Department: Head
Editor: Name, xxxx@email

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 optimized 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++ library. Over time, more and more functionality was moved into the Python interface, while computationally intensive routines and the main data structures remained in C++.

This proved a burden when efforts started to modernize the library to be able to make better use of single-instruction-multiple-data (SIMD) optimization 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 optimization 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 led to the choice of building a Python library based around OpenCL (using the PyOpenCL interface) and Numba.

At the end of 2019, we released the first version (0.1) of Bempp-cl [1]. 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 applications and many

 Published by the IEEE Computer Society

of our users are 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 provide a brief introduction to boundary element methods and describe the necessary steps for their numerical discretization and solution.

The most simple boundary integral equation is of the form

$$\int_\Gamma g(\boldsymbol{x},\boldsymbol{y})\phi(\boldsymbol{y})\,\mathrm{d}s_{\boldsymbol{y}} = f(\boldsymbol{x}), \quad \boldsymbol{x}\in\Gamma. \quad (1)$$

The function $g(\boldsymbol{x},\boldsymbol{y})$ is a Green's function, f is a given right-hand side, and ϕ is an unknown surface density over the boundary Γ of a bounded three dimensional domain $\Omega\subset\mathbb{R}^3$.

As a concrete example, we consider computing the electrostatic capacity of an object Ω. In this case, we solve the above equation with $f(\boldsymbol{x}) = 1$ and $g(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{4\pi|\boldsymbol{x}-\boldsymbol{y}|}$. Once ϕ has been found, the normalized capacity is then obtained using $c = \frac{1}{4\pi}\int_\Gamma \phi(\boldsymbol{x})\,\mathrm{d}s_{\boldsymbol{x}}$.

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 **discretization of the surface** Γ. Surfaces are represented in Bempp-cl as a triangulation into flat triangles (see Figure 1). 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, topology data is also computed: in particular, for each triangle, we compute the neighboring triangles and the type of intersection (i.e. are they connected by an edge or a vertex).

Once a triangulation is given we need to define the necessary data structures for the discretization. Bempp-cl uses a Galerkin discretization: we introduce set of basis functions ϕ_1 to ϕ_N, and define the **trial space** as the span of these function. We then approximate the solution ϕ of Equation (1) by $\phi_h = \sum_{j=1}^N \mathbf{x}_j\phi_j$, where $\mathbf{x}$ is a vector of coefficients. In the most simple case, we can define the function ϕ_j to be equal to 1 on the triangle τ_j and 0 everywhere else. Other spaces are commonly defined to be piecewise polynomials on each triangle.

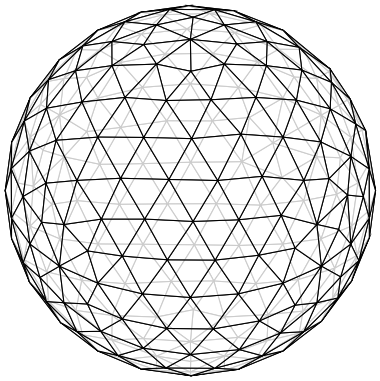

Figure 1. Discretization of a sphere into flat surface triangles.

To discretize Equation (1), we define a **test (or dual) space** in terms of a basis ψ_1 to ψ_N. The discrete representation of the above problem then takes the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

with

$$\mathbf{A}_{ij} = \int_\Gamma \psi_i(\boldsymbol{x}) \int_\Gamma g(\boldsymbol{x},\boldsymbol{y})\phi_j(\boldsymbol{x})\,\mathrm{d}s_{\boldsymbol{y}}\,\mathrm{d}s_{\boldsymbol{x}}$$
$$\mathbf{b}_i = \int_\Gamma \psi_i(\boldsymbol{x}) f(\boldsymbol{x})\,\mathrm{d}s_{\boldsymbol{x}}.$$

In the case of piecewise constant trial and test functions, the definition of $\mathbf{A}_{ij}$ simplifies to $\mathbf{A}_{ij} = \int_\Gamma\int_\Gamma g(\boldsymbol{x},\boldsymbol{y})\,\mathrm{d}s_{\boldsymbol{y}}\,\mathrm{d}s_{\boldsymbol{x}}$.

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 $\mathbf{b}_i = \int_\Gamma \psi_i(\boldsymbol{x})f(\boldsymbol{x})\,\mathrm{d}s_{\boldsymbol{x}}$ 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(s) are defined, the main computational step is performed, namely the **computation of the discrete matrix entries $\mathbf{A}_{ij}$**. For pairs of triangles τ_i and τ_j that do not share a joint edge or vertex this is done through a simple numerical quadrature rule that approximates $\mathbf{A}_{ij} \approx \sum_\ell \sum_q g(\boldsymbol{x}_\ell, \boldsymbol{y}_q)\psi_i(\boldsymbol{x}_\ell)\phi_j(\boldsymbol{y}_q)\omega_\ell\omega_q$, where the $\boldsymbol{x}_\ell$ and $\boldsymbol{y}_q$ are quadrature points in the corresponding triangles τ_i and τ_j, and the values ω_i and ω_j are the quadrature weights. In the case that two triangles share a joint vertex/edge or the triangles τ_i and τ_j are identical, corresponding singularity adapted quadrature numerical quadrature rules are used that are based on singularity removing coordinate transformations.

The values $\mathbf{b}_i$ 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 Γ 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 $\mathbf{Ax} = \mathbf{b}$.
4) Solution of the matrix problem by either a direct or iterative solver.
5) 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.

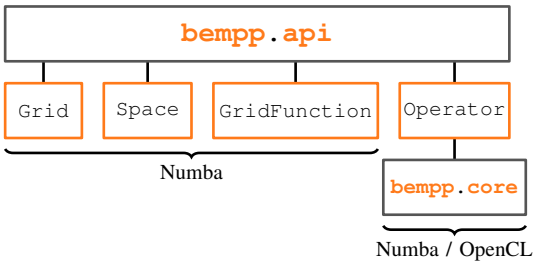

Figure 2. The layout of Bempp-cl with its computational backends.

A HIGH-LEVEL OVERVIEW OF BEMPP-CL

The main user-visible component of Bempp-cl is the module `bempp.api`, which defines all user interface functions and other high-level routines. In particular, it contains the definition of the main object types: `Grid`, `Space`, `GridFunction`, `Operator`. The computational backend routines are contained in the module `bempp.core`. Currently, we support a Numba and OpenCL backends. An overview of this structure is provided in Figure 2.

The main computational cost involved in solving a problem using boundary element methods is due to discretising the boundary integral operator on the left-hand side of Equation (1) to obtain the matrix $\mathbf{A}$: using dense methods, discretising an operator has quadratic complexity in terms of the number of surface triangles (although for larger problems this cost can be reduced through the use of hierarchical matrices or fast multipole methods). Great care is therefore taken to ensure that the operator assembly routines perform as highly as possible.

Once a user has defined in operator using Bempp-cl, the discretization can be computed by calling the `weak_form` method. Upon calling this method, a regular integrator will be used to assemble all the interactions between non-adjacent elements, and a singular integrator will be used to compute the interactions between adjacent triangles (if the trial and test spaces are defined on different grids, this second integrator is not needed). Depending on the user's preferences, these integrators will internally use computational routines defined using either Numba or OpenCL.

For OpenCL assembly, the code checks additional parameters, such as the default vector

length for SIMD operators (e.g., 4 for double precision and 8 for single precision in Intel AVX2, or 1 if a GPU is used), andwhether the discretization should proceed in single or double precision. The OpenCL kernel is then compiled for the underlying compute device using PyOpenCL and executed. If the computational backend is Numba, the call is forwarded to the corresponding Numba discretization routines and executed.

For simple piecewise constant function spaces or other spaces, where the support of each basis function is localized to a single triangle, only one call to the computational routines is necessary. If the support of basis functions is larger than a single triangle, different threads may need to sum into the same global degree of freedom.

Outside the operator discretization, Numba is used in the following contexts:

- Computing the grid topology: this involves iterating through the grid to compute the neighbour relataionships between triangles.
- Definition of local-to-global maps for function spaces: again, this requires traversal through the grid and assigning relationships between global and local indices.
- Grid functions: a right-hand side function f can be defined as a Python callable. This is just-in-time compiled via Numba and then the product with the corresponding basis functions is integrated in each triangle via numerical quadrature, again via Numba accelerated routines.
- Computing sparse matrices.

As the cost of each of these processes is in general much smaller than the cost of operator discretization, these can be permformed using Numba without any need to consider the use of OpenCL for potential further speed up.

ASSEMBLING BOUNDARY INTEGRAL OPERATORS WITH OPENCL

In this section, we discuss in more detail the assembly of boundary integral operators with OpenCL and how we integrated this into our Python workflow. We start with a brief introduction to OpenCL and then dive into how we use OpenCL as part of Bempp-cl.

What is OpenCL?

OpenCL (https://www.khronos.org/opencl/) is a heterogeneous compute standard for CPUs, GPUs, FPGAs, and other types of devices that provide conformant drivers. At its core, OpenCL executes compute kernels that can be written in C99, or (more recently) in C++. The current version of OpenCL is 3.0, though the most widely implemented standard is OpenCL 1.2, which Bempp-cl uses.

OpenCL splits computational tasks into work-items, each of which represents a single execution of a compute kernel. Work items are grouped together into work groups, which share local memory. Barrier synchronization is only allowed within a work group. All work items are uniquely indexed by a one, two, or three dimensional index space, called NDRange. Kernels are launched onto a compute device (e.g., a CPU or GPU) from a host device. OpenCL allows kernels to be loaded as strings and compiled on the fly for a given device, making it well suited for launching from high-productivity languages.

To launch an OpenCL kernel the user must first initialize buffers on the compute device, then copy any relevant data from host to the compute device. A kernel string can then be loaded and just-in-time compiled for the device. The kernel is then run, and the results can be copied back to the host.

OpenCL has very good support for vectorized operations: it provides vector data types and defines a number of standard operations for these vector types. For example, the type `double4` will allow four double values to be held in a SIMD register. This makes it easy to explicitly target modern SIMD execution in a portable way while avoiding difficult compiler intrinsics and keeping kernel code readable.

Python has excellent OpenCL support through the PyOpenCL library by Andreas Kloeckner [2]. PyOpenCL automates much of the initialization of the OpenCL environment and makes it easy to create buffers and launch OpenCL kernels from Python.

OpenCL Assembly in Bempp-cl

Bempp-cl has OpenCL kernels for all its boundary operators. All operators have the same interface and are launched in the same way. In the

```
#include "bempp_base_types.h"
#include "bempp_helpers.h"
#include "bempp_spaces.h"
#include "kernels.h"

__kernel void kernel_function(
    __global uint* testIndices, __global uint* trialIndices,
    __global int* testNormalSigns, __global int* trialNormalSigns,
    __global REALTYPE* testGrid, __global REALTYPE* trialGrid,
    __global uint* testConnectivity, __global uint* trialConnectivity,
    __global uint* testLocal2Global, __global uint* trialLocal2Global,
    __global REALTYPE* testLocalMultipliers,
    __global REALTYPE* trialLocalMultipliers,
    __constant REALTYPE* quadPoints, __constant REALTYPE* quadWeights,
    __global REALTYPE* globalResult,
    __global REALTYPE* kernel_parameters,
    int nTest, int nTrial, char gridsAreDisjoint)
{
  /* */
}
```

Figure 3. Definition of the OpenCL compute kernel for scalar integral equations.

first step, the relevant data will need to be copied to the compute device. This data consists of:

- Test and trial indices of triangles over which to be integrated.
- Signs of the normal directions for the spaces.
- Test and trial grid as flat floating point array, defining each triangle through nine floating point numbers, specifying the (x, y, z) coordinates of each of the three nodes of a triangle.
- Test and trial connectivity information that stores the indices of each node for each triangle.
- Test and trial mappings of local triangle degrees of freedom to global degrees of freedom.
- Test and trial basis function multipliers for each triangle.
- Quadrature points and quadrature weights.
- A buffer that contains the global assembled matrix.
- Additional kernel parameters, such as the wavenumber for Helmholtz problems.
- Number of test and trial degrees of freedom.
- A single byte that is set to one if the test and trial grids are disjoint.

The actual kernel definition is shown in Figure 3. Before the kernel can be launched, it needs to be configured and just-in-time compiled. Kernel configuration happens through C-style preprocessor definitions that are passed through the just-in-time compiler. These include the names of the test and trial space, the name of the function that evaluates the integral kernel, choosing between single and double precision types, and for SIMD enhanced kernel the vector length of the SIMD types. For example, in Figure 3 all floating point types have the name `REALTYPE`. This is substituted with either *float* or *double* during just-in-time compilation.

Each work-item computes via numerical quadrature all interactions via the integral kernel of basis functions on the trial element with basis functions on the test element. Before summing the result into the global result buffer, the kernel checks via the connectivity information if the test and trial triangle are adjoint or identical. For this it simply checks if at at least one of the node indices of the test triangle is identical to one of the node indices of the trial triangle. If this is true and the grids are not disjoint, the result of the kernel is discarded and not summed back into the global result buffer. The reason is that for adjoint triangles a separate singular quadrature rule is used. The effect is that a few work-items do work that is discarded. However, in a grid with N elements the number of triangle pairs requiring a singular quadrature rule is $\mathcal{O}(N)$, while the total number of interactions is N^2. Hence, only a tiny fraction of work-items are discarded.

SIMD optimized kernels

The above description was valid for kernels that do not take advantage of CPU SIMD operations are kernels that run directly on a GPU device. For SIMD optimization on CPUs we proceed slightly differently. The corresponding kernel in principle works as described above, but batches together the interaction on one test triangle with X trial triangles, where X is either 4, 8, or 16, depending on the number of available SIMD lanes. This strategy allows us to optimize almost all floating point operations within a kernel run for SIMD operation. If the number of trial elements is not divisible by 4, 8, or 16, then the few remaining trial elements are assembled with the standard-non vectorized kernel.

Each kernel definition is stored in two variants, one with the ending *_novec.cl* and another one with the ending *_vec.cl*. The vectorized variant is configured via preprocessor directives for the desired number of vector lanes. Having to develop two OpenCL kernel codes for each operator is a certain amount of overhead. If we write a new kernel, we first implement the non-vectorized version. Then, with the help of preprocessor directives and a number of helper functions that do the actual implementation of operations depending on whether the kernel is vectorized or not, it is usually a matter of an hour or two to convert the non-vectorized kernel into a vectorized version.

Alternatively, some CPU OpenCL runtime environments have the option to try and auto-vectorize kernels by batching together work items on SIMD lanes, similarly to what we do manually. In our experience this works well for very simple kernels but often fails for more complex OpenCL kernels. This is why we decided to implement this strategy manually.

A completely different SIMD strategy is based on batching together quadrature evaluations within a single test/triangle pair. There are two disadvantages to this approach. First, it only works well if the number of quadrature points is a multiple of the available SIMD lanes. Second, other operations such as the geometry calculations for each element then can not be SIMD optimized as these are only performed once per test/triangle pair.

Assembling the singular part of integral operators

The assembly of the singular part of an integral operator works a bit differently. Remember that the singular part consists of triangle parts which are adjacent to each other or identical. There are $\mathcal{O}(N)$ of such pairs. We are using fully numerical quadrature rules for these integrals that are based on subdividing the four-dimensional integration domain and using transformation techniques to remove the singularities. This gives highly accurate evaluations of these integration pairs but requires a large number (typically over 1000) quadrature points per triangle pair. Hence, we create one workgroup for each triangle pair and inside this workgroup have a number of work-items that are evaluating the quadrature rules and then at the end sum up the result together. Depending on how two triangles are in relationship to each other different types of singular quadrature rules are needed. We solve this pre-loading all possible quadrature rules onto the device and also store for each triangle pair an index into the required quadrature rule so that the kernel function can select the correct rule to evaluate. For the singular quadrature rules we did not implement separate SIMD optimized kernels as the proposed implementation is already highly efficient and requires only a fraction of the computational time of the regular quadrature rules described above. At the end, the singular integral values are either summed into the overall result matrix, or if desired by the user, stored as separate sparse matrix.

NUMBA ASSEMBLY OF INTEGRAL OPERATORS

The main focus of Numba within Bempp-cl is to provide accelerated implementations for linear complexity routines, such as grid iterations, integration of functions over grids or assembly of certain sparse matrices. However, we do also provide a fall-back implementation of the OpenCL dense operators in Numba. Here, we use classic loop parallelism. Each loop iteration is the assembly of one test element with all trial elements. Within each loop we try to optimize for auto-vectorization by linearly passing through the data in memory order for the individual operations. However, a much smaller fraction of operations

is SIMD optimized due to the lack of targeted SIMD constructs in Numba.

REFERENCES

1. T. Betcke and M. W. Scroggs, “Bempp-cl: A fast Python based just-in-time compiling boundary element library,” *Journal of Open Source Software*, vol. 6, no. 59, p. 2879, 2021. DOI: 10.21105/joss.02879
2. A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation,” *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012. DOI: 10.1016/j.parco.2011.09.001