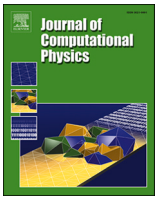
Journal of Computational Physics 547 (2026) 114549

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp



Hardware acceleration for HPS algorithms in two and three dimensions

Owen Melia [a,*], Daniel Fortunato [a,b], Jeremy Hoskins [c,d], Rebecca Willett [c,d,e,f]

[a] Center for Computational Mathematics, Flatiron Institute, New York, 10010, NY, USA
[b] Center for Computational Biology, Flatiron Institute, New York, 10010, NY, USA
[c] Computational and Applied Mathematics, Department of Statistics, University of Chicago, Chicago, 60637, IL, USA
[d] NSF-Simons National Institute for Theory and Mathematics in Biology, Chicago, 60611, IL, USA
[e] Data Science Institute, University of Chicago, Chicago, 60637, IL, USA
[f] Department of Computer Science, University of Chicago, Chicago, 60637, IL, USA

ARTICLE INFO

Keywords:
Partial differential equations
Fast direct solvers
Hardware acceleration
GPU
Numerical analysis

ABSTRACT

We provide a flexible, open-source framework for hardware acceleration, namely massively-parallel execution on general-purpose graphics processing units (GPUs), applied to the hierarchical Poincaré–Steklov (HPS) family of algorithms for building fast direct solvers for linear elliptic partial differential equations. To take full advantage of the power of hardware acceleration, we propose two variants of HPS algorithms to improve performance on two- and three-dimensional problems. In the two-dimensional setting, we introduce a novel recomputation strategy that minimizes costly data transfers to and from the GPU; in three dimensions, we modify and extend the adaptive discretization technique of Geldermans and Gillman [1] to greatly reduce peak memory usage. We provide an open-source implementation of these methods written in JAX, a high-level accelerated linear algebra package, which allows for the first integration of a high-order fast direct solver with automatic differentiation tools. We conclude with extensive numerical examples showing our methods are fast and accurate on two- and three-dimensional problems.

1. Introduction

Many problems in scientific computing require solving systems of linear, elliptic partial differential equations (PDEs). Such PDEs can accurately model a variety of physics, such as wave propagation, electrostatics, and diffusion phenomena. Because analytical solutions of these equations are often unknown, the task of computing numerical solutions has been an area of active research for hundreds of years. Today, there are a myriad of numerical solution methods available, and many are tailored to particular classes of equations or to specific use cases. We are most interested in designing methods for settings such as inverse or control problems, where the PDE implicitly defines some functional which, along with its gradient, is evaluated sequentially hundreds or thousands of times in the inner loop of an iterative algorithm.

In these settings, fast direct solvers [2] are a compelling choice. These solvers are able to rapidly compute a high-accuracy solution operator and can rapidly evaluate the solution given new data by applying the solution operator, often at the cost of a few

* Corresponding author.
E-mail addresses: omelia@flatironinstitute.org (O. Melia), dfortunato@flatironinstitute.org (D. Fortunato), jeremyhoskins@uchicago.edu (J. Hoskins), willett@g.uchicago.edu (R. Willett).

https://doi.org/10.1016/j.jcp.2025.114549
Received 25 March 2025; Received in revised form 13 November 2025; Accepted 24 November 2025
Available online 29 November 2025

matrix-vector multiplications. Fast direct solvers are also preferable for certain PDEs with oscillatory solutions [3], especially ones modeling wave propagation, as they do not incur a data-dependent iteration complexity cost associated with iterative solvers, which can be quite large [4].

Recently, scientific computing has undergone a paradigm shift with the advent of general-purpose hardware accelerators, such as GPUs. These hardware accelerators allow for massively parallel computation—they have thousands of processor cores on a single chip—but have strict memory constraints, a resource profile very different from standard multicore CPU architectures. This paper explores hardware acceleration of fast direct solvers and introduces new methods to facilitate this acceleration.

In particular, we focus on the hierarchical Poincaré–Steklov family of algorithms [3,5,6], a class of direct solution methods for variable-coefficient elliptic PDEs. These methods are characterized by a nested dissection approach combined with a high-order composite spectral discretization. We identify the algorithmic structure of these algorithms which makes them amenable to GPU acceleration and introduce new techniques for reducing the memory footprint of these methods. For two-dimensional problems, we introduce a novel recomputation strategy that minimizes data transfer between the GPU and host memory. In three dimensions, we use an adaptive discretization method, which greatly reduces the algorithm's peak memory complexity. Our numerical examples show these ideas are useful in challenging applied settings such as wave propagation, inverse problems, and molecular biology simulations.

We focus on solving linear, elliptic partial differential equations of the form

$$\mathcal{L}u(x) = f(x), \qquad x \in \Omega, \tag{1}$$

$$u(x) = g(x), \qquad x \in \partial\Omega. \tag{2}$$

In Eq. (1), $\mathcal{L}$ is a linear, elliptic, second-order partial differential operator with spatially-varying coefficient functions, and Ω is a square $\subset \mathbb{R}^2$ or cube $\subset \mathbb{R}^3$. Eq. (2) specifies Dirichlet boundary data, but our methods can also solve problems with Robin or Neumann boundary data. In these problems, we assume we can evaluate the differential operator $\mathcal{L}$, the source f, and the boundary data g at a set of discretization points of our choosing. We represent the solution u by its restriction to the same set of discretization points and rely on high-order polynomial interpolation to evaluate u away from the discretization points. In this paper, we refer to vectors with bold lowercase symbols such as $\boldsymbol{f}$ and matrices with bold uppercase symbols such as $\boldsymbol{A}$. We use x for the spatial variable, and when we want to indicate Cartesian coordinates, we use $(x_1, x_2) \in \mathbb{R}^2$ and $(x_1, x_2, x_3) \in \mathbb{R}^3$. We use the subscript u_n to denote the outward-pointing boundary normal derivative of a function, and we use Δ to denote the Laplace operator, the sum of second derivatives in each dimension.

1.1. Paper outline and contributions

In Section 2, we discuss related work, including algorithmic development for fast direct solvers and GPU-specific optimizations. In Section 3, we give an overview of the hierarchical Poincaré–Steklov method and discuss the potential for massively parallel implementations of the algorithm. In the rest of the paper, we make the following contributions:

- We optimize data transfer patterns to accelerate our method applied to two-dimensional problems (Section 4.1).
- To alleviate peak memory requirements in three-dimensional problems, we extend the two-dimensional adaptive method of [1] to three dimensions, develop the first adaptive 3D GPU-compatible HPS implementation, and provide numerical examples to demonstrate memory and accuracy tradeoffs (Section 4.2).
- We provide a range of numerical examples illustrating the application of our method, focusing on two settings: high-wavenumber scattering problems and the linearized Poisson–Boltzmann equation (Sections 5 and 6).
- We show our proposed algorithm and implementation can be combined easily with standard automatic differentiation software, which makes it particularly amenable to application in optimization, inverse problems, and machine learning contexts (Section 5).
- We make our JAX-based implementation publicly available at https://github.com/meliao/jaxhps.

2. Related work

HPS algorithms are built on two conceptual building blocks: composite high-order spectral collocation methods [7–9], and nested dissection of the computational domain [10]. Composite spectral collocation methods are those which separate the computational domain into a set of disjoint elements, and use a high-order spectral collocation scheme to represent the problem and solution separately on each element. Nested dissection methods break the original problem into a series of subproblems defined on a hierarchy of subdomains. The careful ordering of subproblems reduces the overall computational complexity by leveraging knowledge about properties of the solution, i.e. continuity of the solution and its derivative. Composite spectral collocation and hierarchical matrix decomposition ideas were combined in integral equation methods [11] for constant-coefficient PDEs.

Martinsson [5] first proposed combining these elements in a fast direct solver for variable-coefficient linear elliptic PDEs. The proposed scheme discretizes and merges Dirichlet-to-Neumann (DtN) operators. Gillman and Martinsson [6] proposed a compression scheme that leverages the structure of these DtN operators to build a solver with $O(n)$ computational complexity for n elements. To alleviate the instabilities observed when merging DtN operators for Helmholtz problems, Gillman et al. [3] proposed a scheme that merges impedance-to-impedance (ItI) operators instead. Further analysis for this scheme was provided in Beck et al. [12]. Modifications for three dimensions have been proposed, including Lucero Lorca et al. [13], Hao and Martinsson [14], Kump et al. [15], which all build solvers for three-dimensional Helmholtz problems. To alleviate memory and computational complexity, Lucero Lorca et al. [13] use an iterative method at the highest-level subproblems. In concurrent work to our own, [15] approach three-dimensional

problems with uniform discretizations using a hybrid GPU-CPU approach by combining the composite spectral collocation method with a two-level sparse direct solver. Fortunato et al. [16] use the ultraspherical spectral method to discretize triangular or quadrilateral mesh elements and compute solutions over polygonal domains by merging DtN operators. Fortunato [17] develops a variant of the HPS method which merges DtN and ItI operators to solve PDEs on unstructured meshes of smooth two-dimensional surfaces. Beams et al. [18] develop an implementation of the HPS algorithm targeting parallel shared-memory computer architectures.

There has been significant interest in the GPU acceleration of (low-order) iterative PDE solvers [19]. Other general-purpose packages, such as MFEM and libCEED, implement high-order iterative solvers with GPU acceleration [20,21]. Accelerating these algorithms often requires the rapid application of an extremely sparse system matrix. Applying GPU acceleration to direct solvers [22–24] requires different techniques; the literature has mostly focused on sparse direct solvers which do not employ a nested dissection method.

These sparse direct solvers often have much higher peak memory requirements and heterogeneous computation profiles when compared with iterative PDE solvers.

Other solvers have been designed directly for GPU acceleration. Yesypenko and Martinsson [25,26] designed a composite high-order spectral collocation method and associated sparse direct solver with highly heterogeneous computation patterns, which eases GPU acceleration. This method can solve variable-coefficient 2D problems very quickly; our method solves similar problems, but can also handle three-dimensional problems and interface with automatic differentiation. Developing a GPU-compatible implementation of the solver in Yesypenko and Martinsson [26], as well as the implementation of our method, has been greatly eased by the advent of high-performance hardware-accelerated linear algebra frameworks popularized by deep learning, such as PyTorch [27] and JAX [28]. These frameworks are highly efficient for batched linear algebra tasks and implement automatic differentiation capabilities. Both are high-level packages that sit on top of the XLA compiler [29], which compiles and launches optimized kernels that execute on general-purpose GPUs. There has also been work creating automatic differentiation compatible PDE solvers in JAX, tailored for problems such as synchrotron simulation [30], computational mechanics [31], and ordinary differential equations [32].

3. Introduction to HPS methods

In this section, we provide an overview of the HPS algorithms used in the paper, with a particular eye on their computational structure and the possibilities for GPU acceleration.[1] Full algorithms are available in Appendices A, B, C. We use different variants of this algorithm for merging different types of Poincaré–Steklov operators. A Poincaré–Steklov operator $T : g \mapsto h$ maps from one type of boundary data to another. Take, for example, a Dirichlet-to-Neumann (DtN) operator, which maps from Dirichlet data g on the boundary of Ω to Neumann data h on the same boundary:

$$g = u|_{\partial\Omega},$$
$$h = u_n|_{\partial\Omega},$$

where u satisfies Eq. (1). Another example commonly used is an impedance-to-impedance (ItI) operator, which maps "incoming" impedance data g to "outgoing" impedance data h [3]:

$$g = u_n + i\eta u|_{\partial\Omega},$$
$$h = u_n - i\eta u|_{\partial\Omega},$$

where u satisfies Eq. (1). These Poincaré–Steklov operators are linear operators, and we work with their discretization $\boldsymbol{T}$. Throughout the algorithm, we also work with $\boldsymbol{g}$ and $\boldsymbol{h}$, vectors of incoming and outgoing boundary data evaluated at a set of discretization points.

It is important to remember that because we are solving a linear partial differential equation, we can decompose the solution $u(x)$ into a particular solution $v(x)$ and homogeneous solution $w(x)$ where $u(x) = v(x) + w(x)$. The particular solution $v(x)$ satisfies

$$\begin{cases} \mathcal{L}v(x) = f(x), & x \in \Omega, \\ v(x) = 0, & x \in \partial\Omega, \end{cases}$$

and the homogeneous solution $w(x)$ satisfies

$$\begin{cases} \mathcal{L}w(x) = 0, & x \in \Omega, \\ w(x) = g(x), & x \in \partial\Omega. \end{cases}$$

3.1. Discretization via composite high-order spectral collocation

To numerically solve Eqs. (1) and (2), a discretization is needed to represent $\mathcal{L}$, f, and g in some finite-dimensional basis. HPS methods perform this discretization in two steps: a recursive partition of the domain Ω and a high-order spectral collocation scheme. The first step is to recursively partition Ω using a quadtree or octree structure down to a user-specified maximum depth L.

We use n_{leaves} to denote the number of patches at the finest level of the spatial partition. In this section, we consider uniform discretization trees, so each tree with depth L will have $n_{\text{leaves}} = 2^{dL}$ elements at the lowest level in dimension $d = 2, 3$. In Section 4.2, we consider more general discretization trees. At times, it will be useful to describe the progress of the algorithm using language to

[1] For an instructional introduction to these methods, we refer the interested reader to Martinsson [2], Gillman et al. [3], Martinsson [33].

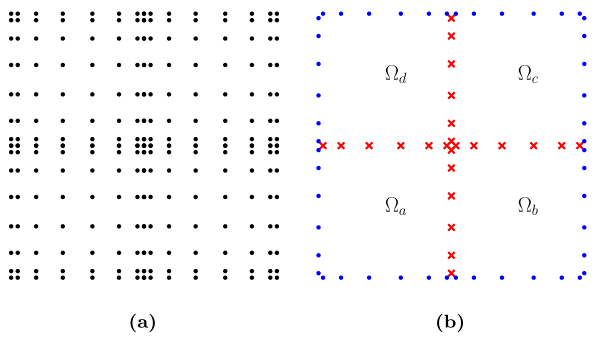


Fig. 1. Visualizing the high-order composite spectral collocation scheme for a simple two-dimensional problem. Fig. 1a shows the Chebyshev points on a two-dimensional problem with polynomial order $p = 8$ and $L = 1$ level of refinement. Fig. 1b shows the Gauss–Lobatto points discretizing the boundaries of the leaves using order $q = 6$. When merging nodes together, it is important to distinguish between the *exterior* boundary points, drawn with blue dots, and the *interior* boundary points, drawn with red ×'s. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

describe the trees representing the spatial partition. To that end, we will sometimes refer to the elements as *nodes* and elements at the lowest level of the tree as *leaves*. The element at the highest level of the tree, which represents the entire computational domain, is sometimes called the *root*.

Each leaf is discretized using a tensor product of Chebyshev–Lobatto points, with user-specified order p. This requires p^d points per leaf in dimension $d = 2, 3$. HPS methods typically call for an order-q Gauss–Legendre quadrature rule to represent the boundary of each leaf. For simplicity and stability, we always use $q = p - 2$, following Gillman et al. [3]. In two and three dimensions, there are $(2\ d)q^{d-1}$ boundary discretization points per leaf. We show the interior and boundary points in Fig. 1. The resulting discretization has $N = n_{\text{leaves}}p^d = (2^L p)^d$ interior discretization points.

3.2. Local solve stage

The first step is to discretize the differential operator restricted to each leaf on the p^d Chebyshev discretization points. We call the resulting matrix $\boldsymbol{L}^{(i)}$ for leaf i. This matrix is a combination of Chebyshev spectral differentiation matrices [34] and evaluations of the spatially varying coefficient functions. The source function f is discretized on the same points, and we call the resulting vector $\boldsymbol{f}^{(i)}$. At this point, the HPS algorithm solves a local boundary-value problem on each patch. Standard practice [3,17] is to solve this problem using a "boundary bordering" technique which enforces the differential operator on the Chebyshev nodes interior to the leaf, and enforces a Dirichlet or impedance boundary condition on the Chebyshev nodes on the leaf's boundary. At this point in the algorithm, the correct boundary values to enforce at each leaf are unknown, so $\boldsymbol{L}^{(i)}$ and $\boldsymbol{f}^{(i)}$ are used to precompute a solution map for the local problem. To precompute this local solution map, the algorithm constructs a matrix $\boldsymbol{Y}^{(i)}$ which maps from any boundary data $\boldsymbol{g}^{(i)}$ to the corresponding homogeneous solution on the Chebyshev nodes. The algorithm also computes $\boldsymbol{v}^{(i)}$, a vector evaluating the particular solution on the Chebyshev nodes. Both $\boldsymbol{Y}^{(i)}$ and $\boldsymbol{v}^{(i)}$ can be expressed simply in terms of the input data; lines 3 and 4 in Algorithm 7 in the Appendix give these expressions. For each leaf, the algorithm also constructs a Poincaré–Steklov matrix $\boldsymbol{T}^{(i)}$ and a vector of outgoing data $\boldsymbol{h}^{(i)}$. Computing $\boldsymbol{T}^{(i)}$ and $\boldsymbol{h}^{(i)}$ only requires multiplying $\boldsymbol{Y}^{(i)}$ and $\boldsymbol{v}^{(i)}$ with a fixed, precomputed operator which composes interpolation matrices from Chebyshev to Gauss–Legendre discretization points and spectral Chebyshev differentiation matrices.

Algorithm 1 shows that this stage of the algorithm is a long loop over linear algebra operations. The size of these operations is controlled by the polynomial order p, and we find these operations are efficient for the orders $p \leq 16$ considered in this work. The units of work inside the loop are *embarrassingly parallel*, meaning that one iteration does not depend on the output of any other iterations. Furthermore, because we hold p constant on all leaves, all of the linear algebra operations are homogeneous, meaning they all operate on the same sizes of matrices. This computational structure facilitates GPU acceleration by batching and parallelizing local solves. We give full details describing the local solve stage in Algorithms 7 and 8 in the Appendix.

3.3. Merge stage

After computing $\boldsymbol{T}^{(i)}$ and $\boldsymbol{h}^{(i)}$ for each leaf in the local solve stage, the HPS algorithm begins merging nodes of the tree together. This process creates a hierarchy of solution operators which will later be used to propagate the boundary data on $\partial\Omega$ to the boundary of each leaf. For 2D problems, our implementation merges nodes four at a time, and for 3D problems, our implementation merges nodes eight at a time. Suppose the algorithm is merging a set of nodes $\{a, b, \ldots\}$ which all share parent node j. The algorithm has access to the following data:

- $\{\boldsymbol{T}^{(a)}, \boldsymbol{T}^{(b)}, \ldots\}$, the Poincaré–Steklov matrices of the nodes being merged.
- $\{\boldsymbol{h}^{(a)}, \boldsymbol{h}^{(b)}, \ldots\}$, the outgoing boundary data due to the particular solution of the nodes being merged.

At this point, it is helpful to distinguish between vectors that are defined along the *exterior* of the patches being merged and vectors that are defined along the *interior* of the patches being merged. We indicate these vectors with subscripts ext and int, respectively. See

Algorithm 1: Local solve stage. Full details are available in Algorithms 7 and 8.

Input: Discretized differential operators $\{\boldsymbol{L}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; discretized source functions $\{\boldsymbol{f}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; precomputed interpolation and differentiation matrices

for Leaf $i = 1, \dots, n_{\text{leaves}}$ **do**
  Perform boundary bordering to $\boldsymbol{L}^{(i)}$
  Invert the resulting matrix `// Main computational work`
  Construct $\boldsymbol{Y}^{(i)}$, the interior solution matrix
  Construct $\boldsymbol{T}^{(i)}$, the Poincaré–Steklov matrix
  Construct $\boldsymbol{v}^{(i)}$, the leaf-level particular solution
  Construct $\boldsymbol{h}^{(i)}$, the outgoing boundary data

Result: Poincaré–Steklov matrices $\{\boldsymbol{T}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; outgoing boundary data $\{\boldsymbol{h}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; interior solution matrices $\{\boldsymbol{Y}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; leaf-level particular solutions $\{\boldsymbol{v}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

Fig. 1b for a diagram of the interior and exterior points in a 2D merge operation. The goal of the merge operation is to precompute a solution operator which propagates the information from the exterior boundary points to the interior boundary points. This solution operator takes the form $\boldsymbol{g}_{\text{ext}}^{(j)} \mapsto \boldsymbol{S}^{(j)}\boldsymbol{g}_{\text{ext}}^{(j)} + \tilde{\boldsymbol{g}}^{(j)}$. In this equation, $\boldsymbol{S}^{(j)}\boldsymbol{g}_{\text{ext}}^{(j)}$ evaluates the homogeneous solution on the interior boundary points, and $\tilde{\boldsymbol{g}}^{(j)}$ evaluates the particular solution on the interior boundary points. As in Section 3.2, the boundary data $\boldsymbol{g}_{\text{ext}}^{(j)}$ is not available at this stage of the algorithm, but it is possible to precompute the other parts of the solution operator. To that end, each merge operation will compute:

- $\boldsymbol{S}^{(j)}$, the propagation operator for node j, which maps incoming homogeneous boundary data from the exterior boundary points to the interior boundary points.
- $\tilde{\boldsymbol{g}}^{(j)}$, the incoming boundary data due to the particular solution evaluated at the interior boundary points.
- $\boldsymbol{T}^{(j)}$, the Poincaré–Steklov matrix for node j.
- $\boldsymbol{h}^{(j)}$, the outgoing boundary data for node j.

$\boldsymbol{S}^{(j)}$ and $\tilde{\boldsymbol{g}}^{(j)}$ will be used in the final stage of the HPS method when applying the precomputed solution operator, and $\boldsymbol{T}^{(j)}$ and $\boldsymbol{h}^{(j)}$ will be used in a future merge operation.

To compute the merge outputs, the HPS algorithm sets up a system of equations for a given $\boldsymbol{g}_{\text{ext}}^{(j)}$ and unknown $\boldsymbol{g}_{\text{int}}^{(j)}$ and solve using a Schur complement approach.[2] The constraints in this system come from our knowledge that the solution and its derivative will be continuous across merge interfaces. The resulting system of constraints is a linear system and can be written in a blockwise fashion:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{g}_{\text{ext}}^{(j)} \\ \boldsymbol{g}_{\text{int}}^{(j)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_{\text{ext}}^{(j)} - \boldsymbol{h}_{\text{ext}}^{(\text{child})} \\ -\boldsymbol{h}_{\text{int}}^{(\text{child})} \end{bmatrix}. \tag{3}$$

The next step is to build the matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ blockwise from the child Poincaré–Steklov matrices $\{\boldsymbol{T}^{(a)}, \boldsymbol{T}^{(b)}, \dots\}$, build $\boldsymbol{h}_{\text{ext}}^{(\text{child})}$ and $\boldsymbol{h}_{\text{int}}^{(\text{child})}$ from the child outgoing boundary data $\{\boldsymbol{h}^{(a)}, \boldsymbol{h}^{(b)}, \dots\}$, and use $\boldsymbol{u}_{\text{ext}}^{(j)}$ to represent the unknown solution evaluated on the exterior boundary points. For 2D DtN merges, these objects are defined in Eqs. (A.2) to (A.8). Derivations for the 2D ItI and 3D DtN cases can be found in Appendices B, C. At this point in the algorithm, $\boldsymbol{u}_{\text{ext}}^{(j)}$ is unknown, which means one can not directly invert the linear system to solve for $\boldsymbol{g}_{\text{ext}}^{(j)}$ or $\boldsymbol{g}_{\text{int}}^{(j)}$. However, one can use a Schur complement approach to partially solve the system:

$$\begin{bmatrix} \boldsymbol{A} - \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{C} & \boldsymbol{0} \\ \boldsymbol{D}^{-1}\boldsymbol{C} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{g}_{\text{ext}}^{(j)} \\ \boldsymbol{g}_{\text{int}}^{(j)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_{\text{ext}}^{(j)} - \boldsymbol{h}_{\text{ext}}^{(\text{child})} - \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{h}_{\text{int}}^{(\text{child})} \\ -\boldsymbol{D}^{-1}\boldsymbol{h}_{\text{int}}^{(\text{child})} \end{bmatrix}.$$

Interpreting the rows of this linear system gives us the desired outputs:

$$\boldsymbol{u}_{\text{ext}}^{(j)} = \underbrace{\left(\boldsymbol{A} - \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{C}\right)}_{\boldsymbol{T}^{(j)}} \boldsymbol{g}_{\text{ext}}^{(j)} + \underbrace{\boldsymbol{h}_{\text{ext}}^{(\text{child})} - \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{h}_{\text{int}}^{(\text{child})}}_{\boldsymbol{h}^{(j)}}, \tag{4}$$

$$\boldsymbol{g}_{\text{int}}^{(j)} = \underbrace{-\boldsymbol{D}^{-1}\boldsymbol{C}}_{\boldsymbol{S}^{(j)}} \boldsymbol{g}_{\text{ext}}^{(j)} + \underbrace{-\boldsymbol{D}^{-1}\boldsymbol{h}_{\text{int}}^{(\text{child})}}_{\tilde{\boldsymbol{g}}^{(j)}}. \tag{5}$$

Algorithm 2 gives pseudocode for the merge stage of the HPS algorithm. The majority of the computational work for each merge is inverting $\boldsymbol{D}$, which has size proportional to the number of discretization points along the merge interfaces. This matrix is quite small at the lowest levels of the discretization tree and grows as the algorithm proceeds to higher nodes in the tree. Similar to the local solve stage, each merge operation is dominated by linear algebra work, and the units of work inside the inner loop are embarrassingly parallel. This means we can easily use GPU acceleration to parallelize the inner loop of Algorithm 2. The outer loop is iterating over

[2] To the best of our knowledge, the block system associated with the merge four box procedure for the 2D DtN method was first documented in [35]. We generalize this presentation to encompass 3D problems and merging ItI matrices as well.

different levels, which means the computation during outer loop iteration ℓ depends on the outputs of the previous iteration. Because we only use a moderate number of refinement levels $L < 10$, we find Algorithm 2 executes very quickly on the GPU despite this dependency structure. We note that our choice to merge nodes four-to-one and eight-to-one (rather than the standard two-to-one) decreases the length of the outer loop by factors of two and three, respectively.

Algorithm 2: Merge stage. Full details are available in Appendices A.2, B.2, C.2.

Input: Leaf-level Poincaré–Steklov matrices $\{\boldsymbol{T}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; Leaf-level outgoing boundary data $\{\boldsymbol{h}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

for *Merge level* $\ell = L-1, \dots, 0$ **do**
  for *Node* j *in level* ℓ **do**
    Let $a, b, \dots$ be the children of node j
    Use $\{\boldsymbol{T}^{(a)}, \boldsymbol{T}^{(b)}, \dots\}$ to build blocks $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$, and $\boldsymbol{D}$
    Use $\{\boldsymbol{h}^{(a)}, \boldsymbol{h}^{(b)}, \dots\}$ to build $\boldsymbol{h}_{\text{ext}}^{(\text{child})}$ and $\boldsymbol{h}_{\text{int}}^{(\text{child})}$
    Invert $\boldsymbol{D}$; // Main computational work
    Evaluate $\boldsymbol{T}^{(j)}$, $\boldsymbol{h}^{(j)}$, $\boldsymbol{S}^{(j)}$, $\tilde{\boldsymbol{g}}^{(j)}$; // Eqs. (4) and (5)

Result: Poincaré–Steklov matrices $\boldsymbol{T}^{(j)}$ for each node; outgoing boundary data $\boldsymbol{h}^{(j)}$ for each node; propagation operators $\boldsymbol{S}^{(j)}$ for each node; incoming particular solution data $\tilde{\boldsymbol{g}}^{(j)}$ for each node

3.4. Downward pass

In the final stage of the HPS method, all parts of the structured solution operators mapping $g \mapsto u$ have been computed. The HPS algorithm evaluates this structured solution operator by propagating information down the discretization tree from the boundary of the root to the interior of the leaves. The $\boldsymbol{S}^{(j)}$ matrices propagate the homogeneous boundary data to the merge interfaces, and the $\tilde{\boldsymbol{g}}^{(j)}$ vectors add back in the particular solution (line 4 in Algorithm 3.) This part of the HPS method is extremely fast, as it only involves matrix-vector products. As with the structure of the merge stage, the iterations of the inner loop are embarrassingly parallel and can be batched on the GPU. We show the pseudocode for this stage in Algorithm 3.

Algorithm 3: Downward pass.

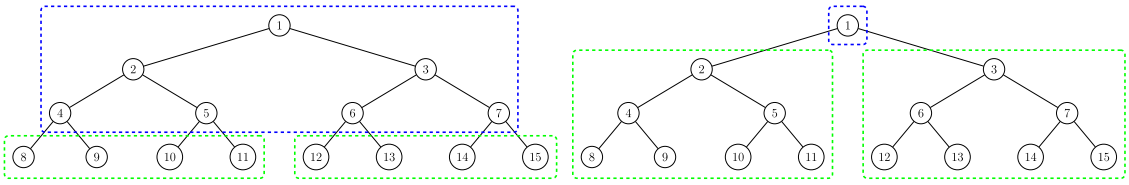
Input: Boundary data $\boldsymbol{g}$; propagation operators $\boldsymbol{S}^{(j)}$ for each node; incoming particular solution data $\tilde{\boldsymbol{g}}^{(j)}$ for each node; leaf-level interior solution matrices $\{\boldsymbol{Y}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; leaf-level particular solutions $\{\boldsymbol{v}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

for *Merge level* $\ell = 0, \dots, L-1$ **do**
  for *Node* j *in level* ℓ **do**
    Look up $\boldsymbol{S}^{(j)}, \boldsymbol{g}^{(j)}$, and $\tilde{\boldsymbol{g}}^{(j)}$
    $\boldsymbol{g}_{\text{int}} = \boldsymbol{S}^{(j)} \boldsymbol{g}^{(j)} + \tilde{\boldsymbol{g}}^{(j)}$
    Let $a, b, \dots$ be the children of node j
    Concatenate $\boldsymbol{g}_{\text{int}}$ and $\boldsymbol{g}^{(j)}$ to form $\{\boldsymbol{g}^{(a)}, \boldsymbol{g}^{(b)}, \dots\}$
for *Leaf* $i = 1, \dots, n_{leaves}$ **do**
  $\boldsymbol{u}^{(i)} = \boldsymbol{Y}^{(i)} \boldsymbol{g}^{(i)} + \boldsymbol{v}^{(i)}$

Result: Leaf-level solutions on the Chebyshev discretization points $\{\boldsymbol{u}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

4. Hardware acceleration for HPS methods

While the algorithms presented in Section 3 have attractive computational complexity ($O(p^6 n_{\text{leaves}} + p^3 n_{\text{leaves}}^{3/2})$ in 2D and $O(p^9 n_{\text{leaves}} + p^6 n_{\text{leaves}}^2)$ in 3D) and possibilities for parallel execution, they also incur large memory footprints. At each step of the algorithm, dense solution matrices are precomputed and must be stored for future use. The outputs of the merge stage dominate the memory complexity of the method. In 2D, the overall memory complexity of storing these matrices is $O(p^2 n_{\text{leaves}} L)$, and in 3D, the memory complexity is $O(p^4 n_{\text{leaves}} 2^L)$, with a large prefactor. This poses a significant challenge for GPU acceleration as general-purpose GPU architectures have significantly more processor cores per unit of memory than standard multicore CPU nodes. While standard multicore compute nodes may have 1TB of available random-access memory (host RAM), high-end GPUs have only 80GB of on-device memory, with slow interconnects between the GPU and host RAM. This means that batched linear algebra operations are extremely fast on the GPU, but the overall algorithm is slowed by steps transferring data between the GPU and the host. Thus, to efficiently accelerate HPS algorithms on the GPU, one must devote significant thought to reducing the memory footprint of these algorithms. In this section, we introduce two ideas to reduce this memory footprint in two and three-dimensional problems.



(a) Leaf recomputation strategy **(b)** Subtree recomputation strategy

Fig. 2. Comparing the batching patterns of the two different recomputation strategies. The leaf recomputation strategy in Fig. 2a performs the local solve stage operations in large batches and then performs all of the merge stages in a separate batch. Our proposed subtree recomputation strategy (Fig. 2b) performs local solves and multiple levels of the merge stage for a complete subtree of the discretization tree structure.

4.1. Recomputation strategies to minimize communication costs

CPU-bound implementations of HPS methods in 2D often spend an order of magnitude longer in the local solve stage than in the merge or downward pass stages [17]. This suggests that HPS schemes can be greatly accelerated by placing the local solve stage computation on the GPU alone. Indeed, such savings have been observed in [25]. We observe that for the lowest levels of the merge stage, each unit of computational work is similarly small, suggesting that the GPU can efficiently accelerate this part of the algorithm, too, provided the algorithm is correctly expressed to leverage its inherent parallelism. In many GPUs, the interconnect between the host device and the GPU's on-device memory is very slow in relation to the speed at which the thousands of processor cores can process data. This means that for large problem sizes, operations transferring precomputed solution operators to and from the GPU are a major impediment to fast execution as they incur a large latency and are often "blocking" operations, which require all parallel threads to complete before executing.

For large problem sizes in 2D, it is advantageous to delete some data computed in the early stages of the algorithm and recompute it later when necessary. This strategy increases the number of floating point operations on the GPU but minimizes costly data transfers. A leaf-level recomputation strategy for implementing HPS algorithms on a GPU is presented in Algorithm 4; a similar method is presented in [25]. The leaf recomputation strategy avoids transferring the $\{\boldsymbol{Y}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$ and $\{\boldsymbol{v}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$ by performing the local solve stage again at the end of the algorithm. Under this recomputation strategy, all of the leaf-level Poincaré–Steklov matrices must be transferred to RAM during the local solve stage, and then back to the GPU during the merge stage.

We find that it is advantageous to push the idea of reducing data transfers at the cost of more floating-point operations further. In our proposed recomputation strategy (Algorithm 5), we delete and recompute the products of the local solve stage and multiple levels of the merge stage. To implement this, we operate in batches defined by "complete subtrees", which are subtrees containing all of the descendants of a particular node j. We break the lowest levels of the discretization tree into the largest complete subtrees where the computations in Algorithms 1 and 2 can all fit into a GPU's on-device memory. The size of these maximal complete subtrees varies depending on GPU memory, polynomial order p, and floating-point datatype; in our experiments, these maximal complete subtrees usually have depth 6 or 7. For each such complete subtree, we perform all of the local solve and merge operations, after which we only save the top-level Poincaré–Steklov matrix and outgoing boundary data vector. Because this is a small amount of data, we can store it on the GPU, and do not need to move the outputs to host RAM. After processing all of the subtrees, the final merge stages are performed on the GPU. The downward pass is evaluated sequentially on the different subtrees, at which point the local solve stage and low-level merges must be recomputed. This recomputation method was inspired by optimizations for contemporary deep learning architectures [36,37], which suggest kernel fusion, a technique that performs multiple steps of a sequential computation at once to keep the necessary data near the processor cores. We visualize the different recomputation methods in Fig. 2.

In Fig. 3, we compare the performance of our method across computer architectures and recomputation strategies. We consider two different architectures, a multicore Intel Xeon node with a 64-core processor, and a GPU architecture using a single Nvidia H100 GPU. Evaluating our method on the GPU gives us significant speedups over the multicore CPU architecture, even when using an implementation with no recomputation strategy, which transfers all precomputed matrices to host RAM after each algorithm step. The two recomputation strategies begin to diverge for problem sizes over 10^7 discretization points, at which point the precomputed matrices cannot all fit on the GPU. We use subtree depth 7 for this experiment; we explore the effect of this choice in Section E by measuring the runtime for different subtree depths. We also estimate the percentage of peak double-precision floating point operations per second (FLOPS) achieved by the different recomputation strategies. Our proposed recomputation strategy uses the most floating-point operations and has the fastest runtime, which means it reaches a higher percentage of peak FLOPS than the other implementations.

4.2. An adaptive discretization strategy to reduce memory complexity in 3D

When extending from two to three dimensions, we face different computational challenges. A large part of the difficulty involves the size of the matrices arising in the merge stage discussed in Section 3.3. For each merge operation, the matrix $\boldsymbol{D}$ must be inverted. This matrix has a number of rows and columns proportional to the number of discretization points that lie along the interfaces being merged. In two dimensions, the size of this merge interface is $O\left(p2^{\ell}\right)$ to merge nodes ℓ levels above the leaves. In three dimensions, the size of this merge interface is $O\left(p^2 4^{\ell}\right)$. Because this quantity grows very quickly for 3D problems as we increase the tree depth L,

Algorithm 4: Leaf recomputation strategy.

Input: Differential operators $\{\boldsymbol{L}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; source functions $\{\boldsymbol{f}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; boundary data $\boldsymbol{g}$

Let b be the maximum batch size that can fit on the GPU
Split the indices $\{1, 2, \dots, n_{\text{leaves}}\}$ into batches $I_1, I_2, \dots, I_{\lceil n_{\text{leaves}}/b \rceil}$
for *Batch* j **do**
    Move $\{\boldsymbol{L}^{(i)}\}_{i\in I_j}$ and $\{\boldsymbol{f}^{(i)}\}_{i\in I_j}$ to the GPU
    Perform the local solve stage for this batch of leaves
    Delete $\{\boldsymbol{Y}^{(i)}\}_{i\in I_j}$ and $\{\boldsymbol{v}^{(i)}\}_{i\in I_j}$
    Transfer $\{\boldsymbol{T}^{(i)}\}_{i\in I_j}$ and $\{\boldsymbol{h}^{(i)}\}_{i\in I_j}$ to host RAM
Concatenate $\{\boldsymbol{T}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$ and $\{\boldsymbol{h}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$ and transfer to GPU
Perform all merge operations on the GPU and transfer all $\boldsymbol{S}^{(i)}$ and $\tilde{\boldsymbol{g}}^{(i)}$ to host RAM
Propagate boundary data to the leaves
Transfer leaf-level boundary data $\{\boldsymbol{g}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$ to host RAM
for *Batch* j **do**
    Move $\{\boldsymbol{L}^{(i)}\}_{i\in I_j}$, $\{\boldsymbol{f}^{(i)}\}_{i\in I_j}$, and $\{\boldsymbol{g}^{(i)}\}_{i\in I_j}$ to the GPU
    Compute local solutions $\boldsymbol{u}^{(i)}_{i\in I_j}$
    Transfer $\{\boldsymbol{u}^{(i)}\}_{i\in I_j}$ to host RAM

Result: Solutions on the Chebyshev discretization points $\{\boldsymbol{u}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

Algorithm 5: Subtree recomputation strategy.

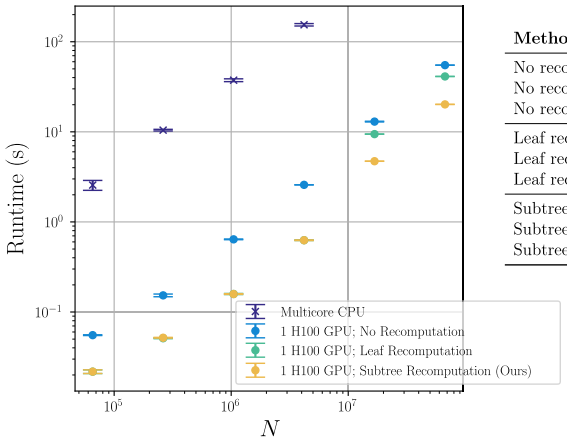
Input: Differential operators $\{\boldsymbol{L}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; source functions $\{\boldsymbol{f}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; boundary data $\boldsymbol{g}$

Let $M = m_1, m_2, \dots$ be the roots of the maximal subtrees
for *Subtree rooted at* m_j **do**
    Let I_j be the set of leaves of the subtree
    Move $\{\boldsymbol{L}^{(i)}\}_{i\in I_j}$ and $\{\boldsymbol{f}^{(i)}\}_{i\in I_j}$ to the GPU
    Perform the local solve stage for this subtree
    Delete $\{\boldsymbol{Y}^{(i)}\}_{i\in I_j}$ and $\{\boldsymbol{v}^{(i)}\}_{i\in I_j}$
    Merge the leaves to the top of subtree m_j, deleting all outputs except $\boldsymbol{T}^{(m_j)}$ and $\boldsymbol{h}^{(m_j)}$
    Keep $\boldsymbol{T}^{(m_j)}$ and $\boldsymbol{h}^{(m_j)}$ on GPU
Perform final merge operations on the GPU
Propagate boundary data to the roots of the maximal subtrees
Transfer boundary data to host RAM
for *Subtree rooted at* m_j **do**
    Let I_j be the set of leaves of the subtree
    Move $\{\boldsymbol{L}^{(i)}\}_{i\in I_j}$, $\{\boldsymbol{f}^{(i)}\}_{i\in I_j}$ and $\boldsymbol{g}^{(m_j)}$ to the GPU
    Perform the local solve stage for this subtree
    Merge the leaves to the top of subtree j
    Propagate boundary information down the tree to the leaves
    $\boldsymbol{u}^{(i)} \leftarrow \boldsymbol{Y}^{(i)}\boldsymbol{g}^{(i)} + \boldsymbol{v}^{(i)}$
    Transfer $\{\boldsymbol{u}^{(i)}\}_{i\in I_j}$ to host RAM

Result: Solutions on the Chebyshev discretization points $\{\boldsymbol{u}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

we quickly run out of memory required to store and invert the matrices on the GPU at the highest level of the merge stage. Performing the top-level merge operation is when the instantaneous memory footprint peaks, as space for $\boldsymbol{D}$, $\boldsymbol{D}^{-1}$, and various buffers must all be allocated on the GPU simultaneously. In contrast with other parts of the algorithm, this peak memory footprint is not reducible by strategies such as batching or data transfer. Because of this need to invert matrices near the memory limits of the GPU, we transfer data to and from the GPU at each merge level and do not use the recomputation strategies discussed in Section 4.1.

To reduce the size of $\boldsymbol{D}$ at the final merge step, we propose to extend the adaptive HPS method presented for 2D problems in Geldermans and Gillman [1] to three dimensions. This method adaptively refines element sizes in a data-dependent manner, in an effort to concentrate discretization points in the regions of the domain where the coefficient and source functions have high local variation. In Table 1, we show that the size of $\boldsymbol{D}$ generated using our adaptive refinement technique is much smaller than that of the uniform refinement with no loss in accuracy.



Method	N	Runtime (s)	% of Peak FLOPS
No recomputation	4,194,304	2.58	1.56%
No recomputation	16,777,216	12.99	2.30%
No recomputation	67,108,864	54.99	4.17%
Leaf recomputation	4,194,304	0.63	6.45%
Leaf recomputation	16,777,216	9.43	3.27%
Leaf recomputation	67,108,864	41.18	5.66%
Subtree recomputation (Ours)	4,194,304	0.63	6.45%
Subtree recomputation (Ours)	16,777,216	4.02	14.86%
Subtree recomputation (Ours)	67,108,864	17.43	20.01%

Fig. 3. Even with a naïve implementation of the HPS algorithm which does not perform any recomputation, using a single GPU achieves large speedups over a multicore CPU system. When we use our proposed subtree recomputation strategy, the speedup increases by another factor of two. (Left) We vary $L = 4, \ldots, 9$ and hold $p = 16$ fixed to generate problems with $N = p^2 4^L = 256 \times 4^L = 4^{L+4}$ degrees of freedom. We measure the total runtime of our 2D method merging DtN matrices; this runtime includes the execution of the local solve stage, the merge stage, and the downward pass. Vertical error bars show $\pm$ 1 standard error computed over five trials. (Right) For each of the GPU implementations, we compute the total number of FLOPS and report this as a percentage of the GPU's peak FLOPS, estimated by the manufacturer to be 34×10^{12}.

Table 1
Adaptive discretization methods can greatly reduce the peak memory requirements of HPS methods in three dimensions. We present the size of the discretization tree and final merge steps in our 3D "wavefront" example (Section 6.1). We compare adaptive and uniform discretizations that have similar errors and observe the adaptive discretization strategy can greatly reduce the size of the final $\boldsymbol{D}$ matrix, a proxy for peak memory usage.

Method	p	Relative ℓ_∞ Error	# Leaves	Size of $\boldsymbol{D}$
Uniform	8	1.48×10^{-4}	512	6,912
Adaptive	8	1.45×10^{-4}	190	2,700
Uniform	12	3.62×10^{-7}	512	19,200
Adaptive	12	2.04×10^{-7}	442	7,500
Uniform	16	4.20×10^{-6}	64	9,408
Adaptive	16	1.41×10^{-6}	57	4,116

Developing a version of the HPS methods presented in Section 3 that is compatible with an adaptive discretization requires slight modification of the algorithms presented in the previous section. The major changes are the introduction of a method for adaptively refining our octree and a method for merging nodes with different levels of refinement.

4.2.1. *Criterion for adaptive refinement*

For a given leaf of the discretization tree, let $\boldsymbol{x}_0$ be the set of p^3 Chebyshev points discretizing the leaf. Let $\boldsymbol{x}_1$ be the set of $8p^3$ discretization points found by breaking the leaf into eight children and creating a Chebyshev grid on each child. Let $\boldsymbol{L}_{8f1}$ be an interpolation matrix mapping from $\boldsymbol{x}_0$ to $\boldsymbol{x}_1$. We evaluate whether a function is sufficiently refined on a leaf by checking whether we can use polynomial interpolation to accurately map from evaluations on $\boldsymbol{x}_0$ to evaluations on $\boldsymbol{x}_1$, relative to the global L_∞ norm of the function. We specify a tolerance parameter ϵ, and for each leaf in our tree, we check the following condition:

$$\frac{\|f(\boldsymbol{x}_1) - \boldsymbol{L}_{8f1} f(\boldsymbol{x}_0)\|_\infty}{\|f\|_\infty} < \epsilon. \tag{6}$$

If this condition is met, we say the leaf is sufficiently refined. Otherwise, we split the leaf into eight children and check each child. We form a final discretization tree by refining each coefficient function in our differential operator, as well as the source term, and taking the union of the resulting trees. Additionally, we refine a few extra leaves to enforce a "level restriction" criterion, which specifies that no leaf can have a side length greater than twice that of its neighbors. This method is an extension of the method for two-dimensional problems presented in [1], which uses a similar relative L_2 convergence criterion and level restriction criterion.

4.2.2. *Local solve stage*

The local solve stage for adaptively refined discretization trees is the same as in the uniform refinement case. Although they are defined over leaves with different volumes, each local boundary value problem has the same number of interior and boundary

discretization points, and the local problems are still embarrassingly parallel. Thus, we can use batched linear algebra to accelerate this part of the algorithm.

4.2.3. Merging nodes with different discretization levels

The nonuniform merge stage is different from the uniform merge stage because neighboring nodes may have different refinement levels, which means the discretization points along either side of the merge interface may not exactly align. Recall the block linear system (Eq. (3)) arising during the merge stage:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{g}_{\text{ext}}^{(j)} \\ \boldsymbol{g}_{\text{int}}^{(j)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_{\text{ext}}^{(j)} - \boldsymbol{h}_{\text{ext}}^{(\text{child})} \\ -\boldsymbol{h}_{\text{int}}^{(\text{child})} \end{bmatrix}.$$

When neighboring volume elements have different refinement levels, there will be a mismatch between the interior boundary discretization points on either side of the merge interface. We need to decide how to represent $\boldsymbol{g}_{\text{int}}^{(j)}$, $\boldsymbol{h}_{\text{int}}^{(\text{child})}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ in Eq. (3). We choose to discretize these objects using the coarser of the two sets of discretization points along the merge interface; the discretization points along the exterior boundary elements are inherited from the child nodes. To assemble the blocks in Eq. (3), this requires projecting some rows and columns of the Poincaré–Steklov matrices using precomputed interpolation operators which map between one and four 2D Gauss–Legendre panels. The "level restriction" constraint greatly simplifies this compression because the resulting projection operations are guaranteed to be four-to-one.

4.2.4. Downward pass

To propagate the boundary information to the leaf nodes, we follow the general structure of Algorithm 3. However, we must undo the projection along merge interfaces that occurs during the nonuniform merge stage. This is accomplished by applying the precomputed interpolation operators to the boundary data $\boldsymbol{g}^{(i)}$.

5. Numerical examples in two dimensions

In this section, we present numerical results on problems with two spatial dimensions. All experiments in this section were conducted using one Nvidia H100 GPU and a host memory space with 100GB of RAM. In all of the experiments, we use the novel subtree recomputation strategy introduced in Section 4.1; the DtN version of this recomputation strategy uses subtrees of depth 7, and the ItI version of this recomputation strategy uses subtrees of depth 6.

5.1. High-order convergence on variable-coefficient problems with known solutions

There are two main ways to increase the accuracy of our composite spectral colocation scheme: refine each leaf patch into four children or increase the polynomial order of the representation of the solution on each patch. Empirically, the error is controlled by the polynomial order p and the side length of each leaf h. In our implementation, p is specified by the user and h is controlled by L, the user-specified depth of the discretization tree. The tradeoff between these two parameters is a widely-studied topic in numerical analysis and goes by the name of "hp-adaptivity". We study the hp-adaptivity properties of our solver using two problems with variable-coefficient differential operators and known solutions. The first problem is a variant of Poisson's equation with spatially-varying coefficients:

$$\begin{cases} \Delta u(x) - \cos(5x_2) u_{x_1}(x) + \sin(5x_2) u_{x_2}(x) = f(x), & x \in [-1,1]^2, \\ u(x) = g(x), & x \in \partial[-1,1]^2, \end{cases} \tag{7}$$

where u_{x_1} and u_{x_2} are the partial derivatives of u in directions x_1 and x_2, respectively. We manufacture the source f and the Dirichlet data g so the solution to this problem is

$$u(x) = u(x_1, x_2) = e^{5x_1} \sin(5x_2) + \sin(10\pi x_1) \sin(\pi x_2).$$

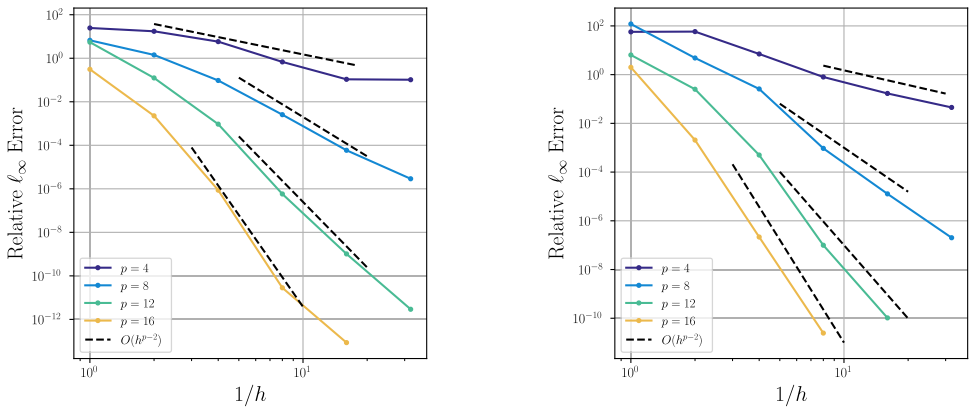
We also study an inhomogeneous Helmholtz problem with a Robin boundary condition:

$$\begin{cases} \Delta u(x) + (1 + e^{-50\|x\|^2}) u(x) = f(x), & x \in [-1,1]^2, \\ u_n(x) + iu(x) = g(x), & x \in \partial[-1,1]^2. \end{cases} \tag{8}$$

We manufacture the source f and the Robin data g so solution to this problem is

$$u(x) = u(x_1, x_2) = e^{i20x_1} + e^{i30x_2}.$$

Fig. 4 shows the convergence of our method on these problems. We measure the relative error of our computed solution $\boldsymbol{u}$ by computing $\|\boldsymbol{u} - \boldsymbol{u}_{\text{true}}\|_\infty / \|\boldsymbol{u}_{\text{true}}\|_\infty$. The ℓ_∞ norms are estimated by taking the maximum over all interior discretization points. In Fig. 4, we see the errors of both the DtN and ItI versions of the method converging at rate $O(h^{p-2})$, even for high polynomial orders.



(a) DtN (b) ItI

Fig. 4. Using p Chebyshev points per dimension on each leaf, and leaves of size h, the relative ℓ_∞ errors of our method converge at rate $O(h^{p-2})$. Fig. 4a shows the convergence of the HPS method using DtN matrices applied to Eq. (7) and Fig. 4b shows the convergence of the HPS method using ItI matrices applied to Eq. (8).

5.2. High-frequency forward wave scattering problems

In this example, we solve a variable-coefficient Helmholtz equation coupled with a Sommerfeld radiation condition. The system is excited by a plane wave with direction $\hat{s} = [1, 0]^\mathsf{T}$ and frequency k:

$$\begin{cases} \Delta u(x) + k^2(1 + q(x))u(x) = -k^2 q(x) e^{ik\langle \hat{s}, x\rangle}, & x \in [-1, 1]^2, \\ \sqrt{r}\left(\frac{\partial u}{\partial r} - iku\right) \to 0, & r = \|x\|_2 \to \infty. \end{cases} \tag{9}$$

Eq. (9) models time-harmonic wave scattering in many imaging modalities, such as sonar or radar imaging, geophysical sensing, and nondestructive testing of materials [38]. As such, forward wave scattering solvers are often used inside an inner loop of optimization routines for solving these inverse problems. These forward solvers must be highly optimized as they are evaluated hundreds or thousands of times over the course of an algorithm.

To solve Eq. (9), we use the ItI variant of our implementation, and at the top level of the merge stage, we solve a boundary integral equation which enforces the Sommerfeld radiation condition [3]. Discretizing the boundary integral equation requires a high-order Nyström method to generate single- and double-layer potential matrices, which we perform in MATLAB using the `chunkIE` package [39]. For each discretization level and frequency, generating these matrices takes a few seconds on a standard laptop. Because these matrices can be precomputed once for a given discretization level and frequency, we do not include the time required to generate these matrices in our runtime measurements. The solution of this boundary integral equation specifies incoming impedance data, which is propagated down the tree to form the interior solution. While we solve Eq. (9) for one source direction $\hat{s}$, this scheme can compute solutions for multiple different sources in parallel at the cost of a few extra matrix-vector multiplications.

In Figs. 6 and 8, we measure the runtime and accuracy of our GPU-accelerated solver. Because analytical solutions for Eq. (9) are unavailable for general scattering potentials $q(x)$, we measure error relative to an overrefined reference solution $\boldsymbol{u}_{\text{over}}$ with approximately 2,800 discretization points in both dimensions. For each computed solution $\boldsymbol{u}$, we compute the relative ℓ_∞ error $\|\boldsymbol{u} - \boldsymbol{u}_{\text{over}}\|_\infty / \|\boldsymbol{u}_{\text{over}}\|_\infty$. The ℓ_∞ norm is estimated by taking the maximum over a grid of 500×500 regularly-spaced grid points. We repeat this experiment for two different choices of the scattering potential $q(x)$. We first choose a single Gaussian bump, which loosely focuses the incoming wave:

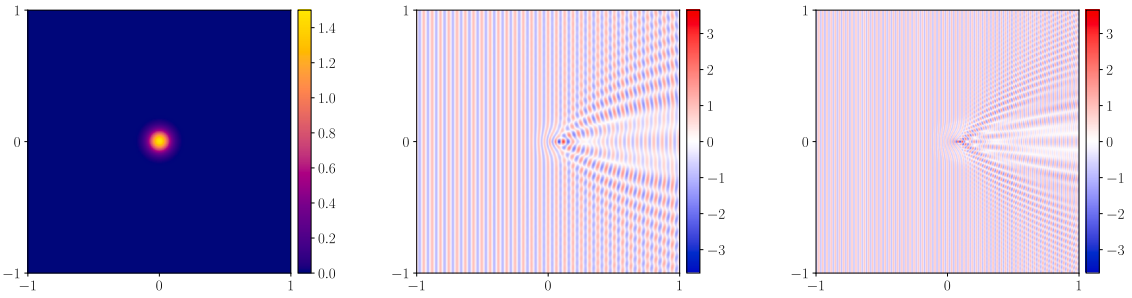
$$q(x) = 1.5e^{-160\|x\|^2}.$$

We also consider a collection of randomly placed Gaussian bumps with centers $\{z^{(i)}\}_{i=1}^{10}$, which causes multiple scattering effects at high wavenumbers:

$$q(x) = \sum_{i=1}^{10} e^{-50\|x - z^{(i)}\|^2}. \tag{10}$$

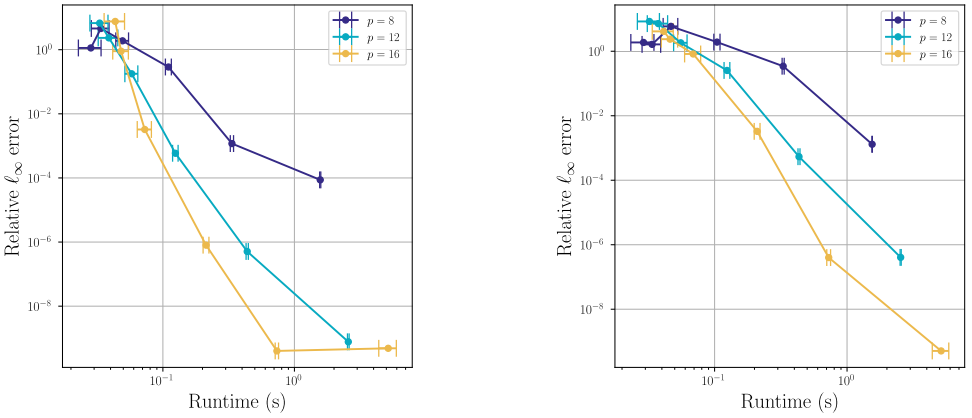
In Figs. 5 and 7, we show these scattering potentials as well as the resulting total wave field $u(x) + e^{ik\langle \hat{s}, x\rangle}$ for different choices of k.

Our GPU-accelerated method with our proposed recomputation strategy is very fast. It is able to compute high-accuracy solutions to these challenging scattering problems in a few seconds. We note that because this method does not rely on any iterative algorithms, the runtime for a fixed discretization level does not depend on the frequency k or the scattering potential $q(x)$. However, finer discretizations are required to achieve a fixed error tolerance as k increases. Figs. 6 and 8 show that polynomial order is $p = 16$ dominates the lower-order methods for all values of k and scattering potentials considered. In these plots, we vary the number of



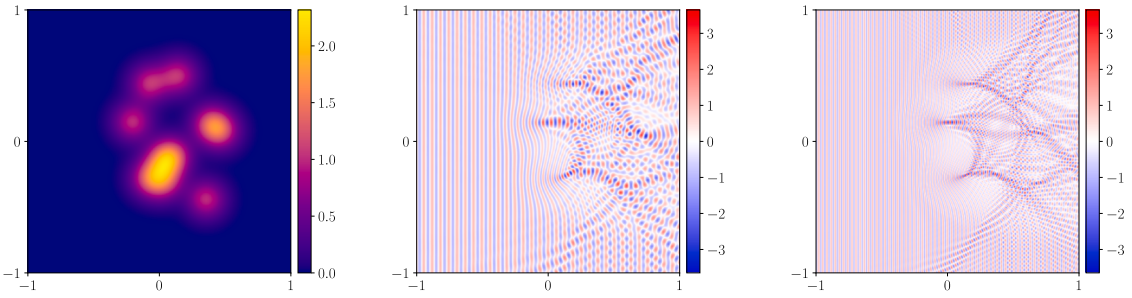
(a) Scattering potential $q(x)$ **(b)** Total wave field; $k = 100$ **(c)** Total wave field; $k = 200$

Fig. 5. Visualizing the solutions of forward scattering problems for the single Gaussian bump scattering potential. Figs. 5b and 5c show the real part of the total wave field.



(a) Runtime and errors when $k = 100$ **(b)** Runtime and errors when $k = 200$

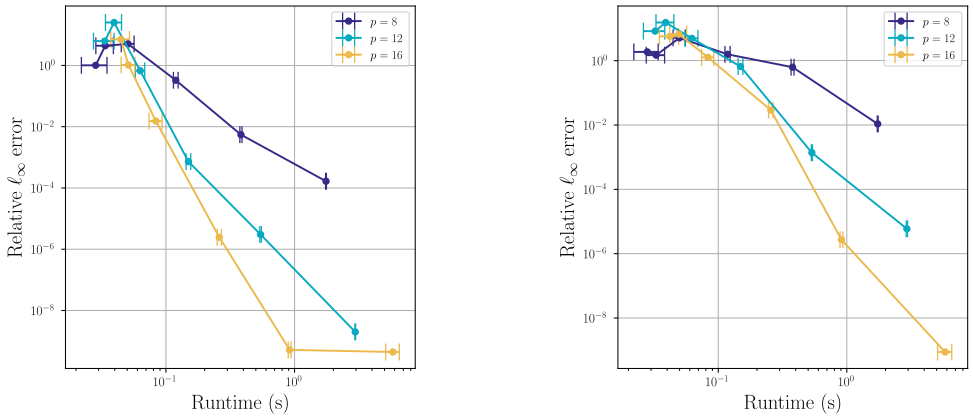
Fig. 6. Error-runtime study on the single Gaussian bump scattering potential. Using GPU acceleration, our method can rapidly converge to high-accuracy solutions in high-frequency wave scattering problems. The runtime measurements include the runtime of the entire HPS algorithm and the setup and solution of the boundary integral equation enforcing the radiation condition. Horizontal error bars show ±1 standard error computed over five trials.



(a) Scattering potential $q(x)$ **(b)** Total wave field; $k = 100$ **(c)** Total wave field; $k = 200$

Fig. 7. Visualizing the solutions of forward scattering problems for the sum of randomly placed Gaussian bumps scattering potential. Figs. 7b and 7c show the real part of the total wave field.

degrees of freedom $N = p^2 4^L$ by varying both p and L. Different colors correspond to different polynomial orders $p \in \{8, 12, 16\}$, and for a given polynomial order, we vary $L \in \{2, 3, \ldots, 7\}$. This generates problems with degrees of freedom varying between 512 and 4,194,304.



(a) Runtime and errors when $k = 100$

(b) Runtime and errors when $k = 200$

Fig. 8. Error-runtime study on the sum of randomly placed Gaussian bumps scattering potential.

5.3. *Solving inverse scattering problems with automatic differentiation*

Our solver is compatible with the JAX automatic differentiation framework, which allows us to very easily implement gradient-based optimization algorithms using our accelerated HPS solver as a forward model. We consider an inverse scattering task to recover an inhomogeneous scattering potential q_θ specified by basis coefficients $\{\theta_j\}$:

$$q_\theta(x) = \sum_{b_j \in B_\gamma} \theta_j b_j(x), \tag{11}$$

$$B_\gamma = \left\{ \sin\left(m_1 \frac{\pi}{2}(x_1 - 1)\right) \sin\left(m_2 \frac{\pi}{2}(x_2 - 1)\right) \middle| \sqrt{m_1^2 + m_2^2} \le \gamma \right\}. \tag{12}$$

The basis B_γ is used in Borges et al. [38] because it spans smooth, bandlimited functions which vanish on the boundary of Ω. The goal of this inverse scattering task is to estimate θ. Eq. (9) describes how $q_\theta(x)$ affects the scattered wave field $u_\theta(x)$. Our forward model $\mathcal{F} \circ \mathcal{B}$ is a composition of maps where $\mathcal{B} : \theta \mapsto q_\theta$ is a discrete sine transform and $\mathcal{F} : q_\theta \mapsto u_\theta$ evaluates the solution of Eq. (9) at points $x^{(j)}$ away from the support of the scattering potential. We choose a ground-truth θ^* to be the basis coefficients of the sum of Gaussian bumps scattering potential Eq. (10) projected onto B_γ and generate data $\mathcal{F}[\theta^*]$. Given this data and our knowledge of the forward model, we wish to recover an estimate of θ^*. We can phrase this problem in an optimization framework:

$$\underset{\theta \in \mathbb{R}^{N_\theta}}{\operatorname{argmin}} \|(\mathcal{F} \circ \mathcal{B})(\theta) - (\mathcal{F} \circ \mathcal{B})(\theta^*)\|_2^2. \tag{13}$$

We evaluate $\mathcal{F}$ using our GPU-accelerated fast direct solver. Because our solver is compatible with automatic differentiation, we can solve this optimization problem using gradient-based methods. We first evaluate the accuracy of JAX automatic differentiation by comparing the outputs with the action of $J[\theta]$, the Fréchet derivative of $\mathcal{F} \circ \mathcal{B}$ centered at θ. Borges et al. [38] characterize this derivative in terms of the solution of linear elliptic PDEs which we can compute with our HPS method; we re-state these results in Appendix D.1 for completeness. Fig. 9a shows the convergence of the outputs of automatic differentiation to the Fréchet derivative when holding polynomial order $p = 16$ fixed and varying the depth of the quadtree, $L = 1, \ldots, 5$. For leaf size h, we observe high-order convergence at rate $O\left(h^{p-2}\right)$. For brevity, we defer the details of this experiment to Appendix D.2. We remark that high-order accuracy is possible because B_γ defines a smooth global basis which can be represented using the HPS discretization. This is in contrast to differentiating with respect to the values of q at the HPS discretization points, which effectively introduces nonsmooth coefficient functions. We also note that the use of standard automatic differentiation software requires the presence of the entire computational graph in memory, which means it can not be used in conjunction with the recomputation strategies introduced in Section 4.1.

To solve Eq. (13), we propose to use Gauss–Newton iterations for nonlinear least squares problems. This algorithm requires access to $J[\theta]$, which we implement using JAX automatic differentiation applied to our HPS solver. We use automatic differentiation to implement the actions $J[\theta]^* f$ and $J[\theta]v$ for arbitrary vectors f, v and estimates θ. We pair these subroutines with a sparse linear algebra least-squares solver [40] from the SciPy library [41] to implement the Gauss–Newton algorithm presented in Algorithm 6.

To solve the optimization problem, we use B_γ, $\gamma = 5$, which gives us $N_\theta = 15$ optimization variables. Following Borges et al. [38], we initialize the optimization variables corresponding to the lowest three frequency components to the ground-truth $\theta_0 = \theta_*$, and we initialize the other variables at $\mathbf{0}$. We run 21 iterations of the Gauss–Newton algorithm. Fig. 9 shows the optimization variables take some iterations to approach θ^* and then converge superlinearly to near machine precision. Calculating each Gauss–Newton update is fast because of the GPU acceleration of the forward model, $J[\theta_t]^* f$, and $J[\theta_t]v$. The entire experiment runs in 75 seconds using one H100 GPU.

Algorithm 6: Gauss–Newton iterations for nonlinear least squares problems.

Input: Data $(\mathcal{F}\circ\mathcal{B})(\theta^*)$; Initial estimate θ_0

$t \leftarrow 0$

while *not converged* **do**

  Compose automatic differentiation and our fast direct solver to compile the function $f \mapsto J[\theta_t]^* f$

  Compose automatic differentiation and our fast direct solver to compile the function $v \mapsto J[\theta_t]v$

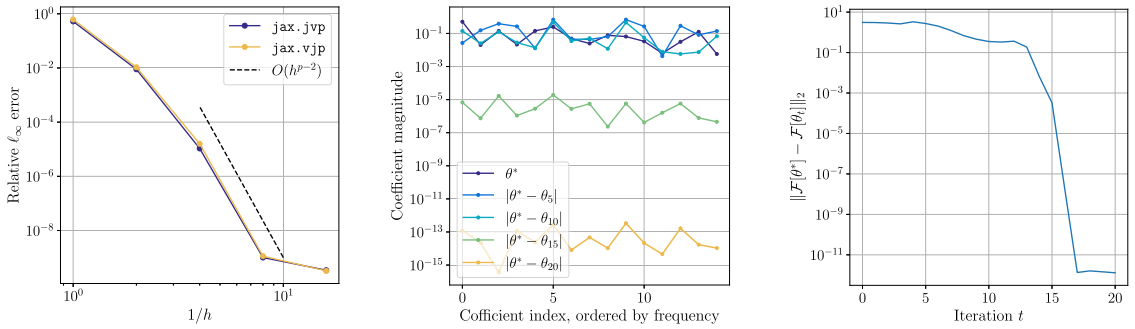
  Define a linear operator $J[\theta_t]$ using subroutines $J[\theta_t]^* f$ and $J[\theta_t]v$

  Use a least-squares solver to compute $\delta \leftarrow \operatorname{argmin}_\delta \|(\mathcal{F}\circ\mathcal{B})(\theta^*) - \big((\mathcal{F}\circ\mathcal{B})(\theta_t) + J[\theta_t]\delta\big)\|_2^2$

  $\theta_{t+1} \leftarrow \theta_t + \delta$

  $t \leftarrow t + 1$

Result: Final estimate θ_t



(a) (b) (c)

Fig. 9. Our GPU-accelerated PDE solver can interface with automatic differentiation to rapidly solve inverse problems. In Fig. 9a, we show that automatic differentiation applied to our HPS method implementing $\mathcal{F}\circ\mathcal{B}$ converges at high order to the Fréchet derivative of this operator. Experimental details are available in Appendix D.2. In Fig. 9b, we show the magnitude of the ground-truth coefficients θ^* as well as the component-wise errors of intermediate estimates. Fig. 9c shows the objective value of the optimization problem, which reaches near machine precision in 17 iterations.

6. Numerical examples in three dimensions

In the three-dimensional examples, we focus on problems with localized regions of high variation. Our adaptive discretization method described in Section 4 is designed for such problems. In these experiments, we use a single Nvidia H100 GPU with 80GB of on-device memory and 200GB of host RAM.

6.1. Adaptive refinement on a problem with known solution

In this example, we study the convergence of our method on a three-dimensional problem with a known analytical solution. We build a problem that is solved by a "wavefront" located along a three-dimensional curved surface. The problem is given by

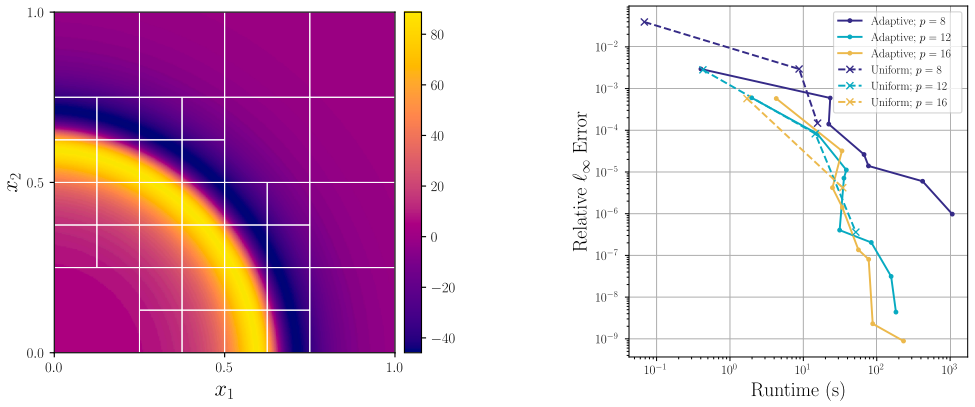
$$\begin{cases} \Delta u(x) = f(x), & x \in [0,1]^3, \\ u(x) = g(x), & x \in \partial[0,1]^3. \end{cases} \tag{14}$$

We manufacture a source term $f(x)$ so the solution takes the form

$$u(x) = u(x_1, x_2, x_3) = \arctan\left(10\sqrt{(x_1-0.5)^2 + (x_2-0.5)^2 + (x_3-0.5)^2} - 0.7\right) \tag{15}$$

and use samples of this function along the boundary to create our boundary data g. Fig. 10a shows that f has a localized region of high variation. A uniform discretization strategy cannot adapt to the locality of this problem, but our adaptive discretization strategy can adaptively refine the octree to place a higher density of discretization points in this region. The adaptive discretization also uses larger leaves, where possible, on the parts of the domain with a very smooth source function.

In Fig. 10b, we show accuracy versus runtime for both a uniform and adaptive discretization applied to this problem. For all methods, we increased the number of discretization points until saturating the GPU's memory limit when inverting the highest-level $\boldsymbol{D}$ matrix. While the uniform methods are fast on this problem, they are not highly accurate because they cannot use more than $L = 3$ levels of uniform refinement. Our adaptive method computes solutions with much higher accuracy before saturating the GPU's memory limit by adaptively placing the discretization points in regions where the source and solution have high variation. Table 1 shows the size of the highest-level merge matrix $\boldsymbol{D}$ for selected points on this graph.



(a) (b)

Fig. 10. Adaptive refinement allows for more accuracy before encountering memory bottlenecks. Fig. 10a shows the source function restricted to the $x_3 = 0$ plane and the adaptive mesh formed with tolerance 1×10^{-7} and Chebyshev parameter $p = 16$. In Fig. 10b, we study the runtime and error of different refinement strategies applied to Eq. (14). For each step of the uniform refinement curve, we refined the uniform grid by one more level. For the adaptive refinement curve, we decreased the adaptive refinement tolerance by a factor of 10. For all methods, we refined until running out of memory on the GPU during the build stage.

6.2. *Linearized Poisson–Boltzmann equation*

An example application where the data and solution have local regions of high variation is the linearized Poisson–Boltzmann equation, a model of the electrostatic properties of a molecule in a solution. This can, for example, be used to compute the stability of a given molecular configuration in a solution. A standard model, developed in [42], starts with atoms represented by point charges $\{z^{(i)}\}_{i=1}^{N_z}$, $z^{(i)} \in \mathbb{R}^3$. These atoms give rise to a charge distribution $\rho(x)$,

$$\rho(x) = \sum_{i=1}^{N_z} e^{-\delta \|x - z^{(i)}\|^2}, \tag{16}$$

and a spatially-varying permittivity function $\varepsilon(x)$,

$$\varepsilon(x) = \epsilon_0 + (\epsilon_\infty - \epsilon_0) e^{-A\rho(x)}. \tag{17}$$

We use parameters $N_z = 50$, $\delta = 45$, $\epsilon_0 = 16$, $\epsilon_\infty = 100$, and $A = 10$ [42]. The atomic centers $\{z^{(i)}\}_{i=1}^{N_z}$ are drawn uniformly from the box $[-0.5, 0.5]^3$. We can now model the electrostatic potential $u(x)$ which is implicitly defined by the linearized Poisson–Boltzmann equation:

$$\begin{cases} \nabla \cdot (\varepsilon(x) \cdot \nabla u(x)) = -\rho(x), & x \in [-1, 1]^3, \\ u(x) = 0, & x \in \partial[-1, 1]^3. \end{cases} \tag{18}$$

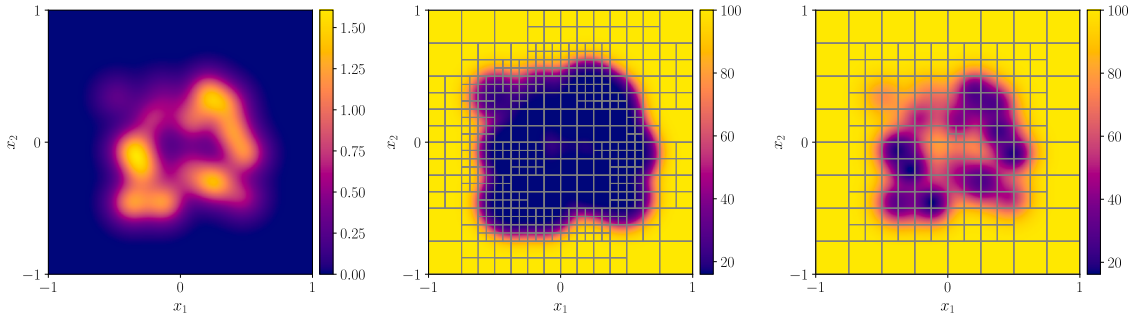
Existing approaches, such as a fast integral equation method [43] and finite difference schemes [44,45] solve a simplified version of Eq. (18), where the permittivity function is replaced by one derived from van der Waals surfaces:

$$\varepsilon_{\text{vdW}}(x) = q(x)\epsilon_0 + (1 - q(x))(\epsilon_\infty - \epsilon_0), \tag{19}$$

$$q(x) = 1 - \prod_{i=1}^{N_z} \left[1 - e^{-\delta \|x - z^{(i)}\|^2}\right]. \tag{20}$$

$\varepsilon_{\text{vdW}}(x)$ is an easier function to resolve to high accuracy as it lacks the steep gradients observed in $\varepsilon(x)$. However, [42] reports "experimentation with a number of dielectric mapping functions using $[\varepsilon_{\text{vdW}}]$ produced dielectric functions that increase toward solvent values far too rapidly with distance from atomic centers," and "$[\varepsilon_{\text{vdW}}]$ also produced undesired patches of high dielectric inside proteins." We use our GPU-accelerated adaptive HPS method to solve Eq. (18) with both permittivity models.

To form an adaptive discretization for this problem, we refine a discretization tree given the charge distribution ρ, the permittivity ε, and the components of $\nabla\varepsilon$. Fig. 11 shows a 2D slice of the charge distribution and permittivity, along with the discretization found by this refinement process. Table 2 gives statistics about the discretization and runtime for a range of tolerances and Chebyshev parameters p. In this table, n_{leaves} is the number of leaves of the resulting discretization tree, $N = n_{\text{leaves}} p^3$ is the total number of discretization points, Max Depth is the maximum number of levels of refinement in the discretization tree, and Runtime measures the wall-clock time in seconds to compute the adaptive discretization and execute all parts of the HPS algorithm.



(a) Source $\rho(x)$ **(b)** Permittivity $\varepsilon(x)$ **(c)** Permittivity $\varepsilon_{\mathrm{vdW}}(x)$

Fig. 11. Visualizing the variable coefficients and source term of our problem. These plots show the source function $\rho(x)$ and permittivity functions $\epsilon(x)$ and $\epsilon_{\mathrm{vdW}}(x)$ restricted to the plane $x_3 = 0$. The adaptive discretizations formed using $p = 8$ and tolerance 1×10^{-4} are shown. This adaptive discretization is found by forming the union of meshes adaptively refined on the source, the permittivity, and components of the gradient of the permittivity.

Table 2
Resource usage statistics for using our 3D adaptive HPS method applied to the linearized Poisson–Boltzmann equation (Eq. (18)) using permittivity $\epsilon(x)$ (top) and simplified permittivity $\epsilon_{\mathrm{vdW}}(x)$ (bottom). We use "OOM" to indicate which discretizations caused out of memory errors when inverting the final $\boldsymbol{D}$ matrix.

p	Tolerance	n_{leaves}	N	Max Depth	Runtime (s)
8	10^{-1}	358	183,296	3	48.4
8	10^{-2}	1,436	735,232	4	177.8
8	10^{-3}	1,884	964,608	5	218.9
8	10^{-4}	7,659	3,921,408	5	1,523.0
10	10^{-1}	253	253,000	3	52.5
10	10^{-2}	449	449,000	4	76.3
10	10^{-3}	1,625	1,625,000	4	243.4
10	10^{-4}	1,982	1,982,000	5	319.1
12	10^{-1}	99	171,072	3	41.5
12	10^{-2}	386	667,008	3	91.0
12	10^{-3}	715	1,235,520	4	190.6
12	10^{-4}	1,695	2,928,960	4	OOM
16	10^{-1}	64	262,144	2	53.9
16	10^{-2}	204	835,584	3	136.2
16	10^{-3}	365	1,495,040	3	OOM
16	10^{-4}	470	1,925,120	4	OOM
p	Tolerance	n_{leaves}	N	Max Depth	Runtime (s)
8	10^{-3}	1,093	559,616	4	154.4
8	10^{-4}	1,737	889,344	5	227.2
8	10^{-5}	5,111	2,616,832	5	869.0
8	10^{-6}	9,423	4,824,576	5	OOM
10	10^{-3}	435	435,000	4	94.5
10	10^{-4}	1,016	1,016,000	4	193.3
10	10^{-5}	1,485	1,485,000	4	241.7
10	10^{-6}	2,605	2,605,000	5	483.0
12	10^{-3}	274	473,472	3	93.7
12	10^{-4}	477	824,256	4	145.8
12	10^{-5}	974	1,683,072	4	275.2
12	10^{-6}	1,366	2,360,448	4	OOM
16	10^{-3}	127	520,192	3	119.7
16	10^{-4}	239	978,944	3	172.2
16	10^{-5}	323	1,323,008	3	227.0
16	10^{-6}	456	1,867,776	4	OOM

7. Conclusion

This paper presents methods for efficiently accelerating HPS algorithms using general-purpose GPUs. Because there is a large amount of inherent parallelism in the structure of the HPS algorithms, they are a natural target for GPU acceleration once adjustments are made to reduce memory complexity. We introduce methods for reducing the memory footprint of HPS algorithms for problems in two and three dimensions.

This work leaves open important questions and avenues for improvement. While our method can efficiently interface with automatic differentiation, we could, in principle, gain much more efficiency by implementing custom automatic differentiation rules to reuse the precomputed solution operators, like those derived in Borges et al. [38]. Our methods of reducing memory complexity could be pushed further by using a hybrid approach, i.e., by performing a few levels of merging via dense linear algebra and then relying on a sparse direct solver such as Yesypenko and Martinsson [26] or Kump et al. [15] for higher-level merges. We hypothesize such a hybrid approach would greatly reduce runtimes for very large problems by reducing the ranks needed to accurately resolve the sparse system matrix. Finally, our methods could be extended to unstructured meshes and surfaces, a setting where nonuniform merge operations make parallel implementations challenging.

CRediT authorship contribution statement

Owen Melia: Writing – original draft, Software, Methodology, Conceptualization; **Daniel Fortunato:** Writing – review & editing, Supervision, Methodology, Conceptualization; **Jeremy Hoskins:** Writing – review & editing, Supervision, Methodology, Conceptualization; **Rebecca Willett:** Writing – review & editing, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Data availability

Our open-source JAX implementation is publicly available at https://github.com/meliao/jaxhps; see [46] for details and documentation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The authors would like to thank Manas Rachh, Leslie Greengard, and Olivia Tsang for many useful discussions. The authors are grateful to the Flatiron Institute for providing the computational resources used to conduct the experiments in this work.

OM and RW gratefully acknowledge the support of NSF DMS-2023109, DOE DE-SC0022232, the Physics Frontier Center for Living Systems funded by the National Science Foundation (PHY-2317138), and the support of the Margot and Tom Pritzker Foundation. JH was supported in part by a Sloan Research Fellowship. This research was supported in part by grants from the NSF (DMS-2235451) and Simons Foundation (MPS-NITMB-00005320) to the NSF-Simons National Institute for Theory and Mathematics in Biology (NITMB). The Flatiron Institute is a division of the Simons Foundation.

Appendix A. Full algorithms for 2D problems using DtN matrices

In this section, we describe the details for the two-dimensional version of our method which merges DtN matrices. In this version of the algorithm, the outgoing boundary data $\boldsymbol{h}$ tabulates the outward-pointing normal derivative of the particular solution, and the incoming boundary data $\boldsymbol{g}$ tabulates the homogenous solution values restricted to patch boundaries. $\boldsymbol{T}$ is a Dirichlet-to-Neumann matrix.

In this section, we use $\boldsymbol{I}_{a\times a}$ to denote the identity matrix of shape $a \times a$ and $\mathbf{0}_d$ to denote a length-d vector filled with 0's. When defining matrices blockwise, we use $\mathbf{0}$ to denote a block filled with 0's, and assume the shape of the block can be determined from the nonzero blocks sharing the same rows and columns.

A.1. Local solve stage

Recall from Section 3.1 that we use a tensor product of order-p Chebyshev–Lobatto points to discretize the interior of each leaf. This results in a grid with p^2 discretization points, and $4p - 4$ of these points lie on the boundary of the leaf. We use order-q Gauss–Legendre points to discretize each side of the leaf's boundary, so there are $4q$ boundary points in total. Thus, to translate the information between the interior and boundary of each leaf, we need to compute spectral differentiation matrices and matrices interpolating between the p Chebyshev and q Gauss points. In particular, we need to precompute the following matrices:

- $\boldsymbol{P}$, with shape $4p-4 \times 4q$, is the operator mapping data sampled on the Gauss boundary points to data sampled on the $4p-4$ Chebyshev points located on the boundary of the leaf. This matrix is constructed using a barycentric Lagrange interpolation matrix mapping from Gauss to Chebyshev points on one side of the leaf; this interpolation matrix is repeated for the other sides. Rows corresponding to the Chebyshev points on the corners of the leaf average the contribution from the two adjoining panels.
- $\boldsymbol{Q}$, with shape $4q \times p^2$, performs spectral differentiation on the p^2 Chebyshev points followed by interpolation to the Gauss boundary points. This matrix is formed by stacking the relevant rows of Chebyshev spectral differentiation matrices to form an operator which evaluates normal derivatives on the $4p-4$ boundary Chebyshev points, and then composing this differentiation operator with a matrix formed from barycentric Lagrange interpolation matrix blocks. These interpolation matrices each map from one Chebyshev panel to one Gauss panel.

To work with $\boldsymbol{L}^{(i)}$, the discretization of the differential operator on leaf i, it is useful to identify I_i and I_e, the sets of discretization points corresponding to the $(p-2)^2$ interior and $4p-4$ exterior Chebyshev points, respectively. Now, we can fully describe the local solve stage in Algorithm 7.

Algorithm 7: 2D DtN local solve stage.

Input: Discretized differential operators $\{\boldsymbol{L}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; discretized source vectors $\{\boldsymbol{f}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; precomputed interpolation and differentiation matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$

for $i = 1, \ldots, n_{leaves}$ **do**

Invert $\boldsymbol{L}^{(i)}(I_i, I_i)$

$$\boldsymbol{Y}^{(i)} = \begin{bmatrix} \boldsymbol{I}_{4p-4\times 4p-4} \\ -\left(\boldsymbol{L}^{(i)}(I_i, I_i)\right)^{-1} \boldsymbol{L}^{(i)}(I_i, I_e) \end{bmatrix} \boldsymbol{P}$$

$$\boldsymbol{v}^{(i)} = \begin{bmatrix} \boldsymbol{0}_{4p-4} \\ -\left(\boldsymbol{L}^{(i)}(I_i, I_i)\right)^{-1} \boldsymbol{f}^{(i)}(I_i) \end{bmatrix}$$

$$\boldsymbol{T}^{(i)} = \boldsymbol{Q}\boldsymbol{Y}^{(i)}$$

$$\boldsymbol{h}^{(i)} = \boldsymbol{Q}\boldsymbol{v}^{(i)}$$

Result: Poincaré–Steklov matrices $\{\boldsymbol{T}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; outgoing boundary data $\{\boldsymbol{h}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; interior solution matrices $\{\boldsymbol{Y}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$, leaf-level particular solutions $\{\boldsymbol{v}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

A.2. Merge stage

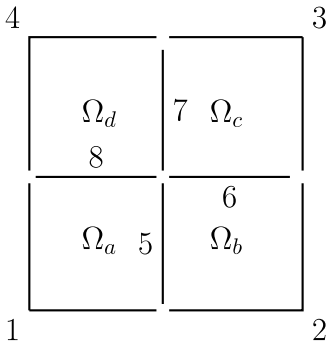


Fig. A.12. Visualizing boundary elements 1 through 8 for two-dimensional merges.

In the 2D merge stage, we are merging four nodes $\Omega_a, \Omega_b, \Omega_c$, and Ω_d, which have *exterior* and *interior* discretization points. We label the exterior boundary sections $1, 2, 3$, and 4, and we label the interior boundary sections $5, 6, 7$, and 8. See Fig. A.12 for a diagram of the different boundary parts. Because the merge stage operates completely on data discretized using Gauss–Legendre panels, there are no discretization points at the corners of nodes. This means each discretization point belongs to exactly one part of the boundary. During this stage of the algorithm, we will be indexing rows and columns of the Dirichlet-to-Neumann matrices according to these boundary sections. For example, we use $\boldsymbol{T}^{(a)}_{1,5}$ to indicate the submatrix of node a's DtN matrix which maps from boundary section 5 to boundary section 1. Suppose each side of $\Omega_a, \Omega_b, \Omega_c$, and Ω_d is discretized with n_{side} discretization points; in this case $\boldsymbol{T}^{(a)}_{1,5}$ will have shape $2n_{\text{side}} \times n_{\text{side}}$.

To implement the merge stage, we use sets of constraints to solve for a mapping from given $\boldsymbol{g}_{\text{ext}}$ to unknown $\boldsymbol{g}_{\text{int}}$. These are vectors tabulating the homogeneous solution along the exterior and interior boundary parts. First are constraints specifying that the solution to the PDE is continuous:

$$\boldsymbol{u}_{\text{ext}} = \boldsymbol{A}\boldsymbol{g}_{\text{ext}} + \boldsymbol{B}\boldsymbol{g}_{\text{int}} + \boldsymbol{h}^{(\text{child})}_{\text{ext}}. \tag{A.1}$$

In this equation $\boldsymbol{u}_{\text{ext}}$ is interpreted as the outward-pointing normal derivative of the solution to the PDE restricted to boundary elements $1, 2, 3$, and 4 with boundary data specified by $\boldsymbol{g}_{\text{ext}}$. In this set of constraints, we define:

$$\boldsymbol{h}_{\text{ext}}^{(\text{child})} = \begin{bmatrix} \boldsymbol{h}_1^{(a)} \\ \boldsymbol{h}_2^{(b)} \\ \boldsymbol{h}_3^{(c)} \\ \boldsymbol{h}_4^{(d)} \end{bmatrix}, \tag{A.2}$$

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{T}_{1,1}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_{2,2}^{(b)} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{3,3}^{(c)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{4,4}^{(d)} \end{bmatrix}, \tag{A.3}$$

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{T}_{1,5}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{1,8}^{(a)} \\ \boldsymbol{T}_{2,5}^{(b)} & \boldsymbol{T}_{2,6}^{(b)} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_{3,6}^{(c)} & \boldsymbol{T}_{3,7}^{(c)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{4,7}^{(d)} & \boldsymbol{T}_{4,8}^{(d)} \end{bmatrix}. \tag{A.4}$$

A second set of constraints enforces that the outward-pointing normal derivatives from neighboring nodes should sum to zero, which is equivalent to enforcing the continuity of the first derivative along the merge interfaces in their respective Cartesian directions. To that end, we use constraints $\boldsymbol{u}_5^{(a)} + \boldsymbol{u}_5^{(b)} = \boldsymbol{0}_{n_{\text{side}}}$, $\boldsymbol{u}_6^{(b)} + \boldsymbol{u}_6^{(c)} = \boldsymbol{0}_{n_{\text{side}}}$, and so on. This gives us an equation:

$$\boldsymbol{0}_{4n_{\text{side}}} = \boldsymbol{C}\boldsymbol{g}_{\text{ext}} + \boldsymbol{D}\boldsymbol{g}_{\text{int}} + \boldsymbol{h}_{\text{int}}^{(\text{child})}. \tag{A.5}$$

In Eq. (A.5), we define

$$\boldsymbol{h}_{\text{int}}^{(\text{child})} = \begin{bmatrix} \boldsymbol{h}_5^{(a)} + \boldsymbol{h}_5^{(b)} \\ \boldsymbol{h}_6^{(b)} + \boldsymbol{h}_6^{(c)} \\ \boldsymbol{h}_7^{(c)} + \boldsymbol{h}_7^{(d)} \\ \boldsymbol{h}_8^{(d)} + \boldsymbol{h}_8^{(a)} \end{bmatrix}, \tag{A.6}$$

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{T}_{5,1}^{(a)} & \boldsymbol{T}_{5,2}^{(b)} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_{6,2}^{(b)} & \boldsymbol{T}_{6,3}^{(c)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{7,3}^{(c)} & \boldsymbol{T}_{7,4}^{(d)} \\ \boldsymbol{T}_{8,1}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{8,4}^{(d)} \end{bmatrix}, \tag{A.7}$$

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{T}_{5,5}^{(a)} + \boldsymbol{T}_{5,5}^{(b)} & \boldsymbol{T}_{5,6}^{(b)} & \boldsymbol{0} & \boldsymbol{T}_{5,8}^{(a)} \\ \boldsymbol{T}_{6,5}^{(b)} & \boldsymbol{T}_{6,6}^{(b)} + \boldsymbol{T}_{6,6}^{(c)} & \boldsymbol{T}_{6,7}^{(c)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_{7,6}^{(c)} & \boldsymbol{T}_{7,7}^{(c)} + \boldsymbol{T}_{7,7}^{(d)} & \boldsymbol{T}_{7,8}^{(d)} \\ \boldsymbol{T}_{8,5}^{(a)} & \boldsymbol{0} & \boldsymbol{T}_{8,7}^{(d)} & \boldsymbol{T}_{8,8}^{(d)} + \boldsymbol{T}_{8,8}^{(a)} \end{bmatrix}. \tag{A.8}$$

Now that the matrices and vectors are defined, we can construct the linear system in Eq. (3) and compute the merged data.

Appendix B. Full algorithms for 2D problems using ItI matrices

In this section, we describe the details for the two-dimensional version of our method which merges ItI matrices. In this version of the algorithm, the outgoing boundary data $\boldsymbol{h}$ tabulates the outgoing impedance data due to the particular solution, and the incoming boundary data $\boldsymbol{g}$ tabulates incoming impedance data due to the homogeneous solution. $\boldsymbol{T}$ is an impedance-to-impedance matrix. To define the impedance data, we need to choose a value $\eta \in \mathbb{R}_+$. In the wave scattering context, we often choose $\eta = k$ [3].

As in the previous section, we use $\boldsymbol{I}_{a\times a}$ to denote the identity matrix of shape $a \times a$ and $\boldsymbol{0}_d$ to denote a length-d vector filled with 0's. We also use $\boldsymbol{0}$ to denote a block filled with 0's, and assume the shape of this block can be inferred from the context.

B.1. Local solve stage

As before, there are $4p-4$ Chebyshev points on the boundary and $4q$ Gauss points on the boundary, and we must map between these two sets of discretization points. In particular, we need to precompute the following matrices:

- $\boldsymbol{P}$, with shape $4p-4 \times 4q$, is the operator mapping data sampled on the Gauss boundary points to data sampled on the $4p-4$ Chebyshev points located on the boundary of the leaf. This matrix is constructed using a barycentric Lagrange interpolation matrix mapping from Gauss to Chebyshev points on one side of the leaf with the final row deleted; this interpolation matrix is repeated for the other sides.

- $\boldsymbol{Q}$, with shape $4q \times 4p$, is the operator mapping data sampled on the Chebyshev points located on the boundary of the leaf to the Gauss points on the boundary of the leaf. This matrix is block-diagonal with four copies of a barycentric Lagrange interpolation matrix mapping from Chebyshev to Gauss points on one side of the leaf. Note this matrix double-counts the Chebyshev points at the corners of the leaf.
- $\boldsymbol{N}$, with shape $4p \times p^2$, is an operator mapping from interior solutions to outward-pointing normal derivative data evaluated on the boundary Chebyshev points. Note this operator counts each corner point twice. This matrix is formed by stacking relevant rows of Chebyshev spectral differentiation matrices.
- $\tilde{\boldsymbol{N}}$, with shape $4p - 4 \times p^2$, is an operator mapping from interior solutions to outward-pointing normal data evaluated on the boundary Chebyshev points. Note this operator counts each corner point only once. This matrix is formed by stacking relevant rows of Chebyshev spectral differentiation matrices.
- $\boldsymbol{H}$, with shape $4p \times p^2$, is an operator mapping from interior solutions to evaluations of outgoing impedance data on the Chebyshev boundary discretization points. This matrix is constructed by taking $\boldsymbol{N}$ and subtracting $i\eta \boldsymbol{I}_{p\times p}$ from the appropriate submatrices. Again, this matrix double-counts the Chebyshev discretization points at the corners of the leaf.
- $\boldsymbol{G}$, with shape $4p - 4 \times p^2$, is an operator mapping from interior solutions to evaluations of incoming impedance data on the Chebyshev boundary discretization points. This matrix is constructed by taking $\tilde{\boldsymbol{N}}$ and adding $i\eta \boldsymbol{I}_{p-1\times p-1}$ to the appropriate submatrices.

To work with $\boldsymbol{L}^{(i)}$, the discretization of the differential operator on leaf i, it is useful to identify I_i and I_e, the sets of discretization points corresponding to the $(p-2)^2$ interior and $4p-4$ exterior Chebyshev points, respectively. As in Appendix A, we solve the local problem by using these precomputed operators to enforce the differential operator on the interior discretization points and the boundary condition on the boundary discretization points. In this case, the boundary condition is an incoming impedance condition, also known as a Robin boundary condition. Now, we can fully describe the local solve stage in Algorithm 8.

Algorithm 8: 2D ItI local solve stage.

Input: Discretized differential operators $\{\boldsymbol{L}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; discretized source functions $\{\boldsymbol{f}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; precomputed interpolation and differentiation matrices $\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{H}$, and $\boldsymbol{G}$.

for $i = 1, \ldots, n_{leaves}$ **do**

$$\boldsymbol{B}^{(i)} = \begin{bmatrix} \boldsymbol{G} \\ \boldsymbol{L}^{(i)}(I_i, :) \end{bmatrix}$$

Invert $\boldsymbol{B}^{(i)}$

$$\boldsymbol{Y}^{(i)} = \left(\boldsymbol{B}^{(i)}\right)^{-1}(:, I_e)\boldsymbol{P}$$

$$\boldsymbol{v}^{(i)} = \left(\boldsymbol{B}^{(i)}\right)^{-1}(:, I_i)\boldsymbol{f}^{(i)}(I_i)$$

$$\boldsymbol{T}^{(i)} = \boldsymbol{Q}\boldsymbol{H}\boldsymbol{Y}^{(i)}$$

$$\boldsymbol{h}^{(i)} = \boldsymbol{Q}\boldsymbol{H}\boldsymbol{v}^{(i)}$$

Result: Poincaré–Steklov matrices $\{\boldsymbol{T}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; outgoing boundary data $\{\boldsymbol{h}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$; interior solution matrices $\{\boldsymbol{Y}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$, leaf-level particular solutions $\{\boldsymbol{v}^{(i)}\}_{i=1}^{n_{\text{leaves}}}$

B.2. Merge stage

As in Appendix A, we are merging four nodes $\Omega_a, \Omega_b, \Omega_c$, and Ω_d with boundary parts labeled $1, 2, \ldots, 8$. See Fig. A.12 for a diagram of the different boundary parts.

To implement the merge stage, we use sets of constraints to solve for a mapping from given $\boldsymbol{g}_{\text{ext}}$ to unknown $\boldsymbol{g}_{\text{int}}$. These are vectors tabulating incoming impedance data due to the homogeneous solution along the boundary parts. Because $\boldsymbol{g}_{\text{int}}$ tabulates impedance data, we must represent the incoming data with respect to neighboring nodes separately. For example, we must represent $\boldsymbol{g}_5^{(a)}$, the data along boundary element 5 incoming to node a, separately from $\boldsymbol{g}_5^{(b)}$. To that end, we use $\boldsymbol{g}_{\text{int}} = \left[\boldsymbol{g}_5^{(a)}, \boldsymbol{g}_8^{(a)}, \boldsymbol{g}_6^{(c)}, \boldsymbol{g}_7^{(c)}, \boldsymbol{g}_5^{(b)}, \boldsymbol{g}_6^{(b)}, \boldsymbol{g}_7^{(d)}, \boldsymbol{g}_8^{(d)}\right]^\top$. The ordering of these boundary elements is chosen specifically to reduce the computation during the merge, which will become apparent later. Again, we use n_{side} to denote the number of discretization points along each side of the nodes being merged, so $\boldsymbol{g}_{\text{int}}$ has length $8n_{\text{side}}$.

The first set of constraints specify that the solution to the PDE is continuous:

$$\boldsymbol{u}_{\text{ext}} = \boldsymbol{A}\boldsymbol{g}_{\text{ext}} + \boldsymbol{B}\boldsymbol{g}_{\text{int}} + \boldsymbol{h}_{\text{ext}}^{(\text{child})}. \tag{B.1}$$

In this equation $\boldsymbol{u}_{\text{ext}}$ is interpreted as the outgoing impedance data of the solution to the PDE restricted to the merged nodes with boundary data specified by $\boldsymbol{g}_{\text{ext}}$. In this set of constraints, we define:

$$\boldsymbol{h}_{\text{ext}}^{(\text{child})} = \begin{bmatrix} \boldsymbol{h}_1^{(a)} \\ \boldsymbol{h}_2^{(b)} \\ \boldsymbol{h}_3^{(c)} \\ \boldsymbol{h}_4^{(d)} \end{bmatrix}, \tag{B.2}$$

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{T}^{(a)}_{1,1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}^{(b)}_{2,2} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(c)}_{3,3} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(d)}_{4,4} \end{bmatrix}, \tag{B.3}$$

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{T}^{(a)}_{1,5} & \boldsymbol{T}^{(a)}_{1,8} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(b)}_{2,5} & \boldsymbol{T}^{(b)}_{2,6} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(c)}_{3,6} & \boldsymbol{T}^{(c)}_{3,7} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(d)}_{4,7} & \boldsymbol{T}^{(d)}_{4,8} \end{bmatrix}. \tag{B.4}$$

A second set of constraints specifies that the outgoing total solution's impedance data from one node must be opposite to the incoming homogeneous solution's impedance data for the neighboring node. For example, along merge interface 5, we enforce this constraint:

$$\boldsymbol{0}_{n_{\text{side}}} = \boldsymbol{u}^{(b)}_5 + \boldsymbol{g}^{(a)}_5. \tag{B.5}$$

In this equation, $\boldsymbol{u}^{(b)}_5$ is the outgoing impedance data due to the total solution of the PDE restricted to the merged nodes with boundary condition $\boldsymbol{g}_{\text{ext}}$. The normal derivative is oriented relative to node b. We can expand $\boldsymbol{u}^{(b)}_5$ to find:

$$\boldsymbol{0}_{n_{\text{side}}} = \boldsymbol{g}^{(a)}_5 + \boldsymbol{T}^{(a)}_{5,5}\boldsymbol{g}^{(a)}_5 + \boldsymbol{T}^{(a)}_{5,8}\boldsymbol{g}^{(a)}_8 + \boldsymbol{T}^{(b)}_{5,5}\boldsymbol{g}^{(b)}_5 + \boldsymbol{T}^{(b)}_{5,6}\boldsymbol{g}^{(b)}_6 + \boldsymbol{T}^{(a)}_{5,5}\boldsymbol{g}^{(a)}_5 + \boldsymbol{T}^{(a)}_{5,1}\boldsymbol{g}^{(a)}_1 + \boldsymbol{h}^{(a)}_5. \tag{B.6}$$

Similar equalities hold in each direction along each merge interface. We can expand to form a second system of constraints:

$$-\boldsymbol{h}^{(\text{child})}_{\text{int}} = \boldsymbol{C}\boldsymbol{g}_{\text{ext}} + \boldsymbol{D}\boldsymbol{g}_{\text{int}}, \tag{B.7}$$

where we define

$$\boldsymbol{h}^{(\text{child})}_{\text{int}} = \begin{bmatrix} \boldsymbol{h}^{(b)}_5 \\ \boldsymbol{h}^{(d)}_8 \\ \boldsymbol{h}^{(b)}_6 \\ \boldsymbol{h}^{(d)}_7 \\ \boldsymbol{h}^{(a)}_5 \\ \boldsymbol{h}^{(c)}_6 \\ \boldsymbol{h}^{(c)}_7 \\ \boldsymbol{h}^{(a)}_8 \end{bmatrix}, \tag{B.8}$$

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{T}^{(b)}_{5,2} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(d)}_{8,4} \\ \boldsymbol{0} & \boldsymbol{T}^{(b)}_{6,2} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(d)}_{7,4} \\ \boldsymbol{T}^{(a)}_{5,1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(c)}_{6,3} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(c)}_{7,3} & \boldsymbol{0} \\ \boldsymbol{T}^{(a)}_{8,1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}, \tag{B.9}$$

$$\boldsymbol{D} = \boldsymbol{I}_{8n_{\text{side}}\times 8n_{\text{side}}} + \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(b)}_{5,5} & \boldsymbol{T}^{(b)}_{5,6} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(d)}_{8,7} & \boldsymbol{T}^{(d)}_{8,8} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(b)}_{6,5} & \boldsymbol{T}^{(b)}_{6,6} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(d)}_{7,7} & \boldsymbol{T}^{(d)}_{7,8} \\ \boldsymbol{T}^{(a)}_{5,5} & \boldsymbol{T}^{(a)}_{5,8} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(c)}_{6,6} & \boldsymbol{T}^{(c)}_{6,7} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}^{(c)}_{7,6} & \boldsymbol{T}^{(c)}_{7,7} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{T}^{(a)}_{8,5} & \boldsymbol{T}^{(a)}_{8,8} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}. \tag{B.10}$$

$\boldsymbol{D}$ has a special structure which allows us to efficiently compute $\boldsymbol{D}^{-1}$ via Schur complement methods. Note that we can re-write Eq. (B.10) as a block matrix with 2×2 blocks:

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{I}_{4n_{\text{side}}\times 4n_{\text{side}}} & \boldsymbol{D}_{12} \\ \boldsymbol{D}_{21} & \boldsymbol{I}_{4n_{\text{side}}\times 4n_{\text{side}}} \end{bmatrix}. \tag{B.11}$$

This structure allows us to construct $\boldsymbol{W} = \boldsymbol{I}_{4n_{\text{side}}\times 4n_{\text{side}}} - \boldsymbol{D}_{12}\boldsymbol{D}_{21}$, the Schur complement of the lower-right $\boldsymbol{I}_{4n_{\text{side}}\times 4n_{\text{side}}}$ block in $\boldsymbol{D}$; this is the only matrix we need to invert to compute $\boldsymbol{D}^{-1}$:

$$\boldsymbol{D}^{-1} = \begin{bmatrix} \boldsymbol{W}^{-1} & -\boldsymbol{W}^{-1}\boldsymbol{D}_{12} \\ -\boldsymbol{D}_{21}\boldsymbol{W}^{-1} & \boldsymbol{I}_{4n_{\text{side}}\times 4n_{\text{side}}} + \boldsymbol{D}_{21}\boldsymbol{W}^{-1}\boldsymbol{D}_{12} \end{bmatrix}. \tag{B.12}$$

We use Eq. (B.12) to compute $\boldsymbol{D}^{-1}$ and then construct the outputs of the merge stage using Eq. (4) and (5).

Appendix C. Full algorithms for 3D problems using DtN matrices with a uniform discretization

In this section, we describe the details for the three-dimensional version of our method which merges DtN matrices. In this version of the algorithm, the outgoing boundary data $\boldsymbol{h}$ tabulates the outward-pointing normal derivative of the particular solution, and the incoming boundary data $\boldsymbol{g}$ tabulates the homogenous solution values restricted to patch boundaries. $\boldsymbol{T}$ is a Dirichlet-to-Neumann matrix.

As in the previous section, we use $\boldsymbol{I}_{a\times a}$ to denote the identity matrix of shape $a \times a$ and $\mathbf{0}_d$ to denote a length-d vector filled with 0's. We also use $\mathbf{0}$ to denote a matrix block filled with 0's, and assume the shape of this block can be inferred from its context.

C.1. Local solve stage

Recall from Section 3.1 that we use a tensor product of order-p Chebyshev–Lobatto points to discretize the interior of each leaf. This results in a grid with p^3 discretization points, and $p^3 - (p-2)^3$ of these points lie on the boundary of the leaf. We use order-q Gauss–Legendre points to discretize each side of the leaf's boundary, so there are $6q^2$ boundary points in total. Thus, to translate the information between the interior and boundary of each leaf, we need to compute spectral differentiation matrices and matrices interpolating between the p^2 Chebyshev and q^2 Gauss points on each face of the leaf. In particular, we need to precompute the following matrices:

- $\boldsymbol{P}$, with shape $p^3 - (p-2)^3 \times 6q^2$, is the operator mapping data sampled on the Gauss boundary points to data sampled on the Chebyshev points located on the boundary of the leaf. This matrix is constructed using a barycentric Lagrange interpolation matrix mapping from Gauss to Chebyshev points on one face of the leaf; this interpolation matrix is repeated for the other sides. Rows corresponding to the Chebyshev points on the corners (edges) of the leaf average the contribution from the three (two) adjoining panels.
- $\boldsymbol{Q}$ with shape $6q^2 \times p^3$, performs spectral differentiation on the p^3 Chebyshev points followed by interpolation to the Gauss boundary points. This matrix is formed by stacking the relevant rows of Chebyshev spectral differentiation matrices to form an operator which evaluates normal derivatives on the boundary Chebyshev points, and then composing this differentiation operator with a matrix formed from barycentric Lagrange interpolation matrix blocks. These interpolation matrices each map from one Chebyshev face to one Gauss face.

To work with $\boldsymbol{L}^{(i)}$, the discretization of the differential operator on leaf i, it is useful to identify I_i and I_e, the sets of discretization points corresponding to the $(p-2)^3$ interior and $p^3 - (p-2)^3$ exterior Chebyshev points, respectively. Once the precomputed operators and index sets are correctly specified, Algorithm 7 can be re-used for the three-dimensional case.

C.2. Merge stage

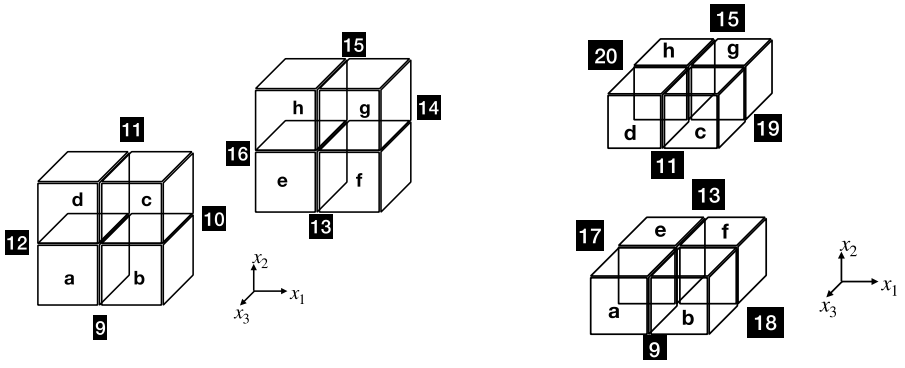


Fig. C.13. Visualizing boundary elements 9 through 20 for three-dimensional merges.

In the 3D merge stage, we are merging eight nodes $\Omega_a, \Omega_b, \Omega_c, \Omega_d, \Omega_e, \Omega_f, \Omega_g$, and Ω_h, which have *exterior* and *interior* discretization points. We label the exterior boundary sections $1, 2, \ldots, 8$, and we label the interior boundary sections $9, 10, \ldots, 20$. See Fig. C.13 for a diagram of the different boundary parts. Because the merge stage operates completely on data discretized using Gauss–Legendre panels, there are no discretization points at the corners or edges of nodes. This means each discretization point belongs to exactly one part of the boundary. During this stage of the algorithm, we will be indexing rows and columns of the Dirichlet-to-Neumann matrices according to boundary sections $1, \ldots, 20$. For example, we use $\boldsymbol{T}^{(a)}_{1,9}$ to indicate the submatrix of node a's DtN matrix which maps from boundary section 9 to boundary section 1. Suppose that each node has n_{side} discretization points on each face. Then $\boldsymbol{T}^{(a)}_{1,9}$ will have shape $3n_{\text{side}} \times n_{\text{side}}$.

Just as in the two-dimensional case, we use sets of constraints to solve for a mapping from given $\boldsymbol{g}_{\text{ext}}$ to unknown $\boldsymbol{g}_{\text{int}}$, vectors tabulating the homogeneous solution along the boundary parts. First are constraints specifying that the solution to the PDE is continuous:

$$\boldsymbol{u}_{\text{ext}} = \boldsymbol{A}\boldsymbol{g}_{\text{ext}} + \boldsymbol{B}\boldsymbol{g}_{\text{int}} + \boldsymbol{h}_{\text{ext}}^{(\text{child})}. \tag{C.1}$$

In this equation $\boldsymbol{u}_{\text{ext}}$ is interpreted as the outward-pointing normal derivative of the solution to the PDE restricted to the merged nodes with boundary data specified by $\boldsymbol{g}_{\text{ext}}$. In this set of constraints, we define:

$$\boldsymbol{h}_{\text{ext}}^{(\text{child})} = \begin{bmatrix} \boldsymbol{h}_1^{(a)} \\ \boldsymbol{h}_2^{(b)} \\ \boldsymbol{h}_3^{(c)} \\ \boldsymbol{h}_4^{(d)} \\ \boldsymbol{h}_5^{(e)} \\ \boldsymbol{h}_6^{(f)} \\ \boldsymbol{h}_7^{(g)} \\ \boldsymbol{h}_8^{(h)} \end{bmatrix}, \tag{C.2}$$

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{T}_{1,1}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_{2,2}^{(b)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{3,3}^{(c)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{4,4}^{(d)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{5,5}^{(e)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{6,6}^{(f)} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{7,7}^{(g)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{8,8}^{(h)} \end{bmatrix}, \tag{C.3}$$

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{T}_{1,9}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{1,12}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{1,17}^{(a)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{T}_{2,9}^{(b)} & \boldsymbol{T}_{2,10}^{(b)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{2,18}^{(b)} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T}_{3,10}^{(c)} & \boldsymbol{T}_{3,11}^{(c)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{3,19}^{(c)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{4,11}^{(d)} & \boldsymbol{T}_{4,12}^{(d)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{4,20}^{(d)} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{5,13}^{(e)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{5,16}^{(e)} & \boldsymbol{T}_{5,17}^{(e)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{6,13}^{(f)} & \boldsymbol{T}_{6,14}^{(f)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{6,18}^{(f)} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{7,14}^{(g)} & \boldsymbol{T}_{7,15}^{(g)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{7,19}^{(g)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{8,15}^{(h)} & \boldsymbol{T}_{8,16}^{(h)} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T}_{8,20}^{(h)} \end{bmatrix}. \tag{C.4}$$

As in the two-dimensional case, we enforce constraints that ensure the normal derivatives from neighboring nodes sum to zero, which gives us a system of constraints:

$$\boldsymbol{0}_{12n_{\text{side}}} = \boldsymbol{C}\boldsymbol{g}_{\text{ext}} + \boldsymbol{D}\boldsymbol{g}_{\text{int}} + \boldsymbol{h}_{\text{int}}^{(\text{child})}, \tag{C.5}$$

where we define

$$\boldsymbol{h}_{\text{int}}^{(\text{child})} = \begin{bmatrix} \boldsymbol{h}_9^{(a)} + \boldsymbol{h}_9^{(b)} \\ \boldsymbol{h}_{10}^{(b)} + \boldsymbol{h}_{10}^{(c)} \\ \boldsymbol{h}_{11}^{(c)} + \boldsymbol{h}_{11}^{(d)} \\ \boldsymbol{h}_{12}^{(d)} + \boldsymbol{h}_{12}^{(a)} \\ \boldsymbol{h}_{13}^{(e)} + \boldsymbol{h}_{13}^{(f)} \\ \boldsymbol{h}_{14}^{(f)} + \boldsymbol{h}_{14}^{(g)} \\ \boldsymbol{h}_{15}^{(g)} + \boldsymbol{h}_{15}^{(h)} \\ \boldsymbol{h}_{16}^{(h)} + \boldsymbol{h}_{16}^{(e)} \\ \boldsymbol{h}_{17}^{(a)} + \boldsymbol{h}_{17}^{(e)} \\ \boldsymbol{h}_{18}^{(b)} + \boldsymbol{h}_{18}^{(f)} \\ \boldsymbol{h}_{19}^{(c)} + \boldsymbol{h}_{19}^{(g)} \\ \boldsymbol{h}_{20}^{(d)} + \boldsymbol{h}_{20}^{(h)} \end{bmatrix}, \tag{C.6}$$

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{T}^{(a)}_{9,1} & \boldsymbol{T}^{(b)}_{9,2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{T}^{(b)}_{10,2} & \boldsymbol{T}^{(c)}_{10,3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(c)}_{11,3} & \boldsymbol{T}^{(d)}_{11,4} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{T}^{(a)}_{12,1} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(d)}_{12,4} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(e)}_{13,5} & \boldsymbol{T}^{(f)}_{13,6} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(f)}_{14,6} & \boldsymbol{T}^{(g)}_{14,7} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(g)}_{15,7} & \boldsymbol{T}^{(h)}_{15,8} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(e)}_{16,5} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(h)}_{16,8} \\ \boldsymbol{T}^{(a)}_{17,1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(e)}_{17,5} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{T}^{(b)}_{18,2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(f)}_{18,6} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(c)}_{19,3} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(g)}_{19,7} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(d)}_{20,4} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(h)}_{20,8} \end{bmatrix}, \tag{C.7}$$

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{T}^{(a)}_{9,9} + \boldsymbol{T}^{(b)}_{9,9} & \boldsymbol{T}^{(b)}_{9,10} & \mathbf{0} & \boldsymbol{T}^{(a)}_{9,12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(a)}_{9,17} & \boldsymbol{T}^{(b)}_{9,18} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{T}^{(b)}_{10,9} & \boldsymbol{T}^{(b)}_{10,10} + \boldsymbol{T}^{(c)}_{10,10} & \boldsymbol{T}^{(c)}_{10,11} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(b)}_{10,18} & \boldsymbol{T}^{(c)}_{10,19} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{T}^{(c)}_{11,10} & \boldsymbol{T}^{(c)}_{11,11} + \boldsymbol{T}^{(d)}_{11,11} & \boldsymbol{T}^{(d)}_{11,12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(c)}_{11,19} & \boldsymbol{T}^{(d)}_{11,20} \\ \boldsymbol{T}^{(a)}_{12,9} & \mathbf{0} & \boldsymbol{T}^{(d)}_{12,11} & \boldsymbol{T}^{(d)}_{12,12} + \boldsymbol{T}^{(a)}_{12,12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(a)}_{12,17} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(d)}_{12,20} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(e)}_{13,13} + \boldsymbol{T}^{(f)}_{13,13} & \boldsymbol{T}^{(f)}_{13,14} & \mathbf{0} & \boldsymbol{T}^{(e)}_{13,16} & \boldsymbol{T}^{(e)}_{13,17} & \boldsymbol{T}^{(f)}_{13,18} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(f)}_{14,13} & \boldsymbol{T}^{(f)}_{14,14} + \boldsymbol{T}^{(g)}_{14,14} & \boldsymbol{T}^{(g)}_{14,15} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(f)}_{14,18} & \boldsymbol{T}^{(g)}_{14,19} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(g)}_{15,14} & \boldsymbol{T}^{(g)}_{15,15} + \boldsymbol{T}^{(h)}_{15,15} & \boldsymbol{T}^{(h)}_{15,16} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(g)}_{15,19} & \boldsymbol{T}^{(h)}_{15,20} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(e)}_{16,13} & \mathbf{0} & \boldsymbol{T}^{(h)}_{16,15} & \boldsymbol{T}^{(h)}_{16,16} + \boldsymbol{T}^{(e)}_{16,16} & \boldsymbol{T}^{(e)}_{16,17} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(h)}_{16,20} \\ \boldsymbol{T}^{(a)}_{17,9} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(a)}_{17,12} & \boldsymbol{T}^{(e)}_{17,13} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(e)}_{17,16} & \boldsymbol{T}^{(a)}_{17,17} + \boldsymbol{T}^{(e)}_{17,17} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{T}^{(b)}_{18,9} & \boldsymbol{T}^{(b)}_{18,10} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(f)}_{18,13} & \boldsymbol{T}^{(f)}_{18,14} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(b)}_{18,18} + \boldsymbol{T}^{(f)}_{18,18} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{T}^{(c)}_{19,10} & \boldsymbol{T}^{(c)}_{19,11} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(g)}_{19,14} & \boldsymbol{T}^{(g)}_{19,15} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(c)}_{19,19} + \boldsymbol{T}^{(g)}_{19,19} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(d)}_{20,11} & \boldsymbol{T}^{(d)}_{20,12} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(h)}_{20,15} & \boldsymbol{T}^{(h)}_{20,16} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{T}^{(d)}_{20,20} + \boldsymbol{T}^{(h)}_{20,20} \end{bmatrix} \tag{C.8}$$

Now that the matrices and vectors are defined, we can construct the linear system in Eq. (3) and compute the merged data.

Appendix D. Details for the inverse scattering experiment

We consider a forward model which composes $\mathcal{F}$, which maps a scattering potential to evaluations of a scattered wave, and $\mathcal{B}$, which maps coefficients of a sine series to scattering potentials:

$$(\mathcal{F}\circ\mathcal{B})(\theta)_j = u_\theta(x^{(j)}); \qquad u_\theta \text{ solves Equation 9 with } q = q_\theta; \tag{D.1}$$

$$q_\theta(x) = \mathcal{B}(\theta)(x) = \sum_{b_j \in B_\gamma} \theta_j b_j(x); \tag{D.2}$$

where the basis B_γ is specified by Eq. (12). In this experiment, we use a domain $\Omega = [-1, 1]$, and we use 100 evaluation points equispaced on a ring of radius 5:

$$x^{(j)} = \left(5\sin\left(\frac{2\pi j}{100}\right), 5\cos\left(\frac{2\pi j}{100}\right)\right), \qquad j = 0, \dots, 99. \tag{D.3}$$

To solve an inverse problem, we are interested in computing the Fréchet derivative of $\mathcal{F}\circ\mathcal{B}$ centered at θ; we call this object $J[\theta]$. By the chain rule, we can decompose this Fréchet derivative

$$J[\theta] = J_{\mathcal{F}}[q_\theta]\circ J_{\mathcal{B}}[\theta].$$

The basis transformation $\mathcal{B}$ is linear, so the action of $J_{\mathcal{B}}[\theta]$ can be computed with standard sine and adjoint sine transforms. The following section will describe how we compute the action of $J_{\mathcal{F}}[q_\theta]$.

D.1. Defining the action of the Fréchet derivative

Borges et al. [38] describe the Fréchet derivative $J_{\mathcal{F}}[q_\theta]$ the action $J_{\mathcal{F}}[q_\theta]v$ and $J_{\mathcal{F}}[q_\theta]^* f$. The action of the derivative and its adjoint can be described by the solution of elliptic partial differential equations. We re-state these results in this section.

Theorem 1 (Theorem 3.1 of Borges et al. [38]). *Let u_θ solve Eq. (9) with scattering potential $q = q_\theta$. Let w solve*

$$\begin{cases} \Delta w(x) + k^2(1 + q(x))w(x) = k^2 v(x)\left(u_\theta(x) + e^{ik\langle \hat{s}, x\rangle}\right), & x \in [-1, 1]^2, \\ \sqrt{r}\left(\frac{\partial w}{\partial r} - ikw\right) \to 0, & r = \|x\|_2 \to \infty. \end{cases} \tag{D.4}$$

Then

$$\left(J_{\mathcal{F}}[q_\theta]v\right)_j = w(x^{(j)}).$$

Theorem 2 (Theorem 3.2 of Borges et al. [38]). *Let u_θ solve Eq. (9) with scattering potential $q = q_\theta$. Let f denote a singular charge distribution supported on the evaluation points $\{x^{(j)}\}_{j=0,\dots,99}$ viewed as a generalized function in $\mathbb{R}^2$. Let w solve*

$$\begin{cases} \Delta w(x) + k^2(1 + q(x))w(x) = k^2 f(x), & x \in \mathbb{R}^2, \\ \sqrt{r}\left(\frac{\partial w}{\partial r} + ikw\right) \to 0, & r = \|x\|_2 \to \infty. \end{cases} \tag{D.5}$$

Then

$$\left(J_{\mathcal{F}}[q_\theta]^* f\right)(x) = w(x)\overline{\left(u_\theta(x) + e^{ik\langle \hat{s}, x\rangle}\right)}.$$

D.2. Experiment details

In this section, we describe the experimental setting used to generate Fig. 9a. In these experiments, we use $k = 20$ and basis B_γ with $\gamma = 25$, which gives us $N_\theta = 465$ basis coefficients. We compute a coefficient vector θ by projecting the scattering potential Eq. (10) onto B_γ.

To measure the accuracy of the outputs of JAX Jacobian-vector products, we generate a random coefficient vector δ distributed i.i.d. Gaussian and compute $J[\theta]\delta$. We compute this Jacobian-vector product for a range of discretization sizes, holding $p = 16$ constant and varying $L = 1, \dots, 5$. We compare this with the action of the Fréchet derivative, which is computed by first computing $J_B[\theta]\delta$ and then applying $J_{\mathcal{F}}[q_\theta]$ from Theorem 1, which is computed using parameters $p = 16$ and $L = 5$. We measure the relative ℓ_∞ error between the outputs of JAX Jacobian-vector products and the action of the Fréchet derivative.

To measure the accuracy of the outputs of JAX vector-Jacobian products, we generate a random perturbation $f \in \mathbb{C}^{100}$. The real and imaginary parts of each component of f are distributed i.i.d. Gaussian. We compute this vector-Jacobian product for a range of discretization sizes, again holding $p = 16$ constant and varying $L = 1, \dots, 5$. We compare this with the action of the Fréchet derivative which is computed by first evaluating $J_{\mathcal{F}}[q_\theta]^* f$ via Theorem 2, which is computed using parameters $p = 16$ and $L = 5$, and then applying $J_B[\theta]^*$. We measure the relative ℓ_∞ error betwen the outputs of JAX vector-Jacobian products and the action of the Fréchet derivative.

We note the choice of evaluation points exterior to the computational domain (Eq. (D.3)) is not necessary for the convergence of automatic differentiation; we choose these evaluation points to ensure numerical stability when computing $J_{\mathcal{F}}[q_\theta]^* f$. In preliminary experiments, we observed accurate vector-Jacobian products when the evaluation points were located at the HPS discretization points. In this setting, the inputs of the vector-Jacobian product routine must be scaled by the appropriate quadrature weights.

Appendix E. Additional timing results

In this appendix, we extend the results shown in Fig. 3 to include our subtree recomputation method evaluated with different subtree depths. As a hueristic, we choose to use the subtree recomputation depth to be the maximum depth where all computations for Algorithms 1 and 2 can fit on a single GPU. In Fig. 3, we consider the DtN version of the method with polynomial order $p = 16$; this results in subtree depth 7. In Table E.3, we show the results of choosing different subtree depth parameters. We measure the runtime for large problem sizes $L = 8$ and 9; recomputation is necessary for these problem sizes.

As we decrease the subtree depth, we see runtimes increase, as more transfers between the GPU and host RAM are required.

Table E.3
Evaluating the effect of the subtree height parameter.

Subtree depth	L	N	Runtime (s)	% of Peak FLOPS
5	8	16,777,216	5.86	6.68 %
5	9	67,108,864	24.50	10.86 %
6	8	16,777,216	4.35	10.58 %
6	9	67,108,864	18.85	15.59 %
7	8	16,777,216	4.02	14.86 %
7	9	67,108,864	17.43	20.01 %

References

[1] P. Geldermans, A. Gillman, An adaptive high order direct solution technique for elliptic boundary value problems, SIAM J. Sci. Comput. 41 (1) (2019) A292–A315. Publisher: Society for Industrial and Applied Mathematics, https://doi.org/10.1137/17M1156320

[2] P.-G. Martinsson, Fast Direct Solvers for Elliptic PDEs, Society for Industrial and Applied Mathematics, Philadelphia, PA, Philadelphia, PA, 2019. https://doi.org/10.1137/1.9781611976045

[3] A. Gillman, A.H. Barnett, P.-G. Martinsson, et al., A spectrally accurate direct solution technique for frequency-domain scattering problems with variable media, BIT Numer. Math. 55 (1) (2015) 141–170. https://doi.org/10.1007/s10543-014-0499-8

[4] O.G. Ernst, M.J. Gander, Why it is difficult to solve Helmholtz problems with classical iterative methods, in: I.G. Graham, T.Y. Hou, O. Lakkis, R. Scheichl (Eds.), Numerical Analysis of Multiscale Problems, Springer, Berlin, Heidelberg, 2012, pp. 325–363. https://doi.org/10.1007/978-3-642-22061-6_10

[5] P.G. Martinsson, A direct solver for variable coefficient elliptic PDEs discretized via a composite spectral collocation method, J. Comput. Phys. 242 (2013) 460–479. https://doi.org/10.1016/j.jcp.2013.02.019

[6] A. Gillman, P.G. Martinsson, A direct solver with $O(N)$ complexity for variable coefficient elliptic PDEs discretized via a high-Order composite spectral collocation method, SIAM J. Sci. Comput. 36 (4) (2014) A2023–A2046. Publisher: Society for Industrial and Applied Mathematics, https://doi.org/10.1137/130918988

[7] D.A. Kopriva, A staggered-grid multidomain spectral method for the compressible Navier-Stokes equations, J. Comput. Phys. 143 (1) (1998) 125–158. https://doi.org/10.1006/jcph.1998.5956

[8] H.P. Pfeiffer, L.E. Kidder, M.A. Scheel, S.A. Teukolsky, et al., A multidomain spectral method for solving elliptic equations, Comput. Phys. Commun. 152 (3) (2003) 253–273. https://doi.org/10.1016/S0010-4655(02)00847-0

[9] B. Yang, J.S. Hesthaven, Multidomain pseudospectral computation of Maxwell's equations in 3-d general curvilinear coordinates, Appl. Numer. Math. 33 (1) (2000) 281–289. https://doi.org/10.1016/S0168-9274(99)00094-X

[10] A. George, Nested dissection of a regular finite element mesh, SIAM J. Numer. Anal. 10 (2) (1973) 345–363. Publisher: Society for Industrial and Applied Mathematics, https://doi.org/10.1137/0710032

[11] K.L. Ho, L. Greengard, A fast direct solver for structured linear systems by recursive skeletonization, SIAM J. Sci. Comput. 34 (5) (2012) A2507–A2532. Publisher: Society for Industrial and Applied Mathematics, https://doi.org/10.1137/120866683

[12] T. Beck, Y. Canzani, J.L. Marzuola, et al., Quantitative bounds on impedance-to-impedance operators with applications to fast direct solvers for PDEs, Pure Appl. Anal. 4 (2) (2022) 225–256. Publisher: Mathematical Sciences Publishers, https://msp.org/paa/2022/4-2/p02.xhtml. https://doi.org/10.2140/paa.2022.4.225

[13] J.P. Lucero Lorca, N. Beams, D. Beecroft, A. Gillman, et al., An iterative solver for the HPS discretization applied to three dimensional Helmholtz problems, SIAM J. Sci. Comput. 46 (1) (2024) A80–A104. Publisher: Society for Industrial and Applied Mathematics, https://epubs.siam.org/doi/full/10.1137/21M1463380. https://doi.org/10.1137/21M1463380

[14] S. Hao, P.-G. Martinsson, A direct solver for elliptic PDEs in three dimensions based on hierarchical merging of Poincaré-Steklov operators, J. Comput. Appl. Math. 308 (2016) 419–434. https://doi.org/10.1016/j.cam.2016.05.013

[15] J. Kump, A. Yesypenko, P.-G. Martinsson, A two-level direct solver for the hierarchical Poincaré–Steklov method (arXiv:2503.04033) (2025). arXiv:2503.04033 [math], https://doi.org/10.48550/arXiv.2503.04033

[16] D. Fortunato, N. Hale, A. Townsend, The ultraspherical spectral element method, J. Comput. Phys. 436 (2021) 110087. https://doi.org/10.1016/j.jcp.2020.110087

[17] D. Fortunato, A high-order fast direct solver for surface PDEs, SIAM J. Sci. Comput. 46 (4) (2024) A2582–A2606. Publisher: Society for Industrial and Applied Mathematics, https://doi.org/10.1137/22M1525259

[18] N.N. Beams, A. Gillman, R.J. Hewett, A parallel shared-memory implementation of a high-order accurate solution technique for variable coefficient Helmholtz problems, Comput. Math. Appl. 79 (4) (2020) 996-1011. https://doi.org/10.1016/j.camwa.2019.08.019

[19] S. Georgescu, P. Chow, H. Okuda, et al., GPU Acceleration for FEM-Based structural analysis, Arch. Comput. Methods Eng. 20 (2) (2013) 111–121. https://doi.org/10.1007/s11831-013-9082-8

[20] A. Abdelfattah, V. Barra, N. Beams, R. Bleile, J. Brown, J.-S. Camier, R. Carson, N. Chalmers, V. Dobrev, Y. Dudouit, P. Fischer, A. Karakus, S. Kerkemeier, T. Kolev, Y.-H. Lan, E. Merzari, M. Min, M. Phillips, T. Rathnayake, R. Rieben, T. Stitt, A. Tomboulides, S. Tomov, V. Tomov, A. Vargas, T. Warburton, K. Weiss, GPU Algorithms for efficient exascale discretizations, Parallel Comput. 108 (2021) 102841. https://doi.org/10.1016/j.parco.2021.102841

[21] T. Kolev, P. Fischer, M. Min, J. Dongarra, J. Brown, V. Dobrev, T. Warburton, S. Tomov, M.S. Shephard, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, N. Chalmers, Y. Dudouit, A. Karakus, I. Karlin, S. Kerkemeier, Y.-H. Lan, D. Medina, E. Merzari, A. Obabko, W. Pazner, T. Rathnayake, C.W. Smith, L. Spies, K. Swirydowicz, J. Thompson, A. Tomboulides, V. Tomov, Efficient exascale discretizations: high-order finite element methods, Int. J. High Perform Comput. Appl. 35 (6) (2021) 527-552. https://doi.org/10.1177/10943420211020803

[22] A. Abdelfattah, P. Ghysels, W. Boukaram, S. Tomov, X.S. Li, J. Dongarra, et al., Addressing irregular patterns of matrix computations on GPUs and their impact on applications powered by sparse direct solvers, in: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, Dallas, TX, USA, 2022, pp. 1–14. https://ieeexplore.ieee.org/document/10046092/. https://doi.org/10.1109/SC41404.2022.00031

[23] P. Ghysels, R. Synk, High performance sparse multifrontal solvers on modern GPUs, Parallel Comput. 110 (2022) 102897. https://doi.org/10.1016/j.parco.2022.102897

[24] X.S. Li, J.W. Demmel, SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Softw. 29 (2) (2003) 110–140. https://doi.org/10.1145/779359.779361

[25] A. Yesypenko, P.-G. Martinsson, GPU Optimizations for the hierarchical Poincaré–Steklov scheme, in: Z. Dostál, T. Kozubek, A. Klawonn, U. Langer, L.F. Pavarino, J. Šístek, O.B. Widlund (Eds.), Domain Decomposition Methods in Science and Engineering XXVII, Springer Nature Switzerland, Cham, 2024, pp. 519–528. https://doi.org/10.1007/978-3-031-50769-4_62

[26] A. Yesypenko, P.-G. Martinsson, SlabLU: a two-level sparse direct solver for elliptic PDEs, Adv. Comput. Math. 50 (4) (2024) 90. https://doi.org/10.1007/s10444-024-10176-x

[27] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C.K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M.Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, S. Chintala, PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation, 2024, 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), https://pytorch.org/assets/pytorch2-2.pdf. https://doi.org/10.1145/3620665.3640366

[28] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: Composable transformations of Python + NumPy programs, 2018, http://github.com/jax-ml/jax.

[29] C. Leary, T. Wang, XLA: TensorFlow, Compiled!, 2017,

[30] K. Diao, Z. Li, R.D.P. Grumitt, Y. Mao, synax: A differentiable and GPU-accelerated synchrotron simulation package, Astrophys. J. Suppl. Ser. 278 (2025) 25. https://doi.org/10.3847/1538-4365/adc5ff

[31] T. Xue, S. Liao, Z. Gan, C. Park, X. Xie, W.K. Liu, J. Cao, et al., JAX-FEM: A differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science, Comput. Phys. Commun. 291 (2023) 108802. https://doi.org/10.1016/j.cpc.2023.108802

[32] P. Kidger, On Neural Differential Equations, PhD Thesis, University of Oxford, 2021.

[33] P.G. Martinsson, The hierarchical Poincaré-Steklov (HPS) solver for elliptic PDEs: A tutorial, 2015, http://arxiv.org/abs/1506.01308.

[34] L.N. Trefethen, Spectral Methods in MATLAB, Software, Environments, and Tools, Society for Industrial and Applied Mathematics, 2000. https://doi.org/10.1137/1.9780898719598

[35] D. Chipman, D. Calhoun, C. Burstedde, et al., A fast direct solver for elliptic PDEs on a hierarchy of adaptively refined quadtrees, 2024, http://arxiv.org/abs/2402.14936.

[36] T. Dao, D. Fu, S. Ermon, A. Rudra, C. Ré, Flashattention: fast and memory-efficient exact attention with IO-awareness, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems, 35, Curran Associates, Inc., 2022, p. 16344-16359. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

[37] A. Gu, T. Dao, Mamba: linear-time sequence modeling with selective state spaces, 2024. https://openreview.net/forum?id=tEYskw1VY2#discussion.

[38] C. Borges, A. Gillman, L. Greengard, High resolution inverse scattering in two dimensions using recursive linearization, SIAM J. Imag. Sci. 10 (2) (2017) 641-664. https://doi.org/10.1137/16M1093562

[39] T. Askham, M. Rachh, M. O'Neil, J. Hoskins, D. Fortunato, S. Jiang, F. Fryklund, T. Goodwill, H.Y. Wang, H. Zhu, `chunkIE`: A MATLAB integral equation toolbox, 2024, https://github.com/fastalgorithms/chunkie.

[40] C.C. Paige, M.A. Saunders, LSQR: An algorithm for sparse linear equations and sparse least squares, ACM Trans. Math. Softw. 8 (1) (1982) 43–71. https://doi.org/10.1145/355984.355989

[41] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K.J. Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C.J. Carey, İ. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R.

Cimrman, I. Henriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors, Scipy 1.0: fundamental algorithms for scientific computing in python, Nat. Methods 17 (2020) 261–272. https://doi.org/10.1038/s41592-019-0686-2

[42] J.A. Grant, B.T. Pickup, A. Nicholls, et al., A smooth permittivity function for Poisson-Boltzmann solvation methods, J. Comput. Chem. 22 (6) (2001) 608–640. https://doi.org/10.1002/jcc.1032

[43] F. Vico, L. Greengard, M. Ferrando, et al., Fast convolution with free-space Green's functions, J. Comput. Phys. 323 (2016) 191–203. https://doi.org/10.1016/j.jcp.2016.07.028

[44] A. Nicholls, B. Honig, A rapid finite difference algorithm, utilizing successive over-relaxation to solve the Poisson-Boltzmann equation, J. Comput. Chem. 12 (4) (1991) 435–445. https://doi.org/10.1002/jcc.540120405

[45] J. Colmenares, J. Ortiz, W. Rocchia, et al., GPU linear and non-linear Poisson-Boltzmann solver module for delphi, Bioinformatics 30 (4) (2014) 569–570. https://doi.org/10.1093/bioinformatics/btt699

[46] O. Melia, D. Fortunato, J. Hoskins, R. Willett, jaxhps: An elliptic PDE solver built with machine learning in mind, J. Open Source Software 10 (115) (2025) 8549. https://doi.org/10.21105/joss.08549