Responses to the Associate Editor and Reviewers

We are grateful for the careful comments by the reviewers and editor. Below you find responses to questions raised by the associate editior and reviewers. For smaller remarks regarding typos, etc., we have done corrections directly in the text without providing responses.

Responses to the Associate Editor comments

The technique described by the authors seems to entail a substantial duplication of code that might be avoidable using certain techniques. What trade-offs led the authors to their approach?

The main code duplication is within OpenCL kernels. Some of this was mitigated through inline functions defined in headers and significant use of define statements. But this does not avoid all code duplication. At the end, the decision was that this was a price worth paying for the advantage of being able to encapsulate expensive computational routines in fast OpenCL kernels.

The redevelopment of kernels using Numba came after OpenCL. This was mainly driven out of curiosity of how well Numba could perform but also to give users an alternative who don't have a well working OpenCL stack on their system (especially Windows and Mac)

Modern FEM codes tend to employ atomic operations instead of graph coloring for improved performance. (Atomic FP operations, also in 64 bit, can be realized via the classical compare-and-swap technique.)

We are using the OpenCL 1.2 standard, which supports a compare and swap operation only for 32 bit floats. Otherwise, I agree that atomics are preferable.

The authors state that SIMD vectorization is not used for the evaluation of singular integrals, on account of the far field representing the lion's share of the work. Under FMM acceleration, near-field and far-field should take approximately equal time. What were the obstacles to applying the optimization in this setting?

In the preprint https://arxiv.org/pdf/2103.01048.pdf we discuss the combination of Bempp-cl and Exafmm for large Poisson-Boltzmann problems. This has figures on combined assembly time and solver times. Singular integrals were not a significant bottleneck here. We are currently preparing a more detailed paper about fast solver integration into Bempp, where this will be discussed more. Should for certain problem sizes the singular integrals become a bottleneck, we can either manually optimize or shift onto GPU with not much effort. One note though, we are not manually SIMD optimizing. But I believe that due to the structure of our singular integration routines some automatic vectorization potential may be exploited by the compiler.

The comment on lower FP64 throughput on Nvidia hardware should be removed or rephrased as it only applies to gaming-targeted consumer hardware.

A lot of our users don't have access to HPC Centre type accelerators. So this is a relevant observation for them. We have clarified in the text though that slow double precision performance is not an issue in data center accelerators.

The authors describe repeated host-device transfers as a particular performance bottleneck. These are likely avoidable if the data is kept on the device throughout. Why was this route not chosen? This question is particularly salient because it directly influences the authors' disqualification of GPUs for direct evaluation tasks.

The reason for this is problem sizes vs available memory in most accelerators. Dense discretisation require significant amounts of memory. Barring data centre type accelerators most cards don't have enough RAM to keep dense matrices on them for interesting problem sizes. Many practical computations do not just need a single matrix, but a number of matrices (e.g. transmission probelms), quickly eating up RAM on any but the most expensive accelerators.

But this is an option that is likely coming soon as part of our push to optimise Bempp-cl for cluster computing on HPC systems with MPI.

Numba is described as a fallback technology, but no particular scenario is given in which the fallback would be required. The installation instructions for PyOpenCL (https://documen.tician.de/pyopencl/misc.html) appear to contain instructions to deploy a full OpenCL stack on nearly any machine.

Numba has two uses in our library, one essential, and one fallback. The essential use is for all O(n) routines that involve grid iterations. Here, we make extensive use of Numba. For operator assembly it started as experiment to see how a simple Numba implementaton fares against OpenCL. The decision to keep this in the code was mainly for Mac users. On Mac we frequently encountered problems with users having crashes in POCL, and the Apple CPU OpenCL implementation being problematic (workgroup sizes of more than 1 item caused problems for us on CPU with the Apple OpenCL driver). We do not have Macs available to properly sort out these problems, and often users don't want to use Docker images as it makes it more difficult to modify the code and link with external libraries (it is certainly possible, but most users don't have this Docker knowledge). Hence, Mac users can use this fallback quite easily. On Windows this is only relevant to some extent as the Intel OpenCL CPU driver for Windows works well.

When the authors characterize OpenCL as a "second language", to what extent is this second language necessitated by the GPU as a "second environment"?

Good point. Personally, I like how libraries such as Numba-Cuda hide the complexity of the second device. But this certainly comes with its own drawbacks.

Roofline plots for cost/performance.

A dedicated paper on the dense algorithms for OpenCL is in planning. This will contain much more detailed performance comparisons than possible in the

available format and time for the special issue.

Responses to Reviewer 1

In the left column on page 6, what are "basis function multipliers" in lines 30-31?

Certain function spaces such as RWG spaces in Maxwell need additional multipliers that depend on the triangle. These are stored here.

In the left column on page 8, maybe an example of how "classic loop parallelism" works in line 33 would be useful. I was wondering how assembling the combination of one test element with all trial elements could still lead to linear complexity.

Each test triangle is assembled together with all trial triangles. We then paralleize over the test triangles. This has been clarified in the text.

In the right column on page 8, why is only AVX used in line 31? As far as I can see, the processor used for the benchmarks supports AVX512, as well.

We have used the default vector width from the OpenCL driver. AVX512 is neither in Intel nor in Pocl enabled by default as it can lead to slower performance than AVX2 due to the significant reduction in the CPU clock rate. Also, on AMD it is not supported at all. We have therefore decided to only present tests with AVX2. In practice, we noticed minor improvements with AVX512.

In the left column on page 11, I was surprised at the works "stack based functions" in lines 29-30. As far as I know, OpenCL assigned registers directly to variables and does not use a stack. The point still stands, of course, since registers are even more efficient than stack-based local variables, as long as there are enough of them available.

I think for the CPU this depends on the implementation and cannot automatically be assumed.

I suggest to include references to previous work on BEM with GPUs, e.g., the paper "GCA-H^2 matrix compression for electrostatic simulations" by S. Börm and S. Christophersen or the papers "Algorithmic patterns for H-matrices on many-core processors" and "A scalable H-matrix approach for the solution of boundary integral equations on multi-GPU clusters" by P. Zaspel.

These are excellent paper of which we are aware. We have not excluded these and other similar papers (e.g. the work by Rio Yokota's group on FMM and H-Matrices on GPUs) since the focus of this publication is on software design for dense matrix assembly with Python and OpenCL. We are working on publications related to the coupling of Bempp-cl and Exafmm, in which these references are more suitable.

In the left column on page 6, what happens if the global assembled matrix does not fit into, e.g., graphics memory when using a GPU in line 33? Is it possible to split the global matrix into submatrices that fit into graphics memory, and

then transfer them to main memory? Given the high computational complexity, would it be possible to overlap memory transfers and computation?

Right now, in this case a memory exception is created and returned to the Python interpreter. Submatrix assembly and swapping would be possible. However, efficient shadowing of memory transfer by computations requires that the computations are at least as long as the memory transfers. From the experiments we have done with simple Laplace operators this was not the case. Assembly is blazingly fast and memory transfer much slower. This may be different with more complex operators such as those from Maxwell.

The reason why we have not focused further on this is that there is not so much relevance for practical applications. On modern CPUs the assembly for everything that can feasibly done dense is fast enough. For anything larger we then want to use FMM or H/H^2 matrix techniques.

Responses to Reviewer 2

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." To me this statement casts a shadow on the interpretation of the benchmarking results since it may not be "fair" to compare highly optimized OpenCL kernels to less well-optimized Numba kernels. I do understand that Numba is meant to be a backup implementation in Bempp, but then the authors should be careful to put these results in context. I ask that they provide some additional discussion on the "fairness" of this comparison and how the results should be interpreted.

This is a fair point and we have emphasised this more in the text. On the last page we have added the following part:

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.

Since Numba provides support for both CUDA and ROCm, the authors should explicitly state that are using Numba CUDA in the abstract and introduction. It's clear from the context since they are using an NVIDIA GPU but they need to say this to avoid confusion.

Just as clarification. We are not using the GPU functionality of Numba. All our Numba code is CPU code. We have added a corresponding comment to the Numba Assembly section. We have considered a Numba Cuda backend. However, currently this would bring no advantage to the existing OpenCL backend for GPUs as they work on Nvidia and ROCm devices.

Having briefly experimented with PyOpenCL/OpenCL myself, I was surprised at the authors' characterization that using it was easy. I struggled to get it configured properly with my NVIDIA vendor driver, for example. Maybe the authors did not have this experience but since OpenCL has a general reputation for being harder to use than CUDA, it would be useful for the authors to acknowledge this and state whether their experience was similar or different.

There are two things here to distinguish. 1.) Installation difficulties of OpenCL. 2.) Complexity of using OpenCL. With regards to OpenCL installation. I am using a Linux laptop with the Nvidia close source drivers and never had any issues. The only small caveat is that if PyOpenCL is installed from conda, one needs to symlink the Nvidia icd from /etc/OpenCL/vendors to the corresponding /etc/ directory in the virtual environment for PyOpenCL to see it. Also, on Windows I did not encounter issues. Indeed, we have users that have used Bempp on Windows with an Nvidia card.

With regards to the complexity of OpenCL itself, here the big advantages of Python bindings come in. While the OpenCL C library is very verbose, the PyOpenCL bindings automate most of the work and allow to launch kernels with just a few lines of code. Since we only ever used OpenCL from Python the verbosity of the C interface to OpenCL was never an issue for us.

Debugging OpenCL can be more of a problem. But I am very used to debugging kernels with simple printf statements. Over time one gets quite proficient at it.

This is a more minor comment but since OpenCL was chosen for portability, I would have been very interested to see benchmarking results on AMD GPUs too. If this is future work it is worth mentioning.

We do not have access to modern AMD hardware. I have a workstation with two AMD Polaris generation GPUs. But my Nvidia card is a modern Quadro RTX 3000, which in all our tests significantly outperforms the older AMD Polaris generation. It would therefore be unfair to bring such comparison benchmarks here. We would only demonstrate the age difference between these GPU generations from different vendors.

We are working on a dedicated paper on the OpenCL algorithms which will contain much more detailed benchmarks on different cards with respect to their peak performance. The focus of the current paper was on demonstrating the potential of mixed Python/OpenCL development and less on detailed performance bencmarking of specific algorithms.

Responses to Reviewer 3

It needs to introduce the PDEs and range of applications at the beginning better, as well as more welcoming notation for the general reader. I was also hoping for more insight into the results (fraction of peak perf, fraction of time on host-to-device transfers, etc), and take-home messages about mistakes made and lessons learned.

I understand this sentiment. There is a strict page limit for this article. Within this limit we decided to focus on the software design and less on mathematical and application aspects.

With regards to more detailed performance measurements, we are preparing a more dedicated paper on the OpenCL algorithms which will have more detail on this than we have scope in this paper.

With regards to take-home message about lessons learned we again would love to talk much more than is possible to fit on a one page summary at the end. Should the referee wish I'd be happy to discuss this topic much more in person than is possible through the refereeing format. There are a number of aspects of our journey from C++ to Python that I could not discuss here, and the journey is continuing. While the development model that we use in the paper works very well for Bempp-cl, we encountered limits of the Python focused model for other types of algorithms that are worth discussing in their own rights (and we are currently planning a corresponding publication).