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 CPUs with SIMD optimization. In this paper, we 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 [7]. Over time, more and more functionality was moved into the Python interface, while computationally intensive routines and the main data structures remained in C++. At the end of 2019, we completed the main steps of a full rewrite of Bempp and released the first version (0.1) of Bempp-cl. This was followed later in 2020 by version 0.2, the first release that we considered feature complete and mature for application use [1]. Since then we have used Bempp-cl in a number of practical applications and many of our users are migrating to it from the old C++ based Bempp. In this article, we discuss the motivation for the rewrite and restructure of Bempp and the reasoning behind the design choices we made when writing Bempp-cl.

The three language problem

The original BEM++ was troubled by what is often called the two language problem. It is common for programming language to be either easy for humans to write (e.g., Python) or easy for computers to run and achieve high performance (e.g., C++). It is not common, however, for a

 Published by the IEEE Computer Society

language to do both of these. Due to this, it is common in scientific computing libraries to write a library in a fast low-level language such as C++, while providing a user interface in a higher-level language such as Python.

This was the model followed by BEM++, but the problem would perhaps better be described as a three language problem: as well as the Python and C++ code contained in libraries like this, it is common to also include a significant amount of code in a third interfacing language. In the case of BEM++, this third language was first Swig and later Cython, an extension of Python with C data types that can be compiled to include functionality from C++ libraries.

Due to the three languages involved, making changes to the library would often mean having to duplicate changes in three places, with many class structures duplicated in all three languages. This made seemingly simple changes into onerous tasks, and provided a barrier to new members of the community looking to contribute to the open-source project.

Delegating computations with PyOpenCL

Prompted by a desire to simplify the library as well as to be able to run on a wide range of CPU and GPU devices, we began a full rewrite in 2018, which led to Bempp-cl. The aims of this rewrite were to support explicit single-instruction-multiple-data (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.

In addition to the performance benefits, the library redesign has greatly improved the issues related to the three language problem. Both OpenCL and Numba are used to compile functions. Each function is provided all the data it needs as inputs, so there is no need to duplicate any class structure outside Python. The need for a third interfacing language is removed, as the interfacing to the OpenCL kernels is handled by PyOpenCL.

Boundary element methods are particularly suited to this library model, as the main performance critical task is the computation of discrete operators. Once fast kernels for this have been implemented, the remaining functionality of the library can be written entirely in Python without any significant decrease in performance.

We begin this article by giving an overview of the boundary element method, and looking at how Bempp can be used to implement such problems. Following this, we discuss the implementation of boundary element kernels using OpenCL in more detail, and provide a number of performance benchmarks on different compute devices, including CPUs and Nvidia GPUs. We conclude with some thoughts on the advantages and disadvantages of OpenCL and Numba.

BOUNDARY ELEMENT METHODS WITH BEMPP

In this section, we provide a brief introduction to boundary element methods (BEM) 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 or even coupling with finite element (FEM) codes. Nevertheless, the fundamental structure of what Bempp-cl does is well described using the above simple problem.

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

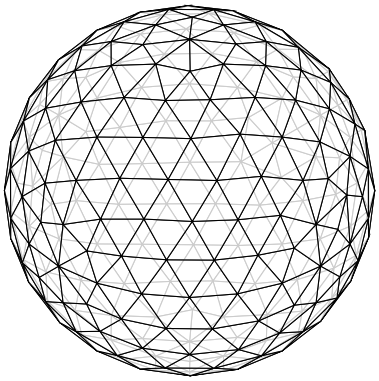

Figure 1. Discretization of a sphere into flat surface triangles.

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, see Figure 2).

Once a triangulation is given we need to define the necessary data structures for the discretization. Bempp-cl uses a Galerkin discretization: we introduce a set of basis functions ϕ_1 to ϕ_N, and define the **trial space** as the span of these functions. 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.

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

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

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

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 spaces, 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 such as a symmetric Gauss rule for triangles. In the case that two triangles share a joint vertex/edge or the triangles τ_i and τ_j are identical (see Figure 2), corresponding singular quadrature rules are used that are based on singularity-removing coordinate transformations [2].

The values $\mathbf{b}_i$ of the right-hand side vector $\mathbf{b}$ are similarly computed through a numerical quadrature rule.

In the final step, **Bempp-cl solves the underlying linear system of equations** either through a direct LU decomposition or through iterative solvers. 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-cl:

1) Import of the surface description as triangulation data.
2) Definition of function spaces and relevant operators.
3) Discretization into a matrix problem $\mathbf{A}\mathbf{x} = \mathbf{b}$.
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 dive into the design of the computational kernels.

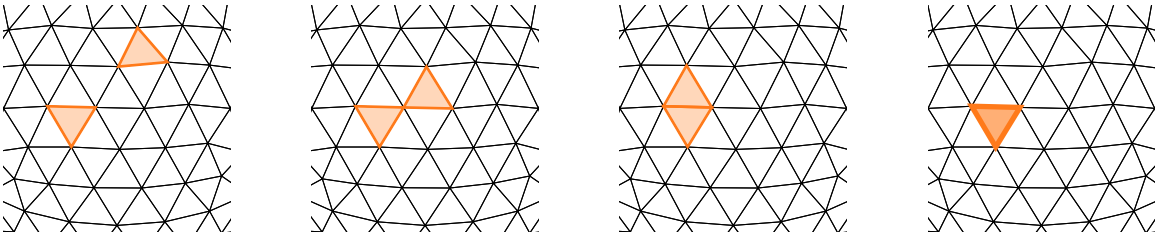

Figure 2. The four types of intersection of two triangles: the triangles can be (left to right) not neighbours, neighbours adjacent via a vertex, neighbours adjacent via an edge, or the same triangle. In the first case, standard quadrature is used. In the other three cases, singular quadrature rules must be used.

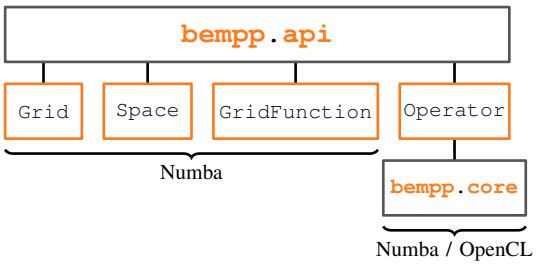

Figure 3. The layout of Bempp-cl with its computational backends.

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 definitions of the main object types: `Grid`, `Space`, `GridFunction`, and `Operator`. The computational backend routines are contained in the module **bempp.core**. Currently, we support Numba and OpenCL backends. An overview of this structure is provided in Figure 3.

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. For larger problems, this cost can be reduced through approximation techniques, such as hierarchical (H-) matrices or fast multipole methods (FMM) with log-linear or even linear complexity. The price of the improved complexity is significantly more involved data structures and additional approximation errors. Bempp-cl provides interfaces to ExaFMM [8] for large problems. Here, we focus on dense discretisation that is natively implemented in Bempp-cl and is suitable for medium sized problems up to a few ten thousand elements, depending on available memory. Great care needs to be taken to ensure that the quadratic complexity operator assembly routines perform as efficiently as possible.

Once a user has defined an 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 AVX, or 1 if a GPU is used), and whether 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 relationships 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 matrix entries, such as for mass matrices that are required to translate between representations of grid functions through basis coefficients or projections, or when we want to evaluate operator products.

As the cost of each of these processes is in general much smaller than the cost of operator discretization, these can be performed 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 [5] 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 OpenCL C, which is based on C99, or (more recently) in C++, with some restrictions on the allowed operations. 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 provide relevant data as buffers, which are transferred from the host to the corresponding 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 [3]. 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 first step, the relevant data will need to be copied to the compute device. This data consists of:

- Test and trial indices denoting the triangles over which to be integrated.
- Signs of the normal directions for the spaces.
- Test and trial grids as flat floating point arrays, defining each triangle through nine floating point numbers, specifying the (x, y, z) coordinates of each of the three nodes of a triangle.

```
#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 4. Definition of the OpenCL compute kernel for scalar integral equations.

- Test and trial connectivity, which are lists of node indices that define the corresponding triangles of the test and trial grid.
- Test and trial mappings of local triangle degrees of freedom to global degrees of freedom.
- Test and trial basis function multipliers for each triangle, which are triangle dependent prefactors needed for certain function spaces (e.g., in electromagnetics).
- Quadrature points and quadrature weights.
- A buffer that contains the global assembled matrix.
- Additional kernel parameters, such as the wavenumber for Helmholtz problems.
- The number of test and trial degrees of freedom.
- A single byte that is set to one if the test and trial grids are disjoint.

An example kernel definition is shown in Figure 4. 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 Green's function, whether we are using single or double precision types, and (for SIMD enhanced kernels) the vector length of the SIMD types. For example, in Figure 4 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 (using numerical quadrature) all interactions 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 triangles are adjacent or identical (see Figure 2). To do this, it simply checks if at at least one of the node indices of the test triangle is equal 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: for these triangles, separate singular quadrature rules need to be 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 triangle interactions is N^2. Hence, only a tiny fraction of work-items are discarded.

SIMD optimized kernels

When we are running on a CPU and want to take advantage of available SIMD optimizations, we need to make a few modifications to our approach. The corresponding kernel works similarly to what is described above, but we compute a batch of interactions between one

test triangle and 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 creates a certain amount of overhead, but once we have implemented 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 only a matter of an hour or two to convert the non-vectorized kernel into a vectorized version.

Alternatively, some CPU OpenCL runtime environments can (optionally) try and auto-vectorize kernels by batching together work-items on SIMD lanes, similar 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 could be taken by batching together quadrature evaluations within a single test/trial 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 cannot be SIMD optimized as these are only performed once per test/trial 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 (the three later cases in Figure 2): there are $\mathcal{O}(N)$ 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 evaluation of these integration pairs but requires a large number (typically over 1000) quadrature points per triangle pair.

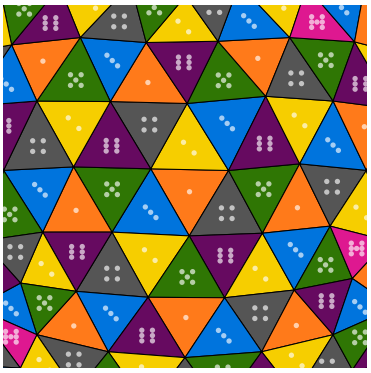

Figure 5. The triangles in this mesh have been colored (using a greedy algorithm) so that no two neighboring triangles are the same color. Sets of triangles of the same color can therefore be processed together as they are guaranteed to not be neighbors, so do not share any degrees of freedom. [The dots on each cell are included as a visual aid for anyone who prints this article in black and white.]

For this assembly, we create one work-group for each singular triangle pair. Inside this work-group, we have a number of work-items that evaluate the quadrature rules then sum up the results. Depending on how two triangles are related to each other, different types of singular quadrature rule are needed. We solve this by pre-loading all possible quadrature rules onto the device, and also store for each triangle pair an index pointing to 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.

Avoiding data races in the global assembly

Data races in global assembly routines are a problem whenever different triangles need to sum into the same global degree of freedom. To solve this we use standard coloring techniques to split up the computations into chunks that access different data regions. To this end, we define two triangles as neighbors if they share at least one global degree of freedom. Based on this relationship we run a simple greedy coloring algorithm in the initialization phase of a function space. An example coloring is shown in Figure 5.

The compute kernels can then be run color-by-color. OpenCL parallelizes over test elements and trial elements, and so we have to iterate over the product space of possible colour combinations. In Numba, we only parallelise over test elements so it is sufficient to iterate over all possible colors in the test space.

NUMBA ASSEMBLY OF INTEGRAL OPERATORS

The main focus of Numba within Bempp-cl is to provide accelerated implementations of routines with linear complexity, such as grid iterations, integration of functions over grids, or assembly of certain sparse matrices. However, we also provide a fall-back implementation of the OpenCL dense operator assembly in Numba.

With Numba, we use loop parallelism: each loop iteration is the assembly of one test triangle with all trial triangles. We then parallelize over the test triangles through a parallel for-loop. 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.

We stress that while Numba provides backends not only for CPU, but also for ROCm and CUDA, we currently only use the CPU component of Numba.

PERFORMANCE BENCHMARKS

In this section, we provide a number of performance benchmarks. The tests were all run on a Dell Precision 7740 Workstation Laptop with 64 GB RAM. Its CPU is an Intel i9-9980HK with a base clock of 2.4GHz and a burst clock of 5GHz. The CPU supports AVX, AVX2, and AVX-512. As a GPU we use an Nvidia Quadro RTX 3000 GPU. All benchmark tests were performed in Linux. For OpenCL on the GPU, we use the Nvidia GPU drivers and as CPU runtime we compare the open-source PoCL driver against the Intel CPU OpenCL runtime environment. All timing runs were repeated several times to make sure that the overhead from running the OpenCL and Numba just-in-time compilers did not skew the results. Though hardly noticeable by users, for smaller experiments the compilation phase typically takes longer than the actual computation.

Dense Operator assembly

We start by benchmarking the dense operator assembly. We assemble the matrix $\mathbf{A}$ defined by

$$\mathbf{A}_{ij} = \int_\Gamma \psi_i(\boldsymbol{x}) \int_\Gamma \frac{\phi_j(\boldsymbol{y})}{4\pi|\boldsymbol{x}-\boldsymbol{y}|} \,\mathrm{d}s_{\boldsymbol{y}} \,\mathrm{d}s_{\boldsymbol{x}},$$

with Γ being the unit sphere. For the basis functions ϕ_j and test functions ψ_i, we compare two cases: piecewise constant functions for both (P0 functions); and nodal, piecewise linear, and globally continuous functions for both (P1 functions). In the P0 case, each triangle is associated with just one piecewise constant function. In the P1 case, each triangle is associated with 3 linear basis functions, one for each node of the triangle.

We first compare the OpenCL CPU performance for the Intel OpenCL runtime and the PoCL OpenCL runtime driver. We run the tests in single precision and double precision. The native vector width for both drivers in single precision is 8, and in double precision is 4, corresponding to AVX instructions. In Bempp-cl, this means that we assemble in vectorized form one test triangle with 8 trial triangles in single precision, and with 4 trial triangles in double precision. Within the assembly, almost all floating point instructions are manually vectorized to take advantage of this. We should hence see up to a factor 2 speed-up between single and double precision assembly.

The timings for a grid with 32,768 elements are shown in Figure 6. We see the expected speed-up between single precision and double precision evaluation. It is interesting to note the difference between the Intel and the PoCL runtime environment. For P0 basis functions, the PoCL driver (gray bar) significantly outperforms the

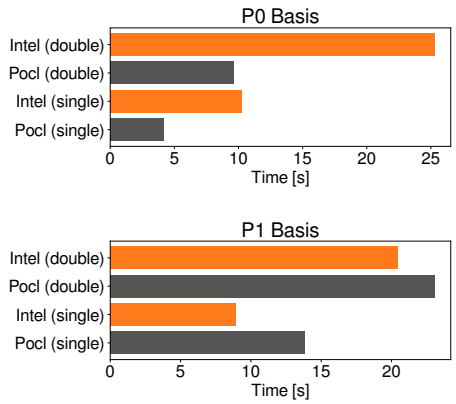

Figure 6. Comparison of the performance of PoCL and the Intel OpenCL runtime for the assembly of the Laplace single-layer boundary operator on a grid with 32,768 elements.

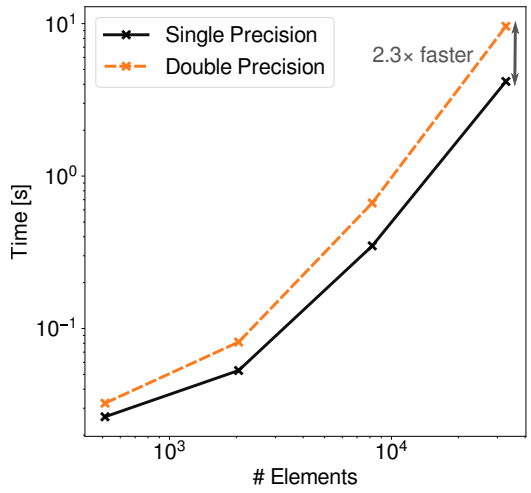

Figure 7. Comparison of the single-precision and double-precision performance for various grid sizes using PoCL and P0 basis functions.

Intel driver (orange bar) in both single and double precision. For P1 basis functions, however, the Intel driver gives better performance.

In Figure 7, we compare specifically the performance of single precision and double precision evaluation for the PoCL driver for various grid sizes in the case of a P0 basis. We can see that the speed-up for larger grid sizes is slightly more than just a factor of two. The final data point in this graph corresponds to the grid size used for Figure 6.

In Bempp-cl, we can easily switch between CPU Assembly, GPU Assembly, and Numba Assembly. Figure 8 shows a comparison between these modes for a grid with 2048 elements. The speed differences are striking. GPU assembly in single precision is a factor of two slower than CPU assembly, and much slower in double precision due to the limited double precision performance of our hardware, though we note that this is not an issue for double precision optimised data center accelerators. Numba is five times slower for single precision and still around three times slower for double precision than PoCL. The GPU behaviour can be explained by data transfer: while the GPU kernels themselves are extremely fast, data transfer over the bus severely limits performance. Even for medium sized problems, we have to transfer the data back to the main memory as GPU RAM is too limited to keep dense matrices with tens of thousands of rows and columns on the device.

	single	double
PoCL	0.05s	0.08s
Numba	0.25s	0.25s
GPU	0.11s	3.00s

Figure 8. Comparison of PoCL, Numba and GPU assembly for a grid with 2048 elements.

An alternative method is to compute the matrix-vector product (matvec) $\mathbf{Ax}$ on the device without first computing the dense matrix $\mathbf{A}$. This can be done by recomputing all matrix elements during each matvec calculation on-the-fly and not storing them. We have done experiments with this, and observed significant speed-ups compared to CPU evaluation as we now only need to transfer single vectors over the bus. For larger problems, however, it is not competitive compared to accelerated methods such as FMM, due to the quadratic complexity of direct evaluation of the matvec compared to linear complexity of FMM. For smaller problems, it is still practically better to just assemble the whole matrix and store it, as then matvecs are much faster for an iterative solver. Hence, we can conclude that there is only limited practical relevance for on-the-fly GPU evaluation of boundary integral operators. There are, however, very significant practical advantages of on-the-fly evaluation of domain potential operators for visualization and post-processing, as we

will see in the next section.

The Numba performance difference seen in Figure 8 is interesting. The main reason for this, we believe, is significantly lower usage of AVX vectorized instructions. We have taken care to optimize the Green's function evaluation for auto-vectorized evaluation in Numba, but the loops over integration points and other operations auto-vectorize very badly when just looping over all trial triangles for a single test triangle, as we currently do. We could tune our code for better auto-vectorization in Numba, but this would give little benefit as we have highly optimized hand-tuned OpenCL kernels already. We therefore recommend using Numba for operator assembly only as a fallback if no OpenCL runtime is available (this is less a judgement about Numba itself but about the limited optimisations we have done for Numba assembly routines).

Evaluating domain potentials for post-processing of electromagnetic problems

Once an integral equation is solved, one is usually interested in evaluating the solution not only at the boundary but also at points away from it. To do this, we evaluate the integral

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

for many points $\boldsymbol{x}$ away from the boundary Γ. For example, if we want to visualize a solution, we take the points $\boldsymbol{x}$ to be a regular grid of points.

Typically, we want to do only a small number of potential evaluations at the end of a calculation. Discretising this operation into a dense matrix and then evaluating the dense matrix-vector product is not practical for larger sizes. For very large problems with hundreds of thousands of elements, we use FMM or other accelerated approximate methods. For moderately sized problems with a few ten thousand elements up to around a hundred thousand elements (depending on the problem at hand), direct evaluation of this integral for every point $\boldsymbol{x}$ is highly efficient. We won't go into the details of the corresponding OpenCL kernels here, but we show some results that demonstrate the relative performance on CPUs and GPUs.

For the dense assembly of boundary integral operators, the performance was limited by the bus transfer of the dense matrix. For the evaluation of domain potentials, however, we only need to transfer to the device the vector of coefficients of the basis functions for ϕ, and then transfer back to the host the values at the points $\boldsymbol{x}$.

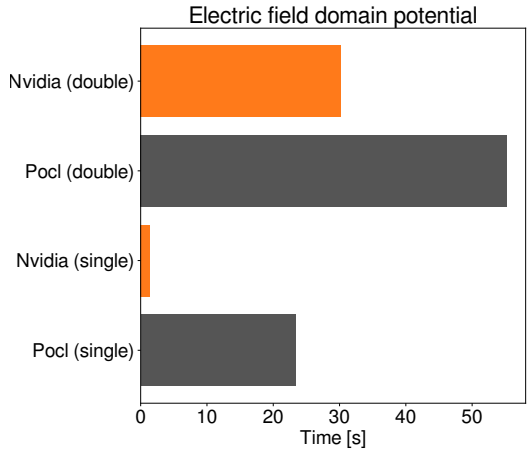

Figure 9. Evaluation of an electric field potential operator on CPU via PoCL vs Nvidia GPU

In this section, we consider the evaluation of the electric potential operator, defined by

$$\begin{aligned}(\mathcal{E}\mathbf{p})(\boldsymbol{x}) = \mathrm{i}k \int_\Gamma \mathbf{p}(\boldsymbol{y}) g(\boldsymbol{x}, \boldsymbol{y}) \, \mathrm{d}s_{\boldsymbol{y}} \\ - \frac{1}{\mathrm{i}k} \nabla_{\boldsymbol{x}} \int_\Gamma \operatorname{div} \mathbf{p}(\boldsymbol{y}) g(\boldsymbol{x}, \boldsymbol{y}) \, \mathrm{d}s_{\boldsymbol{y}}, \quad (2)\end{aligned}$$

where $g(\boldsymbol{x}, \boldsymbol{y}) = \frac{\mathrm{e}^{\mathrm{i}k|\boldsymbol{x}-\boldsymbol{y}|}}{4\pi|\boldsymbol{x}-\boldsymbol{y}|}$ is the Helmholtz Green's function, and k the wavenumber of the problem. The function $\mathbf{p} : \mathbb{R}^3 \to \mathbb{R}^3$ is a vector-valued function, leading to an overall vector solution at each point $\boldsymbol{x}$. The implementation of electromagnetic problems in Bempp is covered in detail in [6]. For these experiments, we again use a grid with 32,768 triangles but this time RWG (Rao–Wilton–Glisson) edge-based basis functions are used. For the potential evaluation, we use 50,000 random evaluation points in the exterior of the unit sphere, and as wavenumber use $k = 1.0$.

In Figure 9, we compare the performance of GPU evaluation with that of the PoCL CPU driver. In single-precision, the GPU significantly outperforms the CPU; and even in double-precision, the Nvidia Quadro RTX GPU is faster than the 8-core CPU, even though its hardware is not optimised for fast double precision operations.

SUMMARY

With Bempp-cl, we have created a Python library that achieves high-performance through use of modern just-in-time compilation technologies. Bempp-cl mixes Numba evaluation for less compute intensive linear complexity loops and sparse matrix generation with highly optimized OpenCL kernels for computationally sensitive dense matrix assembly routines. Basing development on Python instead of classical C/C++ makes it very easy to adapt the library and integrate with other libraries with Python interfaces, such as the recent work integrating Bempp-cl and the ExaFMM library to build electrostatic virus-scale simulations [8].

Strict separation of the computational backend from the interface layer in the library also makes it easy to integrate further backends, allowing us to remain up to date with future compute models. OpenCL itself has proved a valuable choice for the purpose of this library, as it allows us to run CPU and GPU optimized kernels with very little code overhead, and allows the user to easily move between CPU- and GPU-based compute devices with a simple parameter change. In this article, we demonstrate Nvidia benchmarks: the same benchmarking code could be used to run on AMD or Intel GPUs.

A disadvantage of our approach is that using OpenCL kernels introduces a second language (C99) to the library. Using Numba throughout would give a much more native Python experience, but, while Numba is constantly improving, it is currently difficult to achieve optimal performance for complex operations. OpenCL really shines here, as it makes explicit SIMD operations very easy through dedicated constructs. Moreover, OpenCL kernels are completely stack/register based functions, allowing much better compiler optimisations while Numba needs to create every object dynamically, even for very small arrays for objects such as coordinate vectors. We need to stress that we have performed very few optimisations specific to Numba, while significant optimisation has gone into the OpenCL codes. It is therefore well possible that the performance gap between Numba and OpenCL can be significantly reduced. But from other projects our own anecdotal experience is that the more Numba is optimised, the less Pythonic and more C-like Numba functions look. So while Numba is a very powerful tool, it requires its own techniques for optimisation, different from standard Python code.

Another important consideration with respect to Python and just-in-time acceleration is the type of algorithms that benefit. For the dense assembly of integral operators, we have very simple data structures that can easily be passed to compute kernels. More complex data structures with larger mixture of data movement operations and computations (e.g., tree-based algorithms), are much harder to accelerate since the Python layer imposes limits here on the performance.

Overall, with the model of mixed Python/OpenCL/Numba development, we have created a flexible and easy to extend platform for integral equation computations. The initial re-implementation efforts by abandoning our old C++ code base are paying off, as they allow us to develop new features in a far simpler environment without sacrificing performance. Strict separation of compute backends and higher level routines makes it easy for us to integrate other accelerator techniques in the future with little code changes, and to react to new trends in heterogeneous computing.

The current focus of further developments is on letting Bempp-cl take advantage of cluster computing by integrating the `mpi4py` MPI bindings for Python. We have also made big steps forward for large problems by creating a black-box FMM interface that currently interfaces to ExaFMM: this has allowed us to solve problems with 10 million elements on a single workstation. We believe that Python-focused development (with some native routines in lower-level languages) is a scalable model and are aiming to exploit this scalability further as we move from single workstation computations to large cluster problems.

Acknowledgements

The work of Timo Betcke is supported by Engineering and Physical Sciences Research Council Grant EP/V001531/1.

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. S. Erichsen, and A. Sauter, “Efficient automatic quadrature in 3-d Galerkin BEM,” *Computer Methods in Applied Mechanics and Engineering*, vol. 157, no. 3, pp. 215–224, 1998. DOI: 10.1016/S0045-7825(97)00236-3.
3. 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.
4. S. K. Lam, A. Pitrou, and S. Seibert, “Numba: a LLVM-based Python JIT compiler,” *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015. DOI: 10.1145/2833157.2833162.
5. OpenCL, https://www.khronos.org/opencl/.
6. M. W. Scroggs, T. Betcke, E. Burman, Wojciech Śmigaj, and E. van ’t Wout, “Software frameworks for integral equations in electromagnetic scattering based on Caderón identities” *Computers & Mathematics with Applications*, vol. 74, no. 11, pp. 2897–2914, 2017. DOI: 10.1016/j.camwa.2017.07.049.
7. W. Śmigaj, T. Betcke, S. Arridge, J. Phillips, and M. Schweiger, “Solving boundary integral problems with BEM++,” *ADM Transactions on Mathematical Software*, vol. 41, no. 2, article 6, 2015. DOI: 10.1145/2590830
8. T. Wang, C. D. Cooper, T. Betcke, and L. A. Barba, “High-productivity, high-performance workflow for virus-scale electrostatic simulations with Bempp-ExaFMM,” 2021. arXiv (preprint): 2103.01048.