# Designing a high-performance boundary element library with OpenCL and Numba

**T. Betcke**
Department of Mathematics, University College London

**M. W. Scroggs**
Department of Engineering, University of Cambridge

***Abstract*—The Bempp boundary element library is a well known library for the simulation of a range of electrostatic, acoustic and electromagnetic problems in homogeneous bounded and unbounded domains. It originally started as a traditional C++ library with a Python interface. Over the last two years we have completely redesigned Bempp as a native Python library, called Bempp-cl, that provides computational backends for OpenCL (using PyOpenCL) and Numba. The OpenCL backend implements kernels for GPUs and SIMD optimised for CPUs. In this paper we will discuss the design of Bempp-cl, provide performance comparisons on different compute devices and discuss the advantages and disadvantages of OpenCL as compared to Numba.**

■ **THE BEMPP BOUNDARY ELEMENT LIBRARY** (originally named BEM++) started in 2011 as a project to develop an open-source C++ library for the fast solution of boundary integral equations. The original release came with a simple Python wrapper to the C++ interface. Over time more and more functionality was moved into the Python interface. However, computationally intensive routines and the main data structures remained in C++. This proved a burden when efforts started to modernise the library to be able to make better use of Single-Instruction-Multiple-Data (SIMD) optimisation on CPUs and offload computation to GPUs. In 2018 we decided to rewrite the computational core from scratch. The aims were to support explicit SIMD optimisation on CPUs with various instruction lengths, be able to offload computations to AMD, Intel, and Nvidia GPUs, and to base the complete codebase on Python. These aims naturally lead to the choice of building a Python library based around OpenCL (using the PyOpenCL interface) and Numba. The name of the new library was Bempp-cl. At the end of 2019 we released an initial version 0.1 that we used to test the library in a number of real-world scenarios. This was followed later in 2020 by version 0.2, the first release that we considered feature complete and mature for application use.

Since then we have used Bempp-cl in a number of practical application and many of our users are slowly migrating to it from the old C++ based Bempp. In this article we want to discuss the design choices behind Bempp-cl and provide a number of performance benchmarks on different compute devices, including CPUs, AMD, Intel, and Nvidia GPUs.

[To be continued with some more details about content and overview of the Sections]

## Boundary element methods with Bempp

In this section we want to provide a brief introduction to boundary element methods and describe the necessary steps for their numerical discretisation and solution.

The most simple boundary integral equation is of the form

$$\int_\Gamma g(\mathbf{x},\mathbf{y})\phi(\mathbf{y})ds(\mathbf{y}) = f(\mathbf{x}),\ \mathbf{x}\in\Gamma. \qquad (1)$$

The function $g(\mathbf{x},\mathbf{y})$ is a Green's function, $f$ is a given right-hand side and $\phi$ is an unknown surface density over the boundary $\Gamma$ of a bounded three dimensional domain $\Omega\subset\mathbb{R}^3$. To provide a concrete example. To compute the electrostatic capacity of an object $\Omega$ one solves the above equation with $f(\mathbf{x})$ the constant function 1 and $g(\mathbf{x},\mathbf{y}) = \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}$. The normalized capacity is then obtained as $c = \frac{1}{4\pi}\int_\Gamma \phi(\mathbf{x})ds(\mathbf{x})$. Many practical problems have a significantly more complex structure and can involve block systems of integral equations. Nevertheless, the fundamental structure of what Bempp-cl does is well described by the above problem and will be described in the following.

The first step is the **definition of the surface** $\Gamma$. Surfaces are represented in Bempp-cl as a triangulation into flat triangles (see ). The triangulation is internally represented as an array of node coordinates and an associated connectivity array of node indices that define each triangle. In this step also topology data is computed. In particular, for each triangle the neighboring triangles and the type of intersection (i.e. via joint node or edge) is computed.

Once a triangulation is given we need to define the necessary data structures for the discretisation. Bempp-cl uses a Galerkin discretisation.

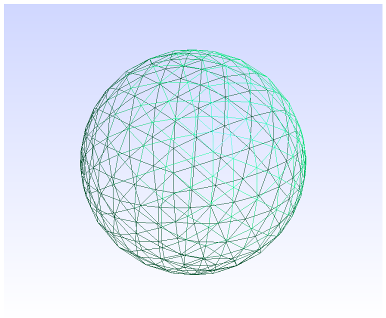

**Figure 1.** Discretisation of a sphere into flat surface triangles.

Consider (1). We represent the solution $\phi$ as $\phi = \sum_{j=}^N x_j\phi_j$, where the $\phi_i$ are basis functions of a **trial space**, defined in the most simple case as constant 1 on the triangle $\tau_j$ and 0 everywhere also. Several other types of discretisation spaces are available in Bempp. Moreover, we require a **test (or dual) space** for the discretisation. Let $\psi = \sum_{i=1}^N y_i\psi_i$, where each $\psi_i$ is a basis function of a space of test functions. For simplicity, here we will assume that the $\psi_i$ are also piecewise constant. The discrete representation of the above problem then takes the form

$$Ax = b$$

with

$$A_{ij} = \int_\Gamma \psi_i(\mathbf{x})\int_\Gamma g(\mathbf{x},\mathbf{y})\phi_j(\mathbf{x})ds(\mathbf{y})ds(\mathbf{x})$$

and $f_i = \int_\Gamma \psi_i(\mathbf{x})f(\mathbf{x})ds(\mathbf{x})$. In the case of piecewise constant trial and test functions the definition of $A_{ij}$ simplifies to $A_{ij} = \int_\Gamma\int_\Gamma g(\mathbf{x},\mathbf{y})ds(\mathbf{y})ds(\mathbf{x})$.

In Bempp-cl an operator definition consists of the type of the operator (e.g. Laplace single-layer in the above example), the definition of the trial and test spaces, and the definition of the range space. The range space is required for operator products and not relevant for the purpose of this paper. The function $f$ is represented as a **Grid function object**, which consists of either the dual representation in the forms of the integrals $b_i = \int_\Gamma \psi_i(\mathbf{x})f(\mathbf{x})ds(x)$ or directly through its coefficients $f_j$ in the representation $f = \sum_{j=1}^N f_j\phi_j$.

Once the grid, the space objects, and the operator are defined, the main computational step is performed, namely the **computation of the discrete matrix entries** $A_{ij}$. For pairs of triangles $\tau_i$ and $\tau_j$ that do not share a joint edge or vertex this is done through a simple numerical quadrature rule that approximates $A_{ij} \approx \sum_\ell \sum_q g(\mathbf{x}_\ell, \mathbf{y}_q)\psi_i(\mathbf{x}_\ell)\phi_j(\mathbf{y}_q)\omega_\ell\omega_q$, where the $\mathbf{x}_\ell$ and $\mathbf{y}_q$ are quadrature points in the corresponding triangles $\tau_i$ and $\tau_j$, and the values $\omega_i$ and $\omega_j$ are the quadrature weights. In the case that two triangles share a joint vertex/edge or the triangles $\tau_i$ and $\tau_j$ are identical, corresponding singularity adapted quadrature numerical quadrature rules are used that are based on singularity removing coordinate transformations.

The values $b_i$ of the right-hand side vector $b$ are similarly computed through a numerical quadrature rule.

In the final step, **Bempp solves the underlying linear system of equations** either through a direct LU decomposition or through iterative solvers such as Gmres. The solution can then be evaluated away from the surface $\Gamma$ through domain potential operators and exported in various formats for visualization.

In summary, to solve a boundary integral equation problem, the following steps are performed by Bempp

1) Import of the surface description as triangulation data
2) Definition of the trial space, test space and range space
3) Discretization into a matrix problem $Ax = b$
4) Evaluation of domain potential operators for visualization and post-processing

All of these steps are accelerated through the use of either Numba or OpenCL. In the following section we provide a high-level overview of the library and how these acceleration techniques are deployed before we deep dive into the design of the computational kernels.