.

Now let us assume that the singular corner is $x = [0,0]^T$, then we can transform the integral in the following way

$$\int_K \frac{f(y_1, y_2)}{4\pi\sqrt{y_1^2 + y_2^2}} dy_1 dy_2 = \int_Q \frac{f(\varepsilon, \varepsilon\eta)}{4\pi\sqrt{\varepsilon^2 + (\varepsilon\eta)^2}} \varepsilon d\varepsilon d\eta = \int_Q \frac{f(\varepsilon, \varepsilon\eta)}{4\pi\sqrt{1+\eta^2}} d\varepsilon d\eta. \tag{3.5}$$

Due to the determinant of the transformation's Jacobian the singularity is canceled.

We can use standard tensor-product Gauss-quadrature on the square in combination with the Duffy-Transformation to integrate the arising type of singular integrals in the collocation method.

Near-singular integrals We also have to take care of integrals that are nearly singular, meaning $\mathbf{x}$ and $\mathbf{y}$ are close together. This is the case if there is a narrow gap in the geometry or during the postprocessing when we evaluate a point near the surface. In this case we use a simple subdividing strategy. We compute the closest point $\mathbf{P}$ on the linear approximation of the curved triangle to the point $\mathbf{x}$. And efficient algorithm for this can be found in [5]. We subdivide the triangle in the closest point and again employ the Duffy transformation to integrate over all smaller triangles increasing the resolution of the quadrature locally around the closest point. There a three different cases how we have to split the triangle depending on the location of the closest point , figure 3.2. If $\mathbf{P}$ is one of the corners we do not subdivide. If $\mathbf{P}$ lies on an edge we split the triangle along the line between $\mathbf{P}$ and the opposite corner. If $\mathbf{P}$ is in the interior of the triangle we create three new triangles: PAB, PBC, PCA.

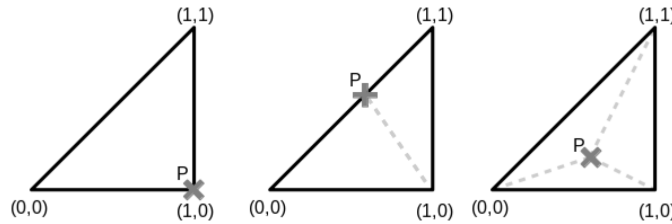


Figure 3.2: Examples of triangle subdivision depending on the location of $\mathbf{P}$

We use a simple heuristic threshold to detect these near singular cases.

$$nearsingular(T, \mathbf{x}) = \frac{|\mathbf{x} - \mathbf{o}|}{r} \tag{3.6}$$

Herein, $\mathbf{x}$ is the collocation point, $\mathbf{o}$ the circumcenter of the triangle T and r its circumcircle radius. If the value of *nearsingular* is smaller than a threshold ε, then we treat the integral as near-singular and use above subdivision method.

3.2 Matrix Assembly

First, we notice that we can assemble the matrix row-wise. We have from the collocation method that the equations hold for each collocation point individually. Each row in the collocation matrix is associated with one collocation point. There is no dependence between individual rows.

Second, due to the linear basis function we have that multiple triangles will contribute to a single entry in the matrix. Or equally, each triangle will contribute to three entries in a row when we assemble the matrix.

The standard method to assemble the matrix is to iterate over each collocation point - triangle pair and compute its local contribution and add it to the right entries of the matrix. For such a collocation point-triangle pair we need to evaluate the integral

$$\int_K \frac{1}{4\pi|\mathbf{x}_i - \mathbf{y}|} b_N^j(\mathbf{y}) dS_y, \tag{3.7}$$

if the collocation point is on a conductor surface or

$$\int_K \frac{(\mathbf{x}_i - \mathbf{y}) \cdot \mathbf{n}(\mathbf{x}_i)}{4\pi|\mathbf{x}_i - \mathbf{y}|} b_N^j(\mathbf{y}) dS_y, \tag{3.8}$$

if the collocation point is on a dielectric surface. Herein, $\mathbf{x}_i$ is the collocation point, K the triangle and b_N^j the j-th basis function.

We suggest the following method to parallelize this process: The assembly is done with one thread per row. In each thread we iterate over all triangle sequentially, where we compute the contribution of the current triangle and update the corresponding entries in the matrix. This approach has the advantage that we don't have to worry about race conditions when updating the matrix entries because each thread writes to a different row.

The realization with OpenCL is straightforward: We use an one-dimensional NDRange. The size of the NDRange matches the number of collocation points. Thus the global index of the thread gives us the index of the collocation point, i.e. the row in the matrix. Then all workitems iterate in lockstep over all triangles. This minimizes the memory reads of the geometry as all workitems work with the same triangle at the same time.

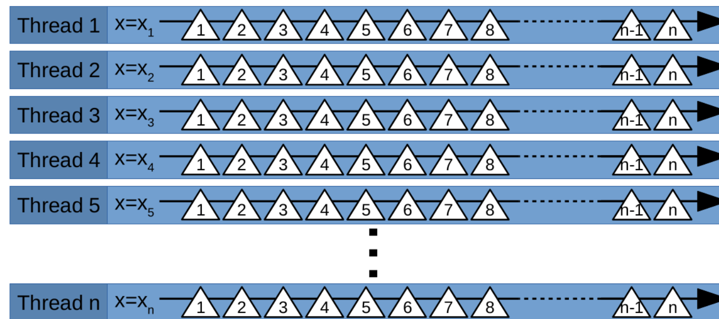


Figure 3.3: Visualization of the parallization of the assembly

Near singular treatment In subsection 3.1.3 is described how to handle singular and near singular integrals. It is an adaptive method for the quadrature. But adaptability can impact the performance of a GPU badly, because we have divergence in the control flow on the GPU and the threads in a warp will be executed sequentially, see section 2.2.1.

Hence we try to minimize the different cases. We only distinguish between two cases. The first is if we have a regular or singular integral. They can be integrated using the standard Duffy-Transformation. By rotating the mapping of the reference triangle to the right corner we can cancel the singularity in any corner of the triangle.

And the other case is if we have a near-singular integral. This case requires the subdivision method. Therefore, to avoid any large control flow divergence, we will first compute the regular and singular pairs and deal with the near-singular integrals later. We iterate over all triangles and the regular and singular pairs are computed but not the near singular pairs. We can identify them with help of (3.6). They will be marked as incomplete. After all triangles have been processed, there are some triangles left that are near singular and thus not yet computed, see figure3.4.

The detailed treatment of the near singular case is described in the previous section. In short, the idea is to split the triangle in multiple triangles so that the closest point is moved into a corner of the smaller triangles and we can again use the Duffy-Transformation. There are three types splits. Either the closest point is already a corner and no splitting is necessary or the closest point lies on an edge then two triangles are needed or it lies in the interior for which three triangles are needed.

But actually we only need one case. The first two are a degenerate case of the third, in which either one or two of the small triangles are degenerated to a line. Hence all near-singular triangles are split into three smaller triangles. If one of the triangles is degenerate, then the corresponding quadrature weights are set to zero by the Gram determinant of the transformation which

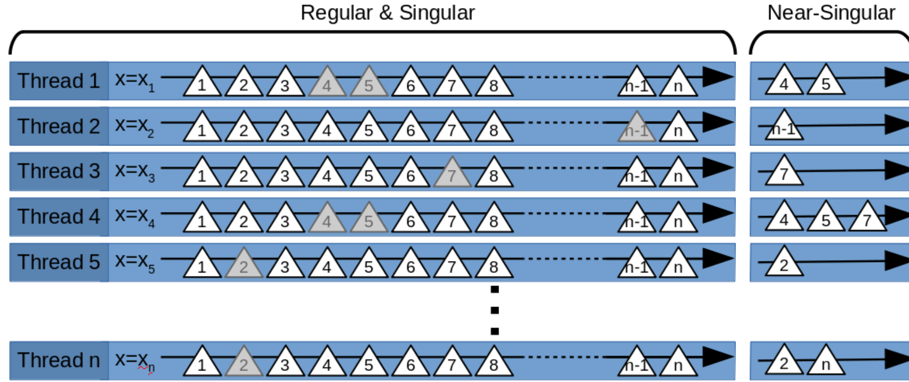


Figure 3.4: Near-singular assembly: The gray triangle are near-singular and are compute separately.

is equal to zero for degenerate triangles. Some computation will be unnecessary, i.e. canceled by a multiplication with zero. But it allows to handle all three case at once, without any divergence in the control flow.

This strategy allows for efficient implementation on a GPU for near-singular integrals with a small overhead for computing the near-singular triangles in the end.

3.3 Constraint Assembly

The constraint condition for a floating potential has the following form, see (2.13)

$$\int_{\Gamma_F} \frac{1}{2}\varepsilon\sigma + \varepsilon K'\sigma dS = 0. \tag{3.9}$$

If we plug in the approximation of $\sigma_i = \sum_i \sigma_i b_N^i$, it can be written as

$$\sum_i \sigma_i \int_{\Gamma_F} \frac{1}{2} b_N^i(\mathbf{x}) + \int_\Gamma \frac{(\mathbf{x}-\mathbf{y})\cdot\mathbf{n}(\mathbf{x})}{4\pi|\mathbf{x}-\mathbf{y}|^3} b_N^i(\mathbf{y}) dS_y dS_x \tag{3.10}$$

We apply a quadrature with points $\mathbf{x}_q$ and weights $w_q$for the outer integral.

$$\sum_i \sigma_i \sum_q^Q \frac{1}{2} w_q b_N^i(\mathbf{x}_q) + w_q \int_\Gamma \frac{(\mathbf{x}_q-\mathbf{y})\cdot\mathbf{n}(\mathbf{x}_q)}{|\mathbf{x}_q-\mathbf{y}|^3} b_N^i(\mathbf{y}) dS_y \tag{3.11}$$

The inner integral in (3.11) is identical to (3.8) if we use the vertex of the mesh as quadrature points $\mathbf{x}_q = \mathbf{x}_i$. Such a quadrature rule would be the 2D trapezoidal rule.

$$\int_K f(\mathbf{x})dx \approx \frac{|K|}{3}(f(\mathbf{a}^1) + f(\mathbf{a}^2) + f(\mathbf{a}^3)) \tag{3.12}$$

The computation of the discretized constraints is done in two steps. First we compute a double layer potential collocation matrix for the collocation points on the surface of the floating potential. This gives us the values of the second part of the integrand in the quadrature points. Then we use these entries to evaluate the outer integral with the 2D trapezoidal rule.

For the first part, we can use the same routines as for the assembly of the collocation matrix rows. The second part is implemented as a special reduction over the columns of the matrix. Where the reduction is done in parallel for all columns at once and according to the trapezoidal rule.

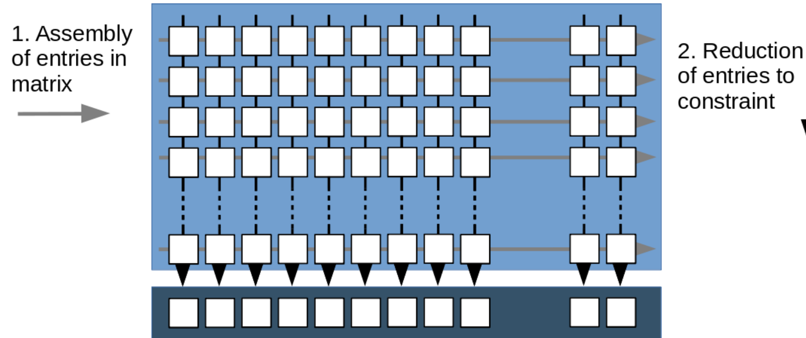


Figure 3.5: Spread out constraint assembly and then reduction with quadrature.

Order of Assembly The full matrix requires a lot of storage as does the first step of the constraint computation. We compute the constraint first, because the memory allocated for the constraint matrix can be freed after the reduction and be reuse for the system matrix.

3.4 GMRES

Now that we assembled the matrix we want to solve it. For that, we use an iterative solver, more specifically GMRES. The most compute intensive parts of the solving process are the matrix-vector multiplications during each iteration. We can do them efficiently on the GPU by computing the scalar product between the matrix rows and the vector in parallel. The matrix is already stored on the GPU, we only have to transfer the vector onto the GPU and extract the result after the matrix-vector multiplication from the GPU.

3.5 Postprocessing

For the evaluation of the potential and the electric field we have to compute integrals similar to (3.8). The differences are, that we now know the coefficients σ_i for the surface charge, the point $\mathbf{x}$ does not need to be part of the surface and that contributions from the basis function are summed together not spread out over a matrix.

$$E(\mathbf{x}) = \int_\Gamma \frac{(\mathbf{x}-\mathbf{y})}{4\pi|\mathbf{x}-\mathbf{y}|}\sigma(\mathbf{y})dS_y \approx \sum_{j=0}^{N} \sigma_j \int_\Gamma \frac{(\mathbf{x}-\mathbf{y})}{4\pi|\mathbf{x}-\mathbf{y}|} b_N^j(\mathbf{y})dS_y \quad \mathbf{x} \notin \Gamma \tag{3.13}$$

Thus we use the similar strategy as for the assembly. We evaluate multiple points in parallel. We can now easily use multiple threads for one point. We split the surface in disjoint subsurface and compute the integral contributions on them individually and, in a second step, sum these local integrals together to the result. This increases the performance on the GPU as we can fill the GPU with hundreds of threads even in the case of a low amount of evaluation points.

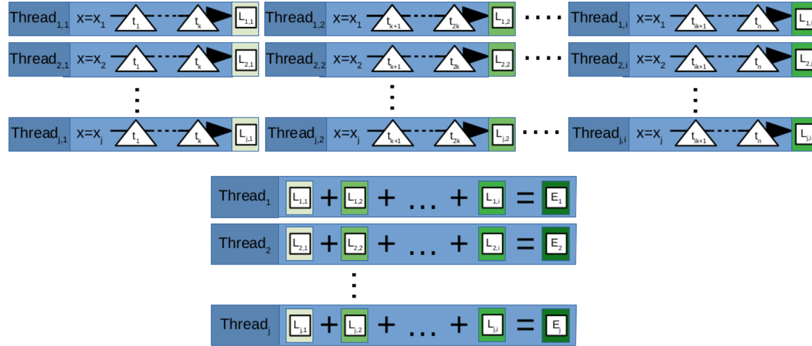


Figure 3.6: Parallelization of the evaluation of potential and electric field

Remark 1 We evaluate the potential and the electric field at the same time, we have to evaluate the expensive quadratic parameterization of the surface only half as often.

Remark 2 We have two versions for the evaluation. One for the case, when the point is on the surface since we have to consider the singular integrals in this case. And one for the case, when the point is not part of the surface since we only have to deal with near-singular integrals.

Evaluation of field lines For the evaluation of field lines the ordinary differential equation (2.24) has to be solved, where the start point will be a

point on the surface. We use the Euler method with a fixed step size. The computational expensive part is the evaluation of the electric field in a point in space. How this is done is discussed in the previous paragraph about postprocessing. Thus, here we will only describe the implementation of the stepping process.

The computation of the field lines is parallelized by computing the same step for all field lines at same time. We start with a set of points $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ which are the current end points of the field lines. At the beginning this will be a set of starting points on the surface. We compute the electric field in these points using the routine for postprocessing. With the electric field in the point $\mathbf{E}(\mathbf{x}_i)$ we can compute the next point

$$\mathbf{x}_i' = \mathbf{x}_i + \Delta s \mathbf{E}(\mathbf{x}_i)$$

where $\mathbf{x}_i'$ is the new point and Δs the fixed stepsize. But we before we update the end points, we have to test if the field line intersects with a surface as this terminates the field line. Thus we test if the field line will intersect with a triangle in the mesh during the step.

To avoid testing with all triangles, we created a short list of triangles for possible intersection during the computation of the electric field because there we iterate anyway over all triangle-point pairs in the evaluation of the electric field. We store all triangles that are closer to the point than the step size plus the circumradius of the triangle. All triangles are then checked for intersections after the computation of the electric field and know the direction of the next step. In case of curved triangles, we resort to a simplified intersection test. We check against the linear approximation with a large tolerance to compensate for inaccuracy and terminate the field line, if the field line is close to the flat triangle and thus an intersection with the curved element is likely. But this is just a crude heuristic that will need improvement in the future.

After we checked for intersections, we only continue with a new set of point X' whose field lines had no intersection. We stop either if the are no more unfinished field lines or if a maximum number of steps is reached.

3.6 Distributed-memory parallelization

Due to the use of dense matrices the problem sizes we can solve are limited by the amount of memory we have to store the matrix. To have more memory and thus to be able to solve large problems we need to use multiple GPU's. Luckily due to the fact that the rows of the matrix are independent from each other the matrix $\mathbf{A}$ can be easily split into equal sized row blocks $\mathbf{A}_1, ..., \mathbf{A}_n$ that can be computed and stored on different GPUs.

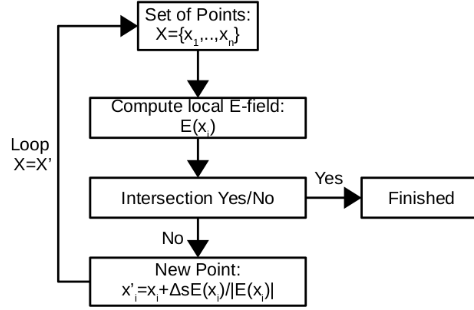


Figure 3.7: Loop of field line evaluation

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

The constraint can be computed as well on multiple GPUs. We split the set of points belonging to the floating potential surface and distribute them onto different GPUs. We compute the entries for the constraint evaluation locally and do a local reduction. Then, we transfer the local constraint row to the GPU with the corresponding matrix block. There, the constraint contributions from the different GPUs can be summed together to build the full constraint row and add it to the matrix.

The overall assembly is split in two phases. First, the assembly of the constraint and second the assembly of the system matrix. In the first phase we assemble the constraint contribution in matrix form. Then, we do the column-wise reduction of the matrix to the constraint row. In the second phase we assemble the full matrix in three steps. In the first step we compute all rows that have single layer entries. In the second step we compute all entries with double layer entries. In the third step we add the constraint entries to system matrix. This process is illustrated for two GPUs in figure 3.8.

Remark 1 The matrix-vector multiplication of the iterative solver is done locally on the row blocks. The local result is then stitched together on the master node that manages the iterative process.

Remark 2 For the post processing the distributed memory parallelization is simple. We just distribute the evaluation point equally onto the available GPUs.

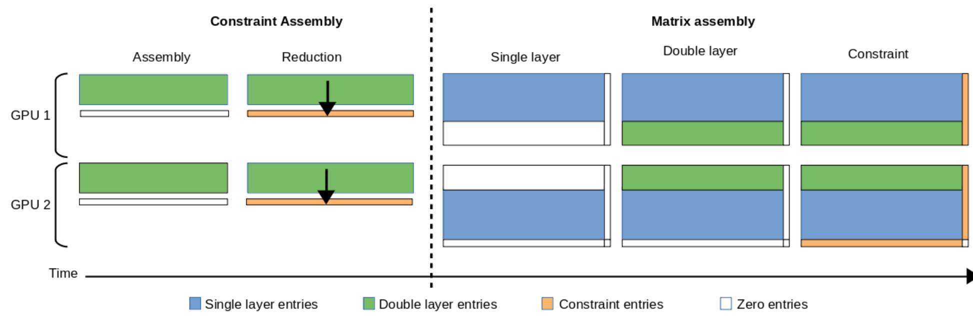


Figure 3.8: Example of assembly for a matrix with two GPUs.

Chapter 4

Numerical results

In this section the results of the numerical experiments are presented. First the results are compared in accuracy to the existing solver POLOPT [2], [4] for a small test case. Then the time to solution is compared for large cases and also for evaluating field lines.

4.1 Comparison H2Lib vs. POLOPT

First, we compare the accuracy of the H2Lib implementation with POLOPT.

4.1.1 Test cases

Bigmac & Sandwich These are two artificial models that cover most of the features that were implemented. The 'Bigmac' model consists of 4 slabs stacked on top of each other. The lowest is a conductor with an applied voltage of $100kV$, then an insulating dielectric slab, then again a conducting slab that has no source connected and and on top is another dielectric part, see figure 4.1. The 'Sandwich' model is almost the same except the floating conductor is reduce to the interface between the to dielectrics, see figure 4.2.

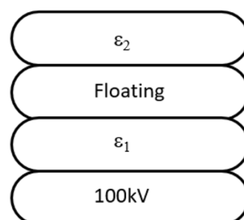


Figure 4.1: Cross-section of 'Bigmac' model.

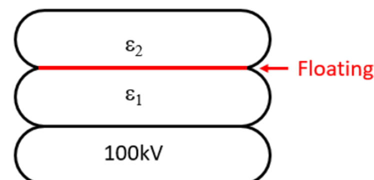


Figure 4.2: Cross-section of 'Sanwich' model.

The permittivity for the insulating dielectric slabs is varied, while everything else remains the same. The voltage of the surface of the floating conductor

of the 'Bigmac' can be found in table 4.1. The voltage for the floating sheet in the 'Sandwich' can be found in table 4.2. As reference the result of an 2D axis-symmetric charge simulation (ELFI) are also given in table 4.1 and 4.2.

ε_1	ε_2	ELFI(2D)	POLOPT	H2LIB
5	10	88.85kV	89.21kV	89.41kV
5	1	90.25kV	89.90kV	89.87kV
1	10	69.51kV	69.82kV	70.27kV
1	1	71.37kV	71.18kV	71.16kV

Table 4.1: Potential on conducting slab in the 'Bigmac' model for different material parameters for different solvers.

ε_1	ε_2	ELFI(2D)	POLOPT	H2LIB
5	10	90.99kV	91.29kV	92.15kV
5	1	93.48kV	94.75kV	94.72kV
1	10	72.59kV	73.48kV	73.38kV
1	1	78.52kV	80.43kV	80.42kV

Table 4.2: Potential on conducting sheet in the 'Sandwich' model for different materials for different solvers.

Overall the new H2Lib results match the value form POLOPT well. For higher material coefficients the difference are larger.

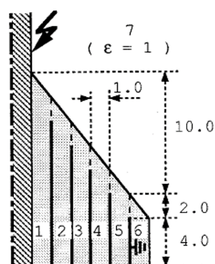


Figure 4.3: Cross-section drawing of the bushing.

Bushing A real-world problem that can be modeled is a bushing. In this case it consists of an insulator wrapped around a high-voltage conductor. Inside insulator, there are five thin conducting sheets. The outermost is assumed to be grounded, the potential of the other sheets is unknown, see figure 4.3 The values of the floating potentials are computed with both POLOPT and the H2Lib , see table 4.3. The results are again compared with ELFI.

	ELFI(2D)	POLOPT	H2LIB
U_{foil1}	70.8kV	70.15kV	70.22kV
U_{foil2}	51.4kV	50.47kV	50.50kV
U_{foil3}	35.0kV	34.00kV	34.02kV
U_{foil4}	18.9kV	18.02kV	18.02kV

Table 4.3: Potential on conducting sheet in the bushing.

4.1.2 Double vs. single precision

Using single precision instead of double precision decreases the memory needed to store the matrix by half. Furthermore standard consumer GPU are optimize for single precision since in graphic application the difference between double and single precision has no visible effect. Thus we tested if the use of single precision does impact the accuracy negatively.

	Single precsion	Double precision	Relative difference
U_{foil1}	70.219841kV	70.220740kV	1.28e-5
U_{foil2}	50.502563kV	50.504335kV	3.51e-5
U_{foil3}	34.017059kV	34.018866kV	5.31e-5
U_{foil4}	18.021936kV	18.022896kV	5.33e-4

Table 4.4: Potential on conducting sheet in the bushing for single precision and double precision.

As we can see in 4.4, the deviations of floating potential values are minimal. But the runtime can increase by a factor of four if we use double precision and a standard consumer GPU (GT 940M) with less double precision ALUs than single precision ALUs. If we do the same computation on a more expensive general purpose GPU (Tesla P100) we can see that the time only increases slightly, see table 4.5.

	Single precsion	Double precision
Time GT940M	2.34s	9.47s
Time P100	1.70s	1.86s

Table 4.5: Time to solution for different GPU models using single and double precision.

There is no evident drawback of using single precision compared to all the advantages, less memory and faster computation on cheaper hardware. Thus, for all further experiments single precision is used.

4.1.3 Time-to-solution comparison

After we showed in the previous subsections that we achieve similar results with the H2Lib implementation, we compare the time it takes to compute

the solution.

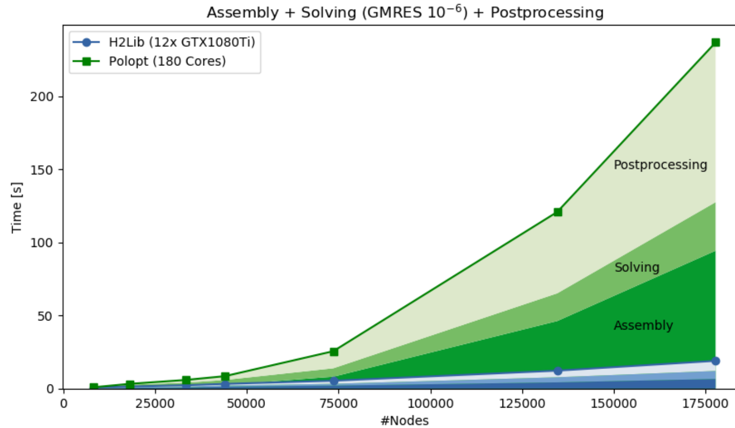


Figure 4.4: Cumulative times for assembly, solving and surface electric field computation for POLOPT and H2LIB.

In figure 4.4 we plotted the time it takes to assembly, solve and compute the surface electric field on different meshes for POLOPT and H2Lib. POLOPT used a total of 180 CPU cores distributed over 5 nodes with two 18-core CPUs each. For H2Lib we used a total of 12 NVidia GTX 1080Ti, distributed over multiple nodes. For POLOPT we can clearly see the quadratic scaling with the mesh size that is expected as the work scales quadratic with the input size.

In figure 4.5 we plotted the timings of H2Lib only. We can also see out that the scaling is also non-linear, but better than the scaling of POLOPT.

Benchmark problem We did some computation with the benchmark problem EXK0-GIS from ABB, see figure 4.6. The same geometry has been meshed in different resolutions. The meshes range from 68'218 nodes up to 330'706 nodes. We again used 180 cores for POLOPT. For the H2Lib the number of GPUs where chosen so that the full matrix stored in single precision fits in the combined memory of all GPUs. One GTX 1080Ti has 11 gigabytes of dedicated memory. Thus for the largest model we need a total of 40 GPUs. The timings can be found in table 4.6.

4.2 Field line computation

In the previous section, we focused on the results concerning the first part of the breakdown prediction, the computation of the electric field on the surface. The more expensive part in term of computation is the evaluation

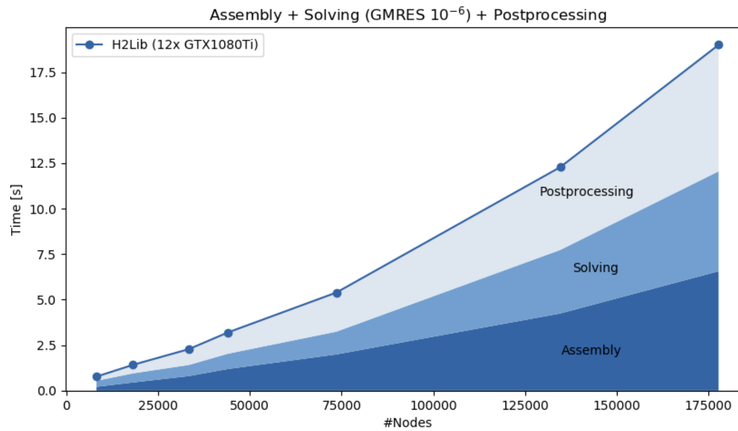


Figure 4.5: Cumulative times for assembly, solving and surface electric field computation for H2Lib.

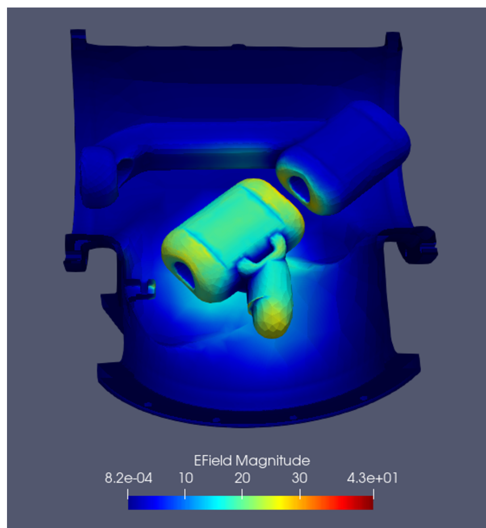


Figure 4.6: Crosssection of the EXK0-GIS model.

of field lines, because it can require hundreds of surface integrals evaluations to compute a single field line.

There is no current configuration of POLOPT that allows for the evaluation of the field lines in parallel. Therefore the time for the evaluation of field lines in table 4.7 is for the serial code only. With the parallelization of the evaluation process and the use of the GPUs we were able to compute 500 times more field lines in a shorter time than the existing tools.

For the result in table 4.7 we use the standard 180-cores for the computation with POLOPT and serial code for the fieldlines. For H2LIB we used 4 NVidia

#Nodes	#GPUs	H2Lib	POLOPT
68’218	2	18s	33s
140’183	8	19s	130s
232’029	20	26s	371s
330’706	40	34s	702s

Table 4.6: Computation time for different meshes. The H2Lib used multiple GTX1080Ti, while POLOPT is always executed on 180 CPU-Cores.

	POLOPT 180 Cores	H2Lib GPU 4xTesla P100
Assembly	11s	2s
Solving	11s	2s
Eval. **E**	11s	2s
Total	33s	6s
Field lines	102s for 17 fieldlines	67s for 9’219 fieldlines

Table 4.7: Computation and fieldline evaluation time. Fieldlines for POLOPT are computed on local workstation without parallelization! Model size: 68’218 Nodes.

Tesla P100 General-Purpose GPUs.

Figure 4.7 shows the 17 field lines that are computed with POLOPT. The highlighted points are the 9’219 starting points that were computed with the H2Lib in the same time.

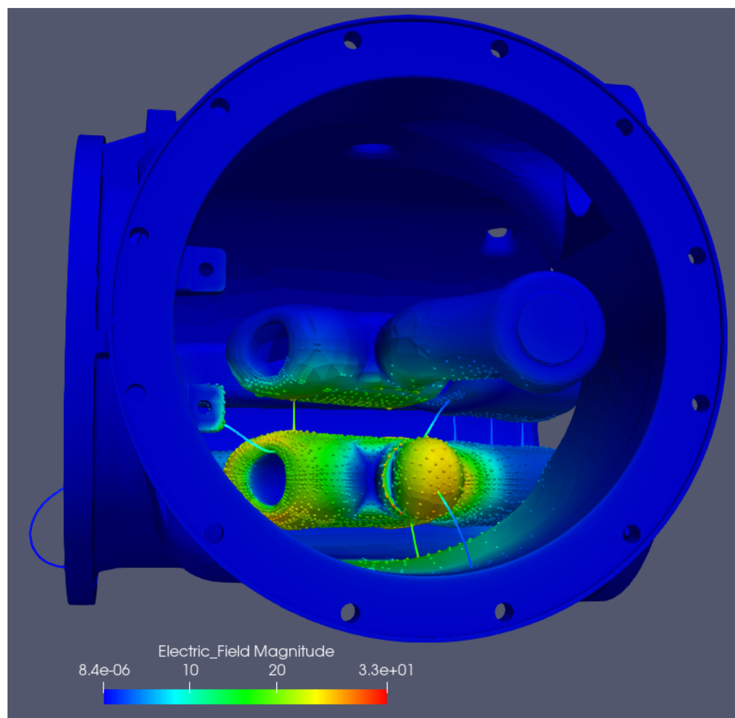


Figure 4.7: EXK-0 GIS with surface electric field and selected fieldlines. The highlight points are all critical points.

Chapter 5

Conclusion & Outlook

In this thesis we implemented the assembly, solving and postprocessing of the electrostatic problem, distributed over multiple GPUs. We achieved the distribution of the computations over several dozen GPUs with good load balancing. The large scale parallelization is achieved due to the fact that the matrix can be split into independent row blocks. These block are then small enough to be computed and stored on a single GPU. By following the data-parallel programming model for GPUs, we implemented the algorithms to use performance benefits of GPUs. Thus, we were able to reduce time-to-solution from minutes to seconds for large real-world problems. It was also experimentally verified that single precision can be used to do the computation and thus we were able to use cheaper consumer GPUs and cut the memory requirement in half. In addition, we parallelized the postprocessing evaluation of field lines. The parallelized version computes thousands of field lines paths in the same time as before only a few selected breakdown paths were compute with a serial code.

There is further work required in the area of the evaluation of dielectric breakdown paths. The detection of intersection with the curved triangle needs to be improved. This would allow for the evaluation of breakdown paths in geometries with dielectrics. That is necessary for the complete dielectric design process. Also compression techniques for the matrix should be considered. As the main obstacle to solve bigger problems is the need to store the full matrix.

Appendix A

Appendix

In following we have code of OpenCL kernel for the computation of single layer row of the collocation matrix.

A.1 OpenCL kernel

Listing A.1: OpenCL kernel for the assembly of single layer with near-singular integration

```
/** @brief OpenCL Kernel to compute the single layer row of collocation matirx
 *
 *
 * @param vertices  Number of nodes in grid
 * @param triangles Number of triagles
 * @param x         Node coordinates
 * @param t         Corner indices of triangles
 * @param s         Edge midpoint indices of triangles
 * @param m         Edge midpoint coordinates
 * @param o         Cirmcumcenter of triangle
 * @param d         Circumcircle radius
 * @param ng        Number of quadrature points
 * @param xg        Quadrature points
 * @param wg        Quadrature weight
 * @param nt        Number of triangles in local to global mapping
 * @param tri2vtx   Local to global mapping
 * @param nv        Number of collocation points
 * @param vtx       Collocation point indices
 * @param crit      Cache to store near-singular triangles
 * @param eps       Tolearance for near-singular triangles
 * @param Ga        Matrix
 * @param ldG       Leading dimension of Matrix
 */
__kernel void
buildmatrix_block_near_sl(int vertices, int triangles,
      __global const real (*x)[3], __global const int (*t)[3],
      __global const int (*s)[3], __global const real (*m)[3],
      __global const real (*o)[3], __global const real *d,
      int ng, __global const real *xg, __global const real *wg,
      int nt, __global const int *tri2vtx,
      int nv, __global const int *vtx,
      __global int *crit, real eps,
      __global real *Ga, int ldG)
```

```
{
  const real factor = 1.0 / (4.0 * 3.14159265358979323846);
  const unsigned permutation[] = { 0, 1, 2, 0, 1 };
  const unsigned *py;
  real x0, x1, x2, y0, y1, y2, n0, n1, n2;
  real xa[3], xb[3], xc[3], xab[3], xbc[3], xca[3];
  real dt[3], ds[3];
  real na[3], nb[3], nc[3], nab[3], nbc[3], nca[3];
  real b0, b1, b2;
  real gy;
  real d0, d1, d2;
  real kv;
  real sum0, sum1, sum2;

  real norm2;
  real dist[3];
  real pa[2];
  real aa[3][2];
  real F[2][2];
  real xp,yp,wp;

  real ab[3], ac[3], bc[3], ap[3], bp[3], cp[3];
  real d3,d4,d5,d6,va,vb,vc,w1,w2,w3,denom,w,v;

  uint ncrit;

  uint i, j, l, ii, jj, k, k1, k2;

  //Get global id
  i = get_global_id(0);

  //Check if id in range
  if(i >= nv)
    return;

  //Get collocaiton point index
  ii = (vtx ? vtx[i] : i);

  //collocation point
  x0 = x[ii][0];
  x1 = x[ii][1];
  x2 = x[ii][2];

  //Reset number of near singular triangles
  ncrit = 0;

  //Set matrix row to zero
  for(jj=0; jj<vertices; jj++)
    Ga[ii+jj*ldG] = 0.0;


  //Loop over triangle
  for(j=0; j<nt; j++)
  {
    //Get triangle index
    jj = tri2vtx[4*j];

    //Distance to cicumcenter of triangle
    dist[0] = o[jj][0]-x0;
    dist[1] = o[jj][1]-x1;
    dist[2] = o[jj][2]-x2;
```

```
norm2 = dist[0]*dist[0]+dist[1]*dist[1]+dist[2]*dist[2];

//Check if ponint is inside critcal sphere and not singular case
if(norm2 < d[jj]*eps*eps
   && t[jj][0] != ii
   && t[jj][1] != ii
   && t[jj][2] != ii)
{
  //Add to near-singular list
  crit[i+nv*ncrit] = j;
  ncrit++;
  //Go to next triangle
  continue;
}

//Rotate reference element for singular case
py = permutation;
if(t[j][1] == ii)
  py = permutation + 1;
else if(t[j][2] == ii)
  py = permutation + 2;

//Get corners of triangle
xa[0] = x[t[jj][py[0]]][0];
xa[1] = x[t[jj][py[0]]][1];
xa[2] = x[t[jj][py[0]]][2];
xb[0] = x[t[jj][py[1]]][0];
xb[1] = x[t[jj][py[1]]][1];
xb[2] = x[t[jj][py[1]]][2];
xc[0] = x[t[jj][py[2]]][0];
xc[1] = x[t[jj][py[2]]][1];
xc[2] = x[t[jj][py[2]]][2];

//Get midpoints of triangle
xbc[0] = m[s[jj][py[0]]][0];
xbc[1] = m[s[jj][py[0]]][1];
xbc[2] = m[s[jj][py[0]]][2];
xca[0] = m[s[jj][py[1]]][0];
xca[1] = m[s[jj][py[1]]][1];
xca[2] = m[s[jj][py[1]]][2];
xab[0] = m[s[jj][py[2]]][0];
xab[1] = m[s[jj][py[2]]][1];
xab[2] = m[s[jj][py[2]]][2];

//Compute difference vectors for midpoints
xbc[0] = 4.0 * xbc[0] - 2.0 * (xb[0] + xc[0]);
xbc[1] = 4.0 * xbc[1] - 2.0 * (xb[1] + xc[1]);
xbc[2] = 4.0 * xbc[2] - 2.0 * (xb[2] + xc[2]);
xca[0] = 4.0 * xca[0] - 2.0 * (xc[0] + xa[0]);
xca[1] = 4.0 * xca[1] - 2.0 * (xc[1] + xa[1]);
xca[2] = 4.0 * xca[2] - 2.0 * (xc[2] + xa[2]);
xab[0] = 4.0 * xab[0] - 2.0 * (xa[0] + xb[0]);
xab[1] = 4.0 * xab[1] - 2.0 * (xa[1] + xb[1]);
xab[2] = 4.0 * xab[2] - 2.0 * (xa[2] + xb[2]);

//Compute quadratic parametrization for normal vector
dt[0] = xb[0] - xa[0] + xab[0];
dt[1] = xb[1] - xa[1] + xab[1];
dt[2] = xb[2] - xa[2] + xab[2];
ds[0] = xc[0] - xa[0] + xca[0];
ds[1] = xc[1] - xa[1] + xca[1];
ds[2] = xc[2] - xa[2] + xca[2];
```

```
na[0] = dt[1] * ds[2] - dt[2] * ds[1];
na[1] = dt[2] * ds[0] - dt[0] * ds[2];
na[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] - xab[0];
dt[1] = xb[1] - xa[1] - xab[1];
dt[2] = xb[2] - xa[2] - xab[2];
ds[0] = xc[0] - xa[0] + xbc[0] - xab[0];
ds[1] = xc[1] - xa[1] + xbc[1] - xab[1];
ds[2] = xc[2] - xa[2] + xbc[2] - xab[2];
nb[0] = dt[1] * ds[2] - dt[2] * ds[1];
nb[1] = dt[2] * ds[0] - dt[0] * ds[2];
nb[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] + xbc[0] - xca[0];
dt[1] = xb[1] - xa[1] + xbc[1] - xca[1];
dt[2] = xb[2] - xa[2] + xbc[2] - xca[2];
ds[0] = xc[0] - xa[0] - xca[0];
ds[1] = xc[1] - xa[1] - xca[1];
ds[2] = xc[2] - xa[2] - xca[2];
nc[0] = dt[1] * ds[2] - dt[2] * ds[1];
nc[1] = dt[2] * ds[0] - dt[0] * ds[2];
nc[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] + (xbc[0] - xca[0] - xab[0]) * 0.5;
dt[1] = xb[1] - xa[1] + (xbc[1] - xca[1] - xab[1]) * 0.5;
dt[2] = xb[2] - xa[2] + (xbc[2] - xca[2] - xab[2]) * 0.5;
ds[0] = xc[0] - xa[0] + (xbc[0] - xca[0] - xab[0]) * 0.5;
ds[1] = xc[1] - xa[1] + (xbc[1] - xca[1] - xab[1]) * 0.5;
ds[2] = xc[2] - xa[2] + (xbc[2] - xca[2] - xab[2]) * 0.5;
nbc[0] = dt[1] * ds[2] - dt[2] * ds[1];
nbc[1] = dt[2] * ds[0] - dt[0] * ds[2];
nbc[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] + (xbc[0] + xab[0] - xca[0]) * 0.5;
dt[1] = xb[1] - xa[1] + (xbc[1] + xab[1] - xca[1]) * 0.5;
dt[2] = xb[2] - xa[2] + (xbc[2] + xab[2] - xca[2]) * 0.5;
ds[0] = xc[0] - xa[0];
ds[1] = xc[1] - xa[1];
ds[2] = xc[2] - xa[2];
nca[0] = dt[1] * ds[2] - dt[2] * ds[1];
nca[1] = dt[2] * ds[0] - dt[0] * ds[2];
nca[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0];
dt[1] = xb[1] - xa[1];
dt[2] = xb[2] - xa[2];
ds[0] = xc[0] - xa[0] + (xca[0] + xbc[0] - xab[0]) * 0.5;
ds[1] = xc[1] - xa[1] + (xca[1] + xbc[1] - xab[1]) * 0.5;
ds[2] = xc[2] - xa[2] + (xca[2] + xbc[2] - xab[2]) * 0.5;
nab[0] = dt[1] * ds[2] - dt[2] * ds[1];
nab[1] = dt[2] * ds[0] - dt[0] * ds[2];
nab[2] = dt[0] * ds[1] - dt[1] * ds[0];

nbc[0] = 4.0 * nbc[0] - 2.0 * (nb[0] + nc[0]);
nbc[1] = 4.0 * nbc[1] - 2.0 * (nb[1] + nc[1]);
nbc[2] = 4.0 * nbc[2] - 2.0 * (nb[2] + nc[2]);
nca[0] = 4.0 * nca[0] - 2.0 * (nc[0] + na[0]);
nca[1] = 4.0 * nca[1] - 2.0 * (nc[1] + na[1]);
nca[2] = 4.0 * nca[2] - 2.0 * (nc[2] + na[2]);
nab[0] = 4.0 * nab[0] - 2.0 * (na[0] + nb[0]);
nab[1] = 4.0 * nab[1] - 2.0 * (na[1] + nb[1]);
```

```
  nab[2] = 4.0 * nab[2] - 2.0 * (na[2] + nb[2]);

  //Reset local contributions
  sum0 = 0.0;
  sum1 = 0.0;
  sum2 = 0.0;

  //Quadrature
  for(k1=0; k1<ng; k1++)
  {
    for(k2=0; k2<ng; k2++)
    {
      //Local shape functions
      b0 = 1.0 - xg[k1];
      b1 = xg[k1] * (1.0 - xg[k2]);
      b2 = xg[k1] * xg[k2];

      //Global quadrature points on trinagle
      y0 = (b0 * xa[0] + b1 * xb[0] + b2 * xc[0]
            + b0 * b1 * xab[0] + b1 * b2 * xbc[0] + b2 * b0 * xca[0]);
      y1 = (b0 * xa[1] + b1 * xb[1] + b2 * xc[1]
            + b0 * b1 * xab[1] + b1 * b2 * xbc[1] + b2 * b0 * xca[1]);
      y2 = (b0 * xa[2] + b1 * xb[2] + b2 * xc[2]
            + b0 * b1 * xab[2] + b1 * b2 * xbc[2] + b2 * b0 * xca[2]);

      //Compute Gram determinant
      n0 = (b0 * na[0] + b1 * nb[0] + b2 * nc[0]
            + b0 * b1 * nab[0] + b1 * b2 * nbc[0] + b2 * b0 * nca[0]);
      n1 = (b0 * na[1] + b1 * nb[1] + b2 * nc[1]
            + b0 * b1 * nab[1] + b1 * b2 * nbc[1] + b2 * b0 * nca[1]);
      n2 = (b0 * na[2] + b1 * nb[2] + b2 * nc[2]
            + b0 * b1 * nab[2] + b1 * b2 * nbc[2] + b2 * b0 * nca[2]);

      gy = sqrt(n0 * n0 + n1 * n1 + n2 * n2);

      //Evaluate kernel
      d0 = x0 - y0;
      d1 = x1 - y1;
      d2 = x2 - y2;
      kv = factor / sqrt(d0 * d0 + d1 * d1 + d2 * d2);

      kv *= xg[k1] * wg[k1] * wg[k2] * gy;

      //Add contribution
      sum0 += b0 * kv;
      sum1 += b1 * kv;
      sum2 += b2 * kv;
    }
  }

  //Add to matrix
  if(tri2vtx[4*j+py[0]+1] != ~0u)
    Ga[ii+tri2vtx[4*j+py[0]+1]*ldG] += sum0;
  if(tri2vtx[4*j+py[1]+1] != ~0u)
    Ga[ii+tri2vtx[4*j+py[1]+1]*ldG] += sum1;
  if(tri2vtx[4*j+py[2]+1] != ~0u)
    Ga[ii+tri2vtx[4*j+py[2]+1]*ldG] += sum2;
}



//Loop over near-singular triangles
```

```
for(l=0; l<ncrit; l++)
{
  //Near-singular element index
  j = crit[i+nv*l];

  //Get triangle index
  jj = tri2vtx[4*j];

  //Corners of triangle
  xa[0] = x[t[jj][0]][0];
  xa[1] = x[t[jj][0]][1];
  xa[2] = x[t[jj][0]][2];
  xb[0] = x[t[jj][1]][0];
  xb[1] = x[t[jj][1]][1];
  xb[2] = x[t[jj][1]][2];
  xc[0] = x[t[jj][2]][0];
  xc[1] = x[t[jj][2]][1];
  xc[2] = x[t[jj][2]][2];

  //Compute nearest point on flat triangle to collocation point
  //Based on 5.1.5 in Real-Time Collision Detection by Christer Ericson
  ab[0] = xb[0]-xa[0];
  ab[1] = xb[1]-xa[1];
  ab[2] = xb[2]-xa[2];

  ac[0] = xc[0]-xa[0];
  ac[1] = xc[1]-xa[1];
  ac[2] = xc[2]-xa[2];

  ap[0] = x0-xa[0];
  ap[1] = x1-xa[1];
  ap[2] = x2-xa[2];

  bp[0] = x0-xb[0];
  bp[1] = x1-xb[1];
  bp[2] = x2-xb[2];

  cp[0] = x0-xc[0];
  cp[1] = x1-xc[1];
  cp[2] = x2-xc[2];

  d1 = ab[0]*ap[0]+ab[1]*ap[1]+ab[2]*ap[2];
  d2 = ac[0]*ap[0]+ac[1]*ap[1]+ac[2]*ap[2];
  d3 = ab[0]*bp[0]+ab[1]*bp[1]+ab[2]*bp[2];
  d4 = ac[0]*bp[0]+ac[1]*bp[1]+ac[2]*bp[2];
  d5 = ab[0]*cp[0]+ab[1]*cp[1]+ab[2]*cp[2];
  d6 = ac[0]*cp[0]+ac[1]*cp[1]+ac[2]*cp[2];

  vc = d1*d4-d3*d2;
  vb = d5*d2-d1*d6;
  va = d3*d6-d5*d4;

  w1 = d1/(d1-d3);
  w2 = d2/(d2-d6);
  w3 = (d4-d3)/((d4-d3)+ (d5-d6));

  denom = 1.0/(va+vb+vc);
  v = vb*denom;
  w = vc*denom;

  //Get barycentric coordinates of closest point
  //Beyond corner A
```

```
if(d1 <= 0.0 && d2 <= 0.0)
{
  pa[0] = 0;
  pa[1] = 0;
}
//Beyond corner B
else if(d3 >= 0.0 && d4 <= d3 )
{
  pa[0] = 1;
  pa[1] = 0;
}
//Beyond corner C
else if(d6 >= 0.0 && d5 <= d6 )
{
  pa[0] = 1;
  pa[1] = 1;
}
//Under edege c
else if(vc  <= 0.0 && d1 >= 0.0 && d3 <= 0.0)
{
  pa[0] = w1;
  pa[1] = 0.0;
}
//Under edge b
else if(vb  <= 0.0 && d2 >= 0.0 && d6 <= 0.0)
{
  pa[0] = w2;
  pa[1] = w2;
}
//Under edge a
else if(va <= 0.0 && (d4-d3) >= 0.0 && (d5-d6) >= 0.0)
{
  pa[0] = 1.0;
  pa[1] = w3;
}
//Inside the triangle
else
{
  pa[0] = v+w;
  pa[1] = w;
}

//Coordinates of feference triangle
aa[0][0] = 0.0; aa[0][1] = 0.0;
aa[1][0] = 1.0; aa[1][1] = 0.0;
aa[2][0] = 1.0; aa[2][1] = 1.0;

//Midpoints of triangle
xbc[0] = m[s[jj]][0]][0];
xbc[1] = m[s[jj]][0]][1];
xbc[2] = m[s[jj]][0]][2];
xca[0] = m[s[jj]][1]][0];
xca[1] = m[s[jj]][1]][1];
xca[2] = m[s[jj]][1]][2];
xab[0] = m[s[jj]][2]][0];
xab[1] = m[s[jj]][2]][1];
xab[2] = m[s[jj]][2]][2];

//Compute difference vectors for midpoints
xbc[0] = 4.0 * xbc[0] - 2.0 * (xb[0] + xc[0]);
xbc[1] = 4.0 * xbc[1] - 2.0 * (xb[1] + xc[1]);
xbc[2] = 4.0 * xbc[2] - 2.0 * (xb[2] + xc[2]);
```

```
xca[0] = 4.0 * xca[0] - 2.0 * (xc[0] + xa[0]);
xca[1] = 4.0 * xca[1] - 2.0 * (xc[1] + xa[1]);
xca[2] = 4.0 * xca[2] - 2.0 * (xc[2] + xa[2]);
xab[0] = 4.0 * xab[0] - 2.0 * (xa[0] + xb[0]);
xab[1] = 4.0 * xab[1] - 2.0 * (xa[1] + xb[1]);
xab[2] = 4.0 * xab[2] - 2.0 * (xa[2] + xb[2]);

//Compute quadratic parametrization for normal vector
dt[0] = xb[0] - xa[0] + xab[0];
dt[1] = xb[1] - xa[1] + xab[1];
dt[2] = xb[2] - xa[2] + xab[2];
ds[0] = xc[0] - xa[0] + xca[0];
ds[1] = xc[1] - xa[1] + xca[1];
ds[2] = xc[2] - xa[2] + xca[2];
na[0] = dt[1] * ds[2] - dt[2] * ds[1];
na[1] = dt[2] * ds[0] - dt[0] * ds[2];
na[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] - xab[0];
dt[1] = xb[1] - xa[1] - xab[1];
dt[2] = xb[2] - xa[2] - xab[2];
ds[0] = xc[0] - xa[0] + xbc[0] - xab[0];
ds[1] = xc[1] - xa[1] + xbc[1] - xab[1];
ds[2] = xc[2] - xa[2] + xbc[2] - xab[2];
nb[0] = dt[1] * ds[2] - dt[2] * ds[1];
nb[1] = dt[2] * ds[0] - dt[0] * ds[2];
nb[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] + xbc[0] - xca[0];
dt[1] = xb[1] - xa[1] + xbc[1] - xca[1];
dt[2] = xb[2] - xa[2] + xbc[2] - xca[2];
ds[0] = xc[0] - xa[0] - xca[0];
ds[1] = xc[1] - xa[1] - xca[1];
ds[2] = xc[2] - xa[2] - xca[2];
nc[0] = dt[1] * ds[2] - dt[2] * ds[1];
nc[1] = dt[2] * ds[0] - dt[0] * ds[2];
nc[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] + (xbc[0] - xca[0] - xab[0]) * 0.5;
dt[1] = xb[1] - xa[1] + (xbc[1] - xca[1] - xab[1]) * 0.5;
dt[2] = xb[2] - xa[2] + (xbc[2] - xca[2] - xab[2]) * 0.5;
ds[0] = xc[0] - xa[0] + (xbc[0] - xca[0] - xab[0]) * 0.5;
ds[1] = xc[1] - xa[1] + (xbc[1] - xca[1] - xab[1]) * 0.5;
ds[2] = xc[2] - xa[2] + (xbc[2] - xca[2] - xab[2]) * 0.5;
nbc[0] = dt[1] * ds[2] - dt[2] * ds[1];
nbc[1] = dt[2] * ds[0] - dt[0] * ds[2];
nbc[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0] + (xbc[0] + xab[0] - xca[0]) * 0.5;
dt[1] = xb[1] - xa[1] + (xbc[1] + xab[1] - xca[1]) * 0.5;
dt[2] = xb[2] - xa[2] + (xbc[2] + xab[2] - xca[2]) * 0.5;
ds[0] = xc[0] - xa[0];
ds[1] = xc[1] - xa[1];
ds[2] = xc[2] - xa[2];
nca[0] = dt[1] * ds[2] - dt[2] * ds[1];
nca[1] = dt[2] * ds[0] - dt[0] * ds[2];
nca[2] = dt[0] * ds[1] - dt[1] * ds[0];

dt[0] = xb[0] - xa[0];
dt[1] = xb[1] - xa[1];
dt[2] = xb[2] - xa[2];
ds[0] = xc[0] - xa[0] + (xca[0] + xbc[0] - xab[0]) * 0.5;
```

```
ds[1] = xc[1] - xa[1] + (xca[1] + xbc[1] - xab[1]) * 0.5;
ds[2] = xc[2] - xa[2] + (xca[2] + xbc[2] - xab[2]) * 0.5;
nab[0] = dt[1] * ds[2] - dt[2] * ds[1];
nab[1] = dt[2] * ds[0] - dt[0] * ds[2];
nab[2] = dt[0] * ds[1] - dt[1] * ds[0];

nbc[0] = 4.0 * nbc[0] - 2.0 * (nb[0] + nc[0]);
nbc[1] = 4.0 * nbc[1] - 2.0 * (nb[1] + nc[1]);
nbc[2] = 4.0 * nbc[2] - 2.0 * (nb[2] + nc[2]);
nca[0] = 4.0 * nca[0] - 2.0 * (nc[0] + na[0]);
nca[1] = 4.0 * nca[1] - 2.0 * (nc[1] + na[1]);
nca[2] = 4.0 * nca[2] - 2.0 * (nc[2] + na[2]);
nab[0] = 4.0 * nab[0] - 2.0 * (na[0] + nb[0]);
nab[1] = 4.0 * nab[1] - 2.0 * (na[1] + nb[1]);
nab[2] = 4.0 * nab[2] - 2.0 * (na[2] + nb[2]);

 //Reset local contributions
sum0 = 0.0;
sum1 = 0.0;
sum2 = 0.0;

//Loop over subtriangles
for(k=0; k<3; k++)
{
  //Transformation of subtriangle
  F[0][0] = aa[k][0]-pa[0];
  F[0][1] = aa[(k+1)%3][0]-aa[k][0];
  F[1][0] = aa[k][1]-pa[1];
  F[1][1] = aa[(k+1)%3][1]-aa[k][1];

  //Loop over quadrature points
  for(k1=0; k1<ng; k1++)
  {
    for(k2=0; k2<ng; k2++)
    {
      //Quadrature point
      xp = F[0][0]*xg[k1]+F[0][1]*(xg[k1]*xg[k2])+pa[0];
      yp = F[1][0]*xg[k1]+F[1][1]*(xg[k1]*xg[k2])+pa[1];
      //Quadrature weight
      wp = (F[0][0]*F[1][1]-F[0][1]*F[1][0])*xg[k1]*wg[k1]*wg[k2];

      //Local shape fucntions
      b0 = 1.0 - xp;
      b1 = xp - yp;
      b2 = yp;

      //Global quadrature points on triangle
      y0 = (b0 * xa[0] + b1 * xb[0] + b2 * xc[0]
          + b0 * b1 * xab[0] + b1 * b2 * xbc[0] + b2 * b0 * xca[0]);
      y1 = (b0 * xa[1] + b1 * xb[1] + b2 * xc[1]
            + b0 * b1 * xab[1] + b1 * b2 * xbc[1] + b2 * b0 * xca[1]);
      y2 = (b0 * xa[2] + b1 * xb[2] + b2 * xc[2]
            + b0 * b1 * xab[2] + b1 * b2 * xbc[2] + b2 * b0 * xca[2]);

      //Compute Gram determinant
      n0 = (b0 * na[0] + b1 * nb[0] + b2 * nc[0]
            + b0 * b1 * nab[0] + b1 * b2 * nbc[0] + b2 * b0 * nca[0]);
      n1 = (b0 * na[1] + b1 * nb[1] + b2 * nc[1]
            + b0 * b1 * nab[1] + b1 * b2 * nbc[1] + b2 * b0 * nca[1]);
      n2 = (b0 * na[2] + b1 * nb[2] + b2 * nc[2]
            + b0 * b1 * nab[2] + b1 * b2 * nbc[2] + b2 * b0 * nca[2]);
```

```
          gy = sqrt(n0 * n0 + n1 * n1 + n2 * n2);

          //Evaluate kernel
          d0 = x0 - y0;
          d1 = x1 - y1;
          d2 = x2 - y2;
          kv = factor / sqrt(d0 * d0 + d1 * d1 + d2 * d2);

          kv *= wp * gy;

          //Add contribution
          sum0 += b0 * kv;
          sum1 += b1 * kv;
          sum2 += b2 * kv;
        }
      }
    }

    //Add to matrix
    if(tri2vtx[4*j+1] != ~0u)
      Ga[ii+tri2vtx[4*j+1]*ldG] += sum0;
    if(tri2vtx[4*j+2] != ~0u)
      Ga[ii+tri2vtx[4*j+2]*ldG] += sum1;
    if(tri2vtx[4*j+3] != ~0u)
      Ga[ii+tri2vtx[4*j+3]*ldG] += sum2;
  }
}
```

Appendix B

Manual

B.1 Prerequisites

The code requires the following hardware and software:

- Access to at least one Nvidia GPU with installed drivers
- OpenCL header files
- MPI implementation
- C compiler
- Make

B.1.1 Code structure

- The 'Library' folder contains: A modified version of H2Lib. With support for .tra-file input and the collocation method for linear basis functions.
- The 'OpenCL' folder contains:
  - C-files for the functions which load and execute the OpenCL kernels.
  - The OpenCL kernels are the 'Kernel' directory.
- The 'Meshes' directory contains input meshes used in the numerical experiment
- The 'Makefile' is to build the code.
- 'option.inc.default' contains settings for H2Lib.
- 'test_mpi_ocl_near_general.c' Main file for computations without post-processing.

- 'test_mpi_ocl_field.c' Main file for computation with field lines.

B.1.2 Compile time options

Single or double precision Uncomment option 'USE_FLOAT' in 'option.inc.default' to use single precision.

Export matrices Uncomment the option 'USE_NETCDF' to export matrices after computations for debugging. Needs 'netcdf' to be installed on machine.

B.1.3 Running examples

A simple example can be executed by the following commands in the top directory of the code. The '-g' argument indicates the number on a single node.

```
module load mpi
make
./test_mpi_ocl_near_general -g 1 Path/To/TraFile.tra
```

This computes the surface electric field on one GPU without MPI.

```
mpirun -np 4 ./test_mpi_ocl_near_general -g 2 Path/To/TraFile.tra
```

This computes the surface electric field on two nodes with two GPUs each.

Bibliography

[1] Dominic Amann, Andreas Blaszczyk, Günther Of, and Olaf Steinbach. Simulation of floating potentials in industrial applications by boundary element methods. *Journal of Mathematics in Industry*, 4(1):13, Oct 2014.

[2] A. Blaszczyk and H. Steinbigler. Region-oriented charge simulation. *IEEE Transactions on Magnetics*, 30(5):2924–2927, Sep. 1994.

[3] Andreas Blaszczyk, Jonas Ekeberg, Sergey Pancheshnyi, and Magne Saxegaard. Virtual high voltage lab. In Ulrich Langer, Wolfgang Amrhein, and Walter Zulehner, editors, *Scientific Computing in Electrical Engineering*, pages 255–263, Cham, 2018. Springer International Publishing.

[4] N. de Kock, M. Mendik, Z. Andjelic, and A. Blaszczyk. Application of the 3d boundary element method in the design of ehv gis components. *IEEE Electrical Insulation Magazine*, 14(3):17–22, May 1998.

[5] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., USA, 2004.

[6] Steffen Börm et. al. H2lib, visited on 2020-02-03. http://h2lib.org.

[7] Khronos OpenCL Working Group. *The OpenCL Specification, v 1.2*, 2012, visted on 2020-02-03. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[8] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Dielectric Breakdown Prediction wiht GPU-Accelerated BEM

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Münger	Cedric

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Schlieren, 12.02.2020	C. Münger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.