A Novel Framework for GPU-Parallelized Peridynamics with Applications in Nonlinear Elasticity and Topology Optimization

John Bartlett

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2022

Reading Committee:

Duane Storti, Chair

Mark Ganter

Eric Rombokas

Program Authorized to Offer Degree:

Mechanical Engineering

University of Washington

Abstract

A Novel Framework for GPU-Parallelized Peridynamics with
Applications in Nonlinear Elasticity and Topology Optimization

John Bartlett

Chair of the Supervisory Committee:
Dr. Duane Storti
Mechanical Engineering

This work focuses on advancing the state-of-the-art of the peridynamic simulation method. Peridynamics is a framework for modelling continuum mechanics, similar to the finite element method (FEM). The peridynamic formulation, which involves a meshfree discretization of a continuum into material points and evaluation of force-exerting interactions between these points, allows representation of complex physical phenomena such as nonlinear mechanics, crack propagation, and material failure more naturally than gradient-based approaches like FEM. This work presents a three-pronged approach towards advancing the state-of-the-art of peridynamic simulation: Chapter 2 definitively improves treatment of the long-standing "peridynamic surface effect", which limits the theoretical accuracy of the peridynamic method due to the method's formulation assuming all material points to be in the bulk of the body. The solution developed here extends an approach which eliminates the effect in simple geometries to apply a specialized boundary treatment to arbitrary domain shapes. Chapter 3 develops a novel, high performance software package for parallelized peridynamic simulation on graphics processing units (GPUs). This approach is several hundred times faster than equivalent serial approaches, and several times faster than alternative GPU-based approaches recently developed by other researchers. Additionally, this software is capable of simulating problem sizes an order of magnitude larger than competing methods

on limited GPU memory. Finally, Chapter 4 applies this highly-efficient software towards novel applications of peridynamics, particularly in nonlinear structural analysis and large scale topology optimization. Specifically, the GPU peridynamic framework is capable of evaluating large deformations 2-3 orders of magnitude faster than state-of-the-art nonlinear FEM solvers. The framework is demonstrated to produce accurate simulations of behavior in auxetic structures exhibiting geometric discontinuities and nonlinear deformations, a use case which demands nonlinear solution for reasonable results; the peridynamic solver runs 1250x faster than the GPU-accelerated commercial software ANSYS. Finally, utilizing the GPU framework for the structural analysis step of large-scale topology optimization problems produces optimal topologies 35-65x faster than state-of-the-art FEM optimizers run on the CPU, and 2-4x faster than GPU-based FEM optimizers running on identical hardware.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to University of Washington & the Mechanical Engineering Department, where he has had the opportunity to pursue these studies, and to his advisor Dr. Duane Storti for his support in this work.

Chapter 1

INTRODUCTION

1.1 Overview

Continuum mechanics is a broad branch of mechanical analysis dealing with the behavior of physical bodies under external conditions such as applied forces, displacements, and temperature changes. As opposed to other branches of mechanics which treat objects as discrete particles, continuum mechanics evaluates bodies as continuous, volume-occupying media [1]. Applications of continuum mechanics arise in a wide variety of common engineering problems, from structural analysis to fluid dynamics.

Typically, classical continuum mechanics analysis is performed by developing partial differential equations to describe the relevant physics in the domain of interest [1]. For simple systems with simple geometries, analytical evaluation of these governing equations may be possible. However, real-world problems are usually more complicated and therefore require some sort of approximate, numerical solution method. The Finite Element Method (FEM) is perhaps the most common choice due to its excellent versatility and decades of development dating back to the 1940s [2]. The FEM involves solving differential equations by discretizing (or *meshing*) a complex geometric domain into simple infinitesimal volumes, or *finite elements*, converting the governing equation(s) into a weak form consisting of an integral over the finite elements, and producing a system of algebraic equations which can be solved to determine the solution field for the problem.

While FEM produces useful results for a wide variety of contexts, the treatment of a body as a continuous uninterrupted medium, which is essential to classical continuum mechanics formulations dealing with partial differential equations, limits the types of problems on which such analysis can be performed. Specifically, any problem domain containing discontinuities which impact the physical behavior of interest does not strictly fit within the definition of classical continuum mechanics. Such problems pose difficulty for analysis with methods like FEM, which rely on evaluating spatial gradients throughout the geometric domain [3]. Problems dealing with material discontinuities, sharp geometric discontinuities, and fracture mechanics are some common examples of problems which do not fit into the core FEM

framework for this reason.

In 2000, Silling developed an alternative formulation to the standard theory of continuum mechanics which specifically allows for discontinuities in the modelled domain. The differential equations comprising classical continuum mechanics formulations require evaluating the gradient of the solution field, which is undefined at discontinuities. Silling's new formulation, called *peridynamics*, avoids this issue by replacing differential equations altogether with integral equations evaluating force interactions, or *bonds*, between discrete material points [4]. The mathematical paradigm shift away from PDEs allows peridynamics to naturally represent physical features such as crack tips and material interfaces which pose difficulties for gradient-based approaches like the FEM [5–8]. The FEM on the other hand must be enriched, or *eXtended* (XFEM), *after* discretization to incorporate discontinuities. Such enrichment typically requires supplementing the governing equations with additional kinematic relationships based on analytical solutions to idealized fracture problems, which can exhibit strong mesh dependence and pose difficulty for modelling complex fracture patterns [9]. By accounting for discontinuities in its core formulation, peridynamics avoids these difficulties while also maintaining an intuitive and simple mechanical model.

It is worth noting that the applicability and utility of the peridynamic method extends well beyond its ability to represent discontinuities. Two additional features of the peridynamic formulation lend the method exceptional versatility: the meshfree discretization approach, and the semi-dynamic nature of the formulation (which will be discussed further in Section 1.2). The combination of these two features allow for extremely broad definitions of the kinematic relationships between material points, which can be easily updated as the material points' states' (deformation, temperature, etc) change. This feature allows peridynamics to be readily applied to a very broad class of physical problems, such as those dealing with crack propagation [10], plasticity [11], thermal couplings [12, 13], wave dispersion [14, 15], multiscale physics [16], and large deformations [17]. As such, peridynamics is a powerful tool that has attracted much research interest in the short time it has existed.

While peridynamics provides exceptional flexibility in the variety of problems it can be

used for, the method is not without its limitations. For example, peridynamics models are known to have issues with material interpenetration [18], can require ad hoc tuning of artificial simulation parameters [19], and have limitations in the classes of physical problems which they can represent [20]. As peridynamics is a relatively novel method, many of these issues are areas of active research, with solutions continually being developed.

For example, in the original peridynamic theory, what is now known as *bond-based* peridynamics was limited in its ability to represent arbitrary material properties. By defining the force of a bond between points to be a function only of those points, the Poisson's ratio of a material could not be specified; instead, all analysis was restricted to the Poisson's ratio of 1/4 for 3D or 1/3 for 2D as the mathematical result of the bond-based formulation [4]. This restriction of Poisson's ratio was resolved with the development of *state-based* peridynamics, in which the interaction force between points is a function of other nearby points as well, thereby allowing the simulation of systems with arbitrary material properties [21].

One of the foremost remaining limitations of peridynamics, particularly in comparison with FEM, is the lack of a clear boundary representation within the formulation. In peridynamics, a surface or void is only indicated by whether bonds do or do not pass through a given region. Furthermore, as is described further in Section 1.2, the peridynamic formulation generally assumes each material point in a domain has a complete, spherical *neighborhood* of points surrounding it [22]. These facts lead to two major difficulties in the enforcement of boundary conditions:

(a) There is no exact method by which displacements or tractions can be applied to a boundary. Instead, such conditions are applied volumetrically, as displacements or body forces acting on material points *near* the material surface. While in practical use cases such treatment is usually acceptable, or even more physically accurate than a true boundary condition, this fact makes comparison with alternative boundary value problem solutions only approximate.

(b) Near boundaries of the modelled domain where the assumption of a complete neigh-

borhood untrue, a well-documented numerical artifact exists, causing the material in those regions to have a lower effective stiffness than specified by the material properties [22–27]. In other words, the zero-traction condition of free surfaces that is automatically guaranteed in FEM is not inherently enforced in peridynamics. While a number of corrections to this issue have been proposed, only one truly eliminates the effect altogether, Le & Bobaru's "fictitious node method" [23]. However, this method was only formulated for application to extremely simple geometries.

In addition to lacking an explicit boundary representation, one of the most crucial drawbacks of peridynamics has been its computational expense. The numerous benefits of peridynamics do not come for free; compared to an equivalent finite element method (FEM) simulation, a serially-computed peridynamic simulation can require considerably more computation time. One contributor to this expense is the dense nodal connectivity found in peridynamics, with material points interacting with more neighbors than local methods like FEM. However, nodal density is a low-order impact on computational complexity; the main limitation on peridynamic simulation speed is the iterative solution approach required for the semi-dynamic modelling discussed above. In order to evaluate changing interactions between material points for phenomena like crack propagation and large deformations, bond forces must be iteratively evaluated through the entire domain for numerous iterations until equilibrium is achieved, which for serial computation is generally far slower than the direct solution methods used in FEM. This expense can be managed to an extent by employing pseudo-dynamic time integration approaches such as the dynamic relaxation method to minimize the number of iterations required [28]. However, evaluating in serial the forces acting on each node in a given timestep ensures a slower solution than any direct solve method. Fortunately, these nodal evaluations can be performed largely independently, and are therefore very well posed for parallel computation. Parallelization based on multi-core CPU setups, namely MPI (Message-Passing Interface) CPU parallelizations, have been well-established and show appreciable acceleration over serial implementations [29], and state-of-the-art peri-

dynamics software certainly incorporates multi-CPU parallelism. GPU-based parallelization was briefly explored a decade ago [30, 31], but was generally found to be less advantageous than MPI parallelization [32]. While this may have been a reasonable conclusion at that time, GPU technology has developed rapidly in recent years, and remains a promising avenue for further acceleration of the peridynamic method.

The overarching mission of this work is to advance the state-of-the-art of peridynamics, removing obstacles inhibiting its widespread adoption. Chapter 2 focuses on the issue of boundary treatment, generalizing Le & Bobaru's fictitious node approach such that it can be applied to arbitrary domains and creating a system for the precise application of true boundary conditions. Chapter 3 develops a GPU parallelization of peridynamics using the CUDA architecture, exploring in detail the various bottlenecks of a basic GPU implementation and performing numerous high- and low- level optimizations of both computational efficiency and memory consumption to obtain an extremely powerful computational implementation of the peridynamic method.

With the completion of Chapters 2 and 3, a highly efficient and accurate tool is developed, capable of simulating a wide variety of problems with greater flexibility than traditional methods like FEM. In particular, the speed-ups yielded by the GPU parallelization substantially expand the utility of peridynamics by drastically reducing the computational overhead of employing the method. Two use cases where these improvements make peridynamics particularly more attractive are topology optimization & nonlinear elasticity. Topology optimization typically involves iteratively simulating the behavior of the design domain then modifying the topology, repeating until an optimal topology is achieved. The finite element method is the most common method used for the simulation step, and a single FEM simulation can be quite expensive; for large domains, the need for numerous FEM simulations for an optimization problem can be prohibitively expensive. Some explorative work has shown that peridynamics may be used in place of the FEM, and is a good choice for the specialized problems for which it is well suited [17, 33, 34]. Chapter 4 explores the application of the GPU-parallelized peridynamic method to topology optimization problems & nonlinear

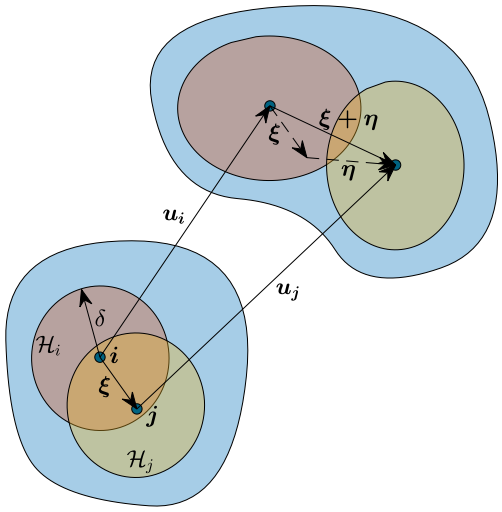

Figure 1.1: Diagram of undeformed and deformed bond configurations in a peridynamic medium.

analysis, and the computational benefits yielded. The GPU peridynamic framework is found to provide significant computational acceleration for use cases in both topology optimization and nonlinear analysis. Furthermore, specialized applications are explored in which FEM-based analysis is intractable, but is made possible with the use of the peridynamic solver.

1.2 Peridynamic Formulations

The following sections of this document all involve improvements on and implementations of the peridynamic simulation method. As such, a mathematical description of the method is included here for reference in the following sections.

Peridynamics discretizes a geometric domain into material points, or nodes. Each node $\mathbf{x}$ is assumed to only interact with other nearby nodes, typically those falling within a spherical neighborhood around x, $\mathcal{H}_x$. The radius of this neighborhood is termed the *horizon*, δ. The governing peridynamic equation, describing the acceleration $\ddot{\mathbf{u}}(\mathbf{x}, \mathbf{t})$ of each node $\mathbf{x}$, is given

as

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},\mathbf{t}) = \int_{\mathcal{H}_x} \mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t)dV_{\hat{\mathbf{x}}} + \mathbf{b}(\mathbf{x},t) \tag{1.1}$$

where $\rho(\mathbf{x})$ is the density of the material, $\mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t)$ is the force exerted on $\mathbf{x}$ by a neighboring point $\hat{\mathbf{x}}$, and $\mathbf{b}(\mathbf{x},t)$ is any body force acting on $\mathbf{x}$.

There are two main formulations for $\mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t)$, bond-based and state-based peridynamics, detailed formulations of which can be found in [4] and [21], respectively. Bond-based peridynamics follows a simpler formulation and is therefore more intuitive and less computationally complex. However, state-based peridynamics is more general than bond-based as it resolves the inability of bond-based peridynamics to represent materials with arbitrary Poisson's ratio. Overviews of each of these approaches are detailed below. While $\mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t)$ can be formulated to represent a wide variety of behaviors including both fluid and solid mechanics [35], and various constitutive relationships within each of those domains, the following overviews focus on linear elastic solids (unless otherwise noted) to maintain a reasonable scope for this work. For such problems, the field being solved for is the displacements of material points within the domain of interest, subject to applied forces and displacements. Although specific damage models are not the focus of this work, a simple brittle fracture model is incorporated for completeness.

1.2.1 Bond-based Peridynamics

In a linearly elastic bond-based peridynamic model, the magnitude f of the bond force vector $\mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t)$ is proportional to the elongation s of the bond:

$$f = cs \tag{1.2}$$

Elongation s is defined as

$$s = \frac{||\boldsymbol{\xi} + \boldsymbol{\eta}|| - ||\boldsymbol{\xi}||}{||\boldsymbol{\xi}||} \tag{1.3}$$

where $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ are the undeformed and deformed bond vectors, respectively (as depicted in Figure 1.1). The bond constant c is defined for a linear elastic material as

$$c = \begin{cases} \frac{18k}{\pi\delta^4} & \text{(3D Elasticity)} \\ \frac{12E}{\pi h\delta^3(1+\nu)} & \text{(2D Elasticity)} \end{cases} \tag{1.4}$$

where k is the classical bulk modulus, E is Young's Modulus, ν is Poisson's Ratio, and h is the thickness of the material in 2D problems. As mentioned in earlier sections, a bond-based model requires Poisson's Ratio to be set at 1/3 for 2D problems and 1/4 for 3D problems for these bond constants to correctly represent linear elasticity.

To automatically account for large deformations and rotations, the bond force acts in the direction of the deformed bond:

$$\mathbf{f}(\mathbf{x}, \hat{\mathbf{x}}, t) = f\frac{\boldsymbol{\xi} + \boldsymbol{\eta}}{||\boldsymbol{\xi} + \boldsymbol{\eta}||} \tag{1.5}$$

Finally, it is relevant to include the peridynamic definition of strain energy density w for a material point $\mathbf{x}$:

$$w(\mathbf{x}) = \frac{1}{4}\int_{\mathcal{H}_\mathbf{x}} cs^2||\boldsymbol{\xi}||dV_\mathbf{x} \tag{1.6}$$

1.2.2 State-based Peridyanmics

A peridynamic state describes some time-dependent attribute of the relation between two interacting nodes; i.e., the state of the bond between those nodes. State-based peridynamics evaluates the deformation of the body in terms of the elongation state $\underline{e} = ||\boldsymbol{\xi} + \boldsymbol{\eta}||$, where $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ are the undeformed and deformed bond vectors. This elongation state is used to determine $\mathbf{f}$ via elastic strain energy.

Let the operator $\bullet$ denote the operation $\underline{A} \bullet \underline{B} = \int_{\mathcal{H}_x} \underline{A} \cdot \underline{B} dV$ on states $\underline{A}$ and $\underline{B}$ of the bonds of node $\mathbf{x}$. The dilation θ of $\mathbf{x}$ is defined as

$$\theta = \begin{cases} \frac{3(\underline{\underline{\omega x}}) \bullet \underline{\underline{e}}}{m} & \text{(3D Elasticity)} \\ \frac{2(2\nu-1)}{\nu-1} \frac{(\underline{\underline{\omega x}}) \bullet \underline{\underline{e}}}{m} & \text{(Plane Stress)} \\ \frac{2(\underline{\underline{\omega x}}) \bullet \underline{\underline{e}}}{m} & \text{(Plane Strain)} \end{cases} \tag{1.7}$$

where $\underline{\omega}$ is a spherical influence function, $\underline{x}$ is the undeformed bond length $||\boldsymbol{\xi}||$, and m is the weighted volume $(\underline{\omega x}) \bullet \underline{x}$. The peridynamic strain energy density W is defined as

$$W = \frac{k' \theta^2}{2} + \frac{\alpha}{2} \underline{(\omega e^d)} \bullet \underline{e^d} \tag{1.8}$$

where $\underline{e^d}$ is the deviatoric elongation state $\underline{e^d} = \underline{e} - \frac{\theta \underline{x}}{3}$ and k' and α are defined as

$$k' = \begin{cases} k & \text{(3D Elasticity)} \\ k + \frac{\mu}{9} \frac{(\nu+1)^2}{(2\nu-1)^2} & \text{(Plane Stress)} \\ k + \frac{\mu}{9} & \text{(Plane Strain)} \end{cases} \tag{1.9}$$

$$\alpha = \begin{cases} \frac{15\mu}{m} & \text{(3D Elasticity)} \\ \frac{8\mu}{m} & \text{(2D Elasticity)} \end{cases} \tag{1.10}$$

with k and μ being the material's bulk and shear moduli. A peridynamic force state $\underline{t}$ can then be obtained by taking the second Fréchet derivative of W [36].

$$\underline{t} = \begin{cases} \frac{2k'\theta}{m}\underline{\omega x} + \alpha\underline{\omega e^d} & \text{(3D Elasticity)} \\ \frac{2(2\nu-1)}{\nu-1}\big(k'\theta - \frac{\alpha}{3}(\underline{\omega e^d} \bullet \underline{x})\big)\frac{\underline{\omega x}}{m} + \alpha\underline{\omega e^d} & \text{(Plane Stress)} \\ 2\big(k'\theta - \frac{\alpha}{3}(\underline{\omega e^d} \bullet \underline{x})\big)\frac{\underline{\omega x}}{m} + \alpha\underline{\omega e^d} & \text{(Plane Strain)} \end{cases} \tag{1.11}$$

where the force vector acting between two nodes $\mathbf{x}$ and $\hat{\mathbf{x}}$ is the sum of the force states of $\mathbf{x}$'s bond to $\hat{\mathbf{x}}$ and $\hat{\mathbf{x}}$'s bond to $\mathbf{x}$, acting in the direction $\frac{\boldsymbol{\xi}+\boldsymbol{\eta}}{||\boldsymbol{\xi}+\boldsymbol{\eta}||}$; i.e.

$$\mathbf{f}(\mathbf{x}, \hat{\mathbf{x}}, t) = \big(\underline{t}[\mathbf{x}, \hat{\mathbf{x}}] + \underline{t}[\hat{\mathbf{x}}, \mathbf{x}]\big)\frac{\boldsymbol{\xi} + \boldsymbol{\eta}}{||\boldsymbol{\xi} + \boldsymbol{\eta}||} \tag{1.12}$$

Thus, Eq. 1.1 can be evaluated in terms of the displacements of each node to determine the accelerations of each node, as a function of the current displacement field.

1.2.3 Solution Methods

The bond- and state-based formulations presented above provide a framework for evaluating the net acceleration of a material point for a given displacement field. To model the system's behavior, some solution method is required to determine the dynamic and/or static response of the system.

Direct Solution

For a subset of use cases of peridynamics, it may be possible to use a direct solution method similar to that used in FEM. Specifically, if the user is only interested in the equilibrium response of the system and material point interactions are not strongly impacted by a changing deformation field, a direct solve approach would be appropriate. These restrictions eliminate problems such as those dealing with large deformations, plasticity, and crack growth, but still include other common peridynamics problems like evaluating displacement fields around

crack tips. In such cases, it has been shown that the peridynamic integral equations can be linearized and assembled into a linear matrix system and solved with linear algebra solvers in the same way as FEM problems, setting the right-hand-side of the Eq. 1.12 to zero [27]. As such, the computation process and speed is comparable to that of FEM, although equivalent problems in peridynamics may produce a somewhat more dense stiffness matrix and therefore a slower solution speed. While these computational results are often favorable compared to the iterative solution methods discussed below, due to the limited range of problem types solvable with this method direct solution will not be focused on further in this work.

Physical Iteration

If a user wishes to model the full dynamic behavior of the system, a true physical iteration technique is required to capture the system's dynamic response. Instead of setting nodal accelerations to zero and solving for the displacement field, the problem is evaluated as an initial value problem, setting initial velocities and displacements of free nodes (those without applied displacements) to be zero, applying boundary conditions, and stepping forward in time.

Various integration schemes can be applied to this end, each with their various benefits. For example, Euler's method is computationally simple but can lead to error accumulation; higher order methods like RK4 are more stable but have larger computational and memory requirements. velocity-Verlet integration is a popular choice for peridynamics which does not require much memory and conserves the energy of the system. The remainder of this work will use the following velocity-Verlet integration process when physical iteration is required, but note that this can easily be swapped with other integration schemes.

$$\dot{\mathbf{u}}(t + \frac{\Delta_t}{2}) = \dot{\mathbf{u}}(t) + \frac{\Delta_t}{2}(\ddot{\mathbf{u}} - \zeta\dot{\mathbf{u}}(t)) \tag{1.13a}$$

$$\mathbf{u}(t + \Delta_t) = \mathbf{u}(t) + \Delta_t\dot{\mathbf{u}}(t + \frac{\Delta_t}{2}) \tag{1.13b}$$

$$\dot{\mathbf{u}}(t+\Delta_t) = \dot{\mathbf{u}}(t+\frac{\Delta_t}{2}) + \frac{\Delta_t}{2}(\ddot{\mathbf{u}} - \zeta\dot{\mathbf{u}}(t+\frac{\Delta_t}{2})) \tag{1.13c}$$

where Δ_t is the timestep size and ζ is an artificial damping term ensuring convergence.

Artificial Relaxation Iteration

While the physical iteration scheme maintains a direct representation of real physical dynamics, it can demand numerous iterations before reaching equilibrium, and can therefore be unfavorable. To establish some middle ground between the restrictive direct solve method and computationally expensive physical iteration, an adaptive dynamic relaxation (ADR) method was developed to allow for updating material point interactions while still arriving at equilibrium in a minimal number of iterations [28]. Unlike direct solution, this method can be applied to problems dealing with large deformation and plasticity. It is possible to model crack growth with ADR; however, for complex fracture patterns it is not recommended as the artificial damping can affect the final simulation results.

ADR works by introducing an artificial damping term to the governing equation (Eq. 1.12), which is updated every iteration to ensure the solution field rapidly approached equilibrium:

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) + c\dot{\mathbf{u}}(\mathbf{x},t) = \mathbf{F}(\mathbf{x},t) \tag{1.14}$$

where c is the artificial damping coefficient and $\mathbf{F}(\mathbf{x},t)$ is the result of the force integral comprising the right-hand side of Eq. 1.12. Material point velocity $\dot{\mathbf{u}}$ and acceleration $\ddot{\mathbf{u}}$ at iteration n can be approximated with the central difference theorem as

$$\dot{\mathbf{u}}^n = \frac{1}{2\Delta_t}\left(\mathbf{u}^{n+1} - \mathbf{u}^{n-1}\right) \tag{1.15}$$

$$\ddot{\mathbf{u}}^n = \frac{1}{\Delta_t^2}\left(\mathbf{u}^{n+1} - 2\mathbf{u}^n + \mathbf{u}^{n-1}\right) \tag{1.16}$$

Equations 1.14, 1.15 and 1.16 can be combined to produce

$$\mathbf{u}^{n+1} = \frac{1}{2 + c\Delta_t}\left(2\Delta_t^2\mathbf{F}^n + 4\mathbf{u}^n + (c\Delta_t - 2)\mathbf{u}^{n-1}\right) \tag{1.17}$$

For 2- and 3- dimensional problems, the Courant–Friedrichs–Lewy convergence condition can be used to determine a conservative critical timestep [37]:

$$\Delta_t = \frac{2}{n}\Delta_x\sqrt{\frac{\rho}{E}} \tag{1.18}$$

where n is the dimension of the problem and Δ_x is the grid spacing. Finally, the damping ratio at a given iteration c^n is determined using Rayleigh's quotient [28]:

$$c^n = 2\sqrt{\frac{(\mathbf{u}^n)^T\mathbf{k}^n\mathbf{u}^n}{\mathbf{u}^n)^T\mathbf{u}^n}} \tag{1.19}$$

where $\mathbf{k}^n$ is the diagonal local stiffness matrix

$$k_{ii}^n = -\frac{f_i^n - f_i^{n-1}}{\Lambda_{ii}(u_i^n - u_i^{n-1})} \tag{1.20}$$

Here f_i is the ith component of $\mathbf{F}$, and Λ_{ii} is typically set to 1. In the event of zero velocity ($u_i^n = u_i^{n-1}$), k_{ii}^n is set to 0 to avoid an undefined stiffness.

Chapter 2

PERIDYNAMIC SURFACE EFFECT

2.1 Background

As with any numerical method, assumptions made withing the peridynamic formulation limit the accuracy of the simulation. The PD surface effect is generated by the assumption of a "complete horizon" of all simulation nodes; formulations of peridynamics assume that the horizon of every node is completely filled with material, which of course is untrue for the layer of nodes at the boundaries of the geometric domain. Because of this incorrect assumption, surface nodes end up with an erroneously lower effective Young's Modulus than nodes lying in the bulk of the domain.

A number of methods have been formulated to correct for this effect. A single approach has yet to become globally accepted as the best method, but the most common contenders are as follows:

- **Volume Method [24]:** The volume method for surface effect correction applies a correction factor λ to the stiffness of a bond between surface points $\mathbf{x}$ and $\hat{\mathbf{x}}$ based on the volumes V of the neighborhoods of $\mathbf{x}$ and $\hat{\mathbf{x}}$, relative to the volume V_0 of a point in the bulk:

$$\lambda = \frac{2V_0}{V_{\mathbf{x}} + V_{\hat{\mathbf{x}}}} \tag{2.1}$$

- **Force Density Method [25]:** Like the volume method, the force density correction method also applies a correction factor to the stiffness of each bond, based on the force density rather than neighborhood volume. In each spatial dimension, a uniaxial strain is applied, and the resulting force density g is calculated; for example in the x dimension for a point $\mathbf{x}$:

$$g_x(\mathbf{x}) = \int_{\mathcal{H}_{\mathbf{x}}^+} f_x dV_{\hat{\mathbf{x}}} - \int_{\mathcal{H}_{\mathbf{x}}^-} f_x dV_{\hat{\mathbf{x}}} \tag{2.2}$$

  where $\mathcal{H}_{\mathbf{x}}^+$ and $\mathcal{H}_{\mathbf{x}}^+$ are the positive- and negative-facing halves of $\mathcal{H}_{\mathbf{x}}$ in the x-direction, and f_x is the x-component of the force vector defined in Section 1.2. Each directional

component of this force density is assembled into a vector $\mathbf{g}(\mathbf{x})$, and compared to the force density of a point in the bulk, $\mathbf{g}(\mathbf{x}_{bulk})$:

$$\mathbf{h}(\mathbf{x}) = \frac{\mathbf{g}(\mathbf{x}_{bulk})}{\mathbf{g}(\mathbf{x})} \tag{2.3}$$

The bond-wise equivalent of this vector $\mathbf{k}$ is determined:

$$\mathbf{k}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{\mathbf{h}(\mathbf{x}) + \mathbf{h}(\hat{\mathbf{x}})}{2} \tag{2.4}$$

Finally, this vector is reduced to a scalar λ, which is used as the factor of correction for the stiffness of each bond:

$$\lambda = \frac{1}{||\frac{\mathbf{n}}{\mathbf{k}}||} \tag{2.5}$$

where $\mathbf{n}$ is the unit normal vector of the bond between $\mathbf{x}$ and $\hat{\mathbf{x}}$. Once an initial value has been determined for λ of each bond, the process may be iterated, recalculating the new force density of the surface points and recomputing λ, until sufficiently converged.

- **Energy Density Method [26]:** The energy density correction method is identical to the force density method, simply replacing the force density with strain energy density. The strain energy density is defined in Section 1.2. The results for each directional applied strain are assembled into a vector $\mathbf{w}(\mathbf{x})$, which replaces $\mathbf{g}(\mathbf{x})$ in Eq. 2.3. The remaining steps of the force density method are followed identically.

- **Force Normalization Method [27]:** The force normalization method also applies a correction factor to the stiffness of bonds near the material surface, based on the difference in the required force needed to restore an applied virtual displacement, relative to points in the bulk. If a displacement vector $\mathbf{u}$ is applied to a point $\mathbf{x}$, the force needed to counteract this displacement is $-\mathbf{P}\mathbf{u}$, where $\mathbf{P}$ is the tensor

$$\mathbf{P}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{u}} \int_{\mathcal{H}_\mathbf{x}} \mathbf{f} dV_{\hat{\mathbf{x}}} \tag{2.6}$$

The stiffness correction factor λ is then given as

$$\lambda = \frac{p^1(\mathbf{x}_{bulk})}{p^1(\mathbf{x})} \tag{2.7}$$

where p^1 is the maximum eigenvalue of $\mathbf{P}$. Finally, as with the other methods, the correction factors of points $\mathbf{x}$ and $\hat{\mathbf{x}}$ are averaged to set the stiffness correction factor for the bond connecting those points.

- **Position-Aware Linear State-Based Method [22, 23]:** The position-aware linear state-based correction method (PALS) is, as the name implies, a correction method for state-based peridynamics based in the modification of nodal influence functions $\underline{\omega}$ for points near surfaces; in other words, making influence functions "position-aware". With this modification, $\underline{\omega}$ can no longer be any spherical function, but is constructed from a spherical function $\underline{\omega}^0$ as follows. A modified function $\underline{\omega}^*$ is defined as

$$\underline{\omega}^* = \frac{2(2\nu - 1)}{(\nu - 1)(\underline{\omega}^0 \underline{x}) \bullet \underline{x}} \underline{\omega}^0 \tag{2.8}$$

From there, the final corrected influence function $\underline{\omega}$ is obtained:

$$\underline{\omega} = \underline{\omega}^* + \sum_{n=1}^{K} \lambda^n \underline{xe}^k \tag{2.9}$$

where the constants λ^n are determined by solving the following system:

$$\sum_{n=1}^{K} \lambda^n (\underline{xe}^n \bullet (\underline{xe}^k) = Tr(\mathbf{H}^k) = (\underline{\omega}^* \underline{x}) \bullet \underline{e^k}. \tag{2.10}$$

Tensors $\mathbf{H}^k$ are applied virtual strain tensors, as defined in [22]. Additionally, a second influence function $\underline{\sigma}$ is introduced, replacing $\underline{\omega}$ in Eqs. 1.8 and 1.11. $\underline{\sigma}$ is constructed

in a similar way, defining $\underline{\sigma}^*$ as

$$\underline{\sigma}^* = \frac{4}{(\underline{\omega}^0 \underline{x}) \bullet \underline{x}} \underline{\omega}^0 \tag{2.11}$$

and constructing $\underline{\sigma}$ with

$$\underline{\sigma} = \underline{\sigma}^* + \sum_{k=1}^{K} \tau^k \underline{e}^{dk} \underline{e}^{dk} \tag{2.12}$$

where τ^k is determined by solving the system

$$\sum_{n=1}^{K} \tau^n (\underline{e}^{dn} \underline{e}^{dn}) \bullet (\underline{e}^{dk} \underline{e}^{dk}) = \sum_{i,j=1}^{2} (\epsilon_{ij}^{dk})^2 + \frac{1}{2} (\epsilon_{33}^{dk})^2 \tag{2.13}$$

where ϵ_{ij}^{dk} is the ij element of the deviatoric component of the strain tensors $\mathbf{H}^k$

Despite these various attempts at resolving the surface effect issue, none achieve a completely satisfactory result, for a few reasons. Firstly, it should be noted that the first 4 methods are all only formulated for bond-based peridynamics, which as Section 1.2 discusses is not a fully general method. PALS, the only method here formulated for state-based peridynamics, is complicated and computationally expensive. Most importantly though is the fact that none of these methods truly eliminate the surface effect altogether, instead only reducing it through approximate adjustments to the surface points [23].

In 2018, Le & Bobaru evaluated each of these approaches, and introduced a novel 5th method, the fictitious node method. Unlike other available methods, this fictitious node approach is capable of suppression of the surface effect to within machine precision. However, the formulation of this approach developed by Le & Bobaru can only be applied to an extremely restricted set of geometries, namely those consisting only of surfaces aligned with a rectangular nodal grid.

Herein lies the motivation for the first step of this research project: develop an abstraction of Le & Bobaru's approach to the solution of the surface effect, such that total or near-total suppression of the surface effect can be achieved with arbitrary geometries. Achieving this

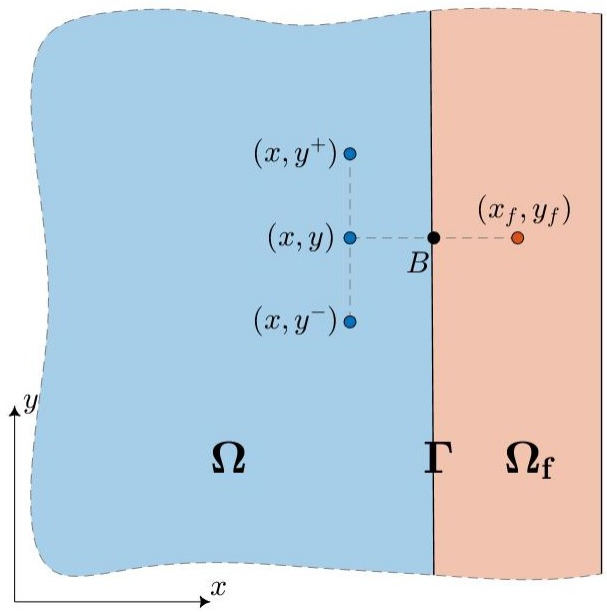

Figure 2.1: Material domain Ω with boundary line parallel to the y axis. The fictitious domain Ω_f covers a layer to the right of the boundary line with sufficient thickness to complete the horizons of all nodes in Ω.

goal will not only eliminate the major remaining obstacle in using peridynamics for the foundation of a practical, novel topology optimizer, but also be a significant advancement for the field of peridynamics in general.

2.2 The Fictitious Node Method

The general method of a fictitious node boundary treatment consists of padding the body of interest (Ω) with a layer of 'fictitious nodes' (Ω_f) such that every material node within the body has a complete horizon; i.e., $(H_x \cap \Omega) \cup (H_x \cap \Omega_f) = H_x$ for all x in Ω. Unlike the material nodes, whose displacements are set by the peridynamic governing equation (Eq. 1.1), the fictitious nodes provide additional degrees of freedom that can be chosen to enforce the boundary conditions (including zero-traction boundary conditions) being applied at the surface on which they are located.

Fig. 2.1 shows a section of an axis-aligned geometry boundary Γ, with the true domain Ω shown in blue and the fictitious padding Ω_f shown in red. A fictitious node is shown in red at (x_f, y_f), and material nodes are shown in blue at (x, y^-), (x, y), and (x, y^+). To determine the appropriate locations of this fictitious node, Le & Bobaru reference the classical Hooke's Law relationships, which for plane stress are

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{xy} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & 1-\nu \end{bmatrix} \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_{xy} \end{bmatrix} \tag{2.14}$$

where $\sigma_{\mathbf{x}}$, $\sigma_{\mathbf{y}}$, and $\sigma_{\mathbf{xy}}$ are the nonzero components of the Cauchy stress tensor, $\epsilon_{\mathbf{x}}$, $\epsilon_{\mathbf{y}}$, and $\epsilon_{\mathbf{xy}}$ are the corresponding strains, and material properties E and ν are Young's Modulus and Poisson's Ratio respectively. Assuming small deformation, the strains ϵ can be defined in terms of the gradient of the displacement field u as

$$\epsilon_{xx} = \frac{\partial u_x}{\partial x}, \epsilon_{yy} = \frac{\partial u_y}{\partial y}, \epsilon_{xy} = \frac{1}{2}\Big(\frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y}\Big) \tag{2.15}$$

On Γ, some boundary condition must be specified. If a traction or free surface boundary condition is specified, σ_{xx} and σ_{xy} are known. Additionally, the surface strain at point B can be approximated through a differencing scheme based on displacements of the material nodes indicated in Fig. 2.1:

$$\epsilon_{xx} = \frac{u_x(x_f, y_f) - u_x(x, y)}{x_f - x} \tag{2.16a}$$

$$\epsilon_{yy} = \frac{u_y(x, y^+) - u_y(x, y^-)}{y^+ - y^-} \tag{2.16b}$$

$$\epsilon_{xy} = \frac{1}{2}\left(\frac{u_y(x_f, y_f) - u_y(x, y)}{x_f - x} + \frac{u_x(x, y^+) - u_x(x, y^-)}{y^+ - y^-} \right) \tag{2.16c}$$

Combining Eqns. 2.14 and 2.16 and solving for $u_x(x_f, y_f)$ and $u_y(x_f, y_f)$ yields

$$u_x(x_f, y_f) = u_x(x, y) + (x_f - x)\left(\frac{\sigma_{xx}(1 - \nu^2)}{E} - \nu \frac{u_y(x, y^+) - u_y(x, y^-)}{y^+ - y^-} \right) \tag{2.17a}$$

$$u_y(x_f, y_f) = u_y(x, y) + (x_f - x)\left(\frac{2\sigma_{xy}(1 + \nu)}{E} - \frac{u_x(x, y^+) - u_x(x, y^-)}{y^+ - y^-} \right) \tag{2.17b}$$

Setting the displacements of the fictitious node layer according to Eq. 2.17 sets the appropriate traction boundary conditions at a set of points B along the surface of the body, thereby allowing the PD system's equilibrium state to correspond to the classical solution. However, since it assumes $y_f = y$ and Γ is parallel to the line of material nodes, Eq. 2.17 is ill-posed for all but the simplest geometries. Le & Bobaru show how a forward differencing scheme may be applied in place of the central differencing scheme shown in Eq. 2.16 in cases where nodes (x, y^-) or (x, y^+) are not present, for example at exterior corners on a rectangular domain. However, for cases where boundaries are not aligned with the axes of the nodal grid, the fictitious node formulation proposed by Le & Bobaru cannot be used to define fictitious node locations that satisfy given boundary conditions.

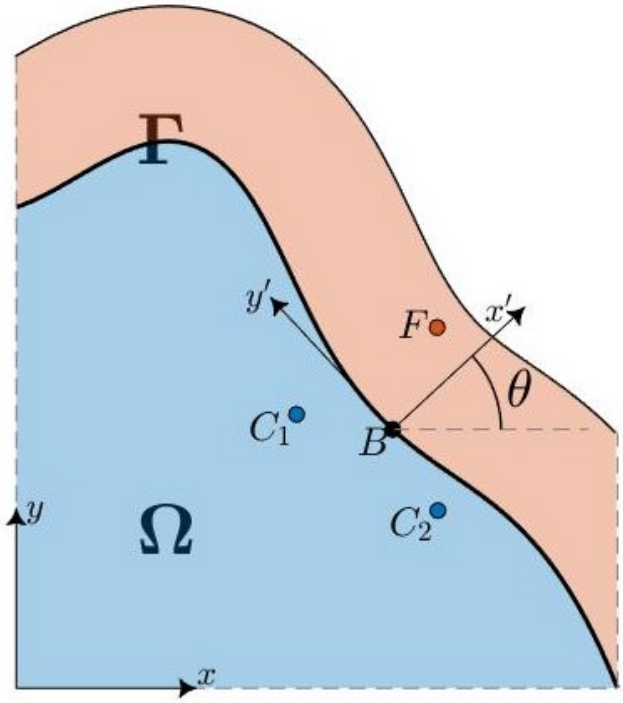

Figure 2.2: Control points C, fictitious node F, and boundary point B around an arbitrary boundary geometry. The material and fictitious domains are denoted with red and blue respectively, as in Fig. 2.1.

2.3 The Generalized Fictitious Node Method

Le & Bobaru's fictitious node approach lays promising groundwork for a more geometrically robust method. At its core, the method is simply an extrapolation of the displacement field within the bulk of the material to the fictitious padding which enforces a traction boundary condition at the material surface. The method Le & Bobaru present is a simplified version of this extrapolation which assumes an axis-aligned nodal grid; here the fictitious node approach is extended to accept arbitrary node locations and thereby extend to arbitrary geometries.

Consider a point B on a surface with arbitrary orientation & topology, as shown in Fig. 2.2 that is subject to the traction boundary conditions $\sigma_{x'x'}$ and $\sigma_{x'y'}$, where the (x', y') reference frame is defined such that the x' direction coincides with the surface normal at B.

As in the finite element and finite point methods [3, 38], a polynomial basis may be used to locally approximate the displacement field. For example, the displacement field in the vicinity of B can be approximated with a low-order basis as

$$u_{x'}(x', y') = a_0 + a_1 x' + a_2 y' \tag{2.18a}$$

$$u_{y'}(x', y') = a_3 + a_4 x' + a_5 y' \tag{2.18b}$$

for some set of coefficients a_n. Combining Eqs. 2.14, 2.15, and 2.18 (imposing the applied surface traction as the stresses in Eq. 2.14) yields

$$\begin{bmatrix} 0 & \frac{E}{1-\nu} & 0 & 0 & 0 & \frac{\nu E}{1-\nu^2} \\ 0 & 0 & \frac{E}{2(1-\nu^2)} & 0 & \frac{E}{2(1-\nu^2)} & 0 \end{bmatrix} \mathbf{a} = \begin{bmatrix} \sigma_{x'x'} \\ \sigma_{x'y'} \end{bmatrix} \tag{2.19}$$

where

$$\mathbf{a^T} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{bmatrix}$$

To fully constrain this system, the locations $\mathbf{x_i}'$ and displacements $\mathbf{u_i}'$ of the material nodes acting as control points C_i may be introduced to Eqs. 2.18 and 2.19:

$$\mathbf{Va} = \mathbf{f} \tag{2.20}$$

where

$$\mathbf{V} = \begin{bmatrix} 0 & \frac{E}{1-\nu} & 0 & 0 & 0 & \frac{\nu E}{1-\nu^2} \\ 0 & 0 & \frac{E}{2(1-\nu^2)} & 0 & \frac{E}{2(1-\nu^2)} & 0 \\ 1 & x'_1 & y'_1 & 0 & 0 & 0 \\ 1 & x'_2 & y'_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x'_1 & y'_1 \\ 0 & 0 & 0 & 1 & x'_2 & y'_2 \end{bmatrix}, \mathbf{f} = \begin{bmatrix} \sigma_{x'x'} \\ \sigma_{x',y'} \\ u_{x'_1} \\ u_{x'_2} \\ u_{y'_1} \\ u_{y'_1} \end{bmatrix}$$

Provided that $\mathbf{V}$ is invertible and points C_i and F are sufficiently close to B such that a linear displacement field approximation is reasonable, Eq. 2.20 may be used to solve for $\mathbf{a}$. After computing $\mathbf{a}$, the displacement of F can be set via Eq. 2.18 such that the boundary condition at B is enforced. For free surfaces, the stresses in $\mathbf{f}$ are simply set to 0.

Conveniently, all terms of $\mathbf{V}$ are dependent only upon nodal coordinates and material properties, which are typically known at the beginning of a peridynamic simulation. As such, the $\mathbf{V}$ matrix for each fictitious node can be assembled and inverted as a precomputation, leaving only the need to update $\mathbf{f}$ and recalculate $\mathbf{a}$ at each peridynamic iteration.

This method also provides a convenient mechanism by which displacement boundary conditions may be set in a peridynamic simulation. To apply a displacement $(u_{x'_B}, u_{y'_B})$ at B, the first two rows of $\mathbf{V}$ and $\mathbf{f}$ may be replaced with

$$\mathbf{V}_{(1:2,:)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \mathbf{f}_{(1:2)} = \begin{bmatrix} u_{x'_B} \\ u_{y'_B} \end{bmatrix}$$

Preexisting methods for applying displacement boundary conditions in peridynamic simulations typically set the displacements of a layer of fictitious nodes around B with the displacement $(u_{x'_B}, u_{y'_B})$, which *approximately* sets the displacement at B to be the same. This method on the other hand strictly enforces $(u_{x'_B}, u_{y'_B})$ at B.

The method presented above resolves the requirement of Le & Bobaru's method that geometry boundaries be aligned with the nodal grid. In doing so, however, it introduces the need for a method of selecting nearby boundary and control points for each fictitious node.

For the results presented below, the following heuristic was used:

Algorithm 1 Boundary and control point selection

```
function ASSEMBLEV(B,C1,C2)
    Evaluate boundary condition at boundary point B
    return Matrix V for B and control points C1,C2
end function

for F in {fictitious nodes} do
    B ← point on geometry's surface nearest to F
    T ← material node in material nearest to B
    (maxdet,C1,C2) ← (0, None, None)
    for M in {T, material nodes in horizon of T} do
        for N in {T, material nodes in horizon of T} do
            if M ≠ N then
                V ← ASSEMBLEV(B,M,N)
                vdet ← DETERMINANT(V)
                if |vdet| ≥ maxdet then
                    (maxdet,C1,C2) ← (|vdet|,M,N)
                end if
            end if
        end for
    end for
    (B_F,C1_F,C2_F) ← (B,C1,C2)
end for
```

Algorithm 1 essentially uses the horizon sets of each node to identify sets material points within the proximity of fictitious nodes, then uses an exhaustive combinatorial test to identify the subset of these material nodes that creates the best-conditioned $\mathbf{V}$ matrix. This process ensures that points C_i and F are reasonably close to B relative to the horizon size of the simulation, and that $\mathbf{V}$ is well-conditioned for inversion.

2.4 Discussion

Le & Bobaru use two test problems to measure the effectiveness of various surface correction methods: a rectangular bar under uniaxial tension, and a square cracked plate under mode I loading. Both of these problems have the advantage of meeting the geometric simplicity

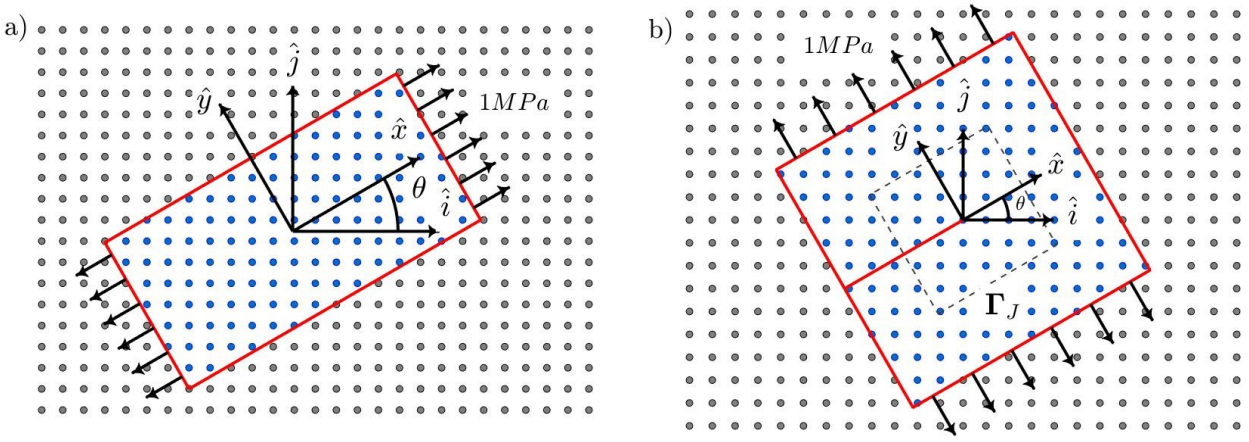

Figure 2.3: a) Test geometry for uniaxial tension test shown in red on the nodal grid. Grid and geometry reference frames ($[\hat{i}, \hat{j}]$ and $[\hat{x}, \hat{y}]$ respectively) are indicated. θ is the angle of rotation between the two reference frames. The applied traction in the $\hat{x}$ direction is shown. b) Cracked-plate geometry for the J-integral test is shown in red on the nodal grid. The dashed line denotes the contour for the J-integral evaluation, Γ_J. The Mode I loading in the $\hat{y}$ direction is shown. The nodal and geometry reference frames are indicated as in a).

requirements of Le & Bobaru's fictitious node method. The same test problems will be evaluated here so the results presented by Le & Bobaru may be used for comparison. However, to test the generalized fictitious node method's ability to handle arbitrary geometries, these tests will be repeated for $\theta = [0, 2\pi]$, where θ indicates the angle between the axes of geometric boundaries and that of the nodal grid as depicted in Fig. 2.3.

For the uniaxial tension test, a rectangular geometry (50mm x 100mm) is subjected to a uniaxial load $\sigma_{xx} = 1$ MPa. A 2D plane stress peridynamic model with a grid spacing of 1mm and a horizon size of 5mm is used to evaluate the resulting displacement field. A Young's Modulus E of 2.0 GPa and Poisson's ratio ν of 1/3 are used. The analytical solution for this problem is the homogeneous 2D strain field $\{\epsilon_{xx}, \epsilon_{yy}, \epsilon_{xy}\} = \{\frac{\sigma_{xx}}{E}, -\nu\frac{\sigma_{xx}}{E}, 0\}$. Fig 2.4 shows the resulting maximum absolute error in the simulated displacement fields for the θ parameter sweep described above. Nodes on the central vertical line have zero horizontal displacement and therefore excluded from the horizontal displacement error analysis; the

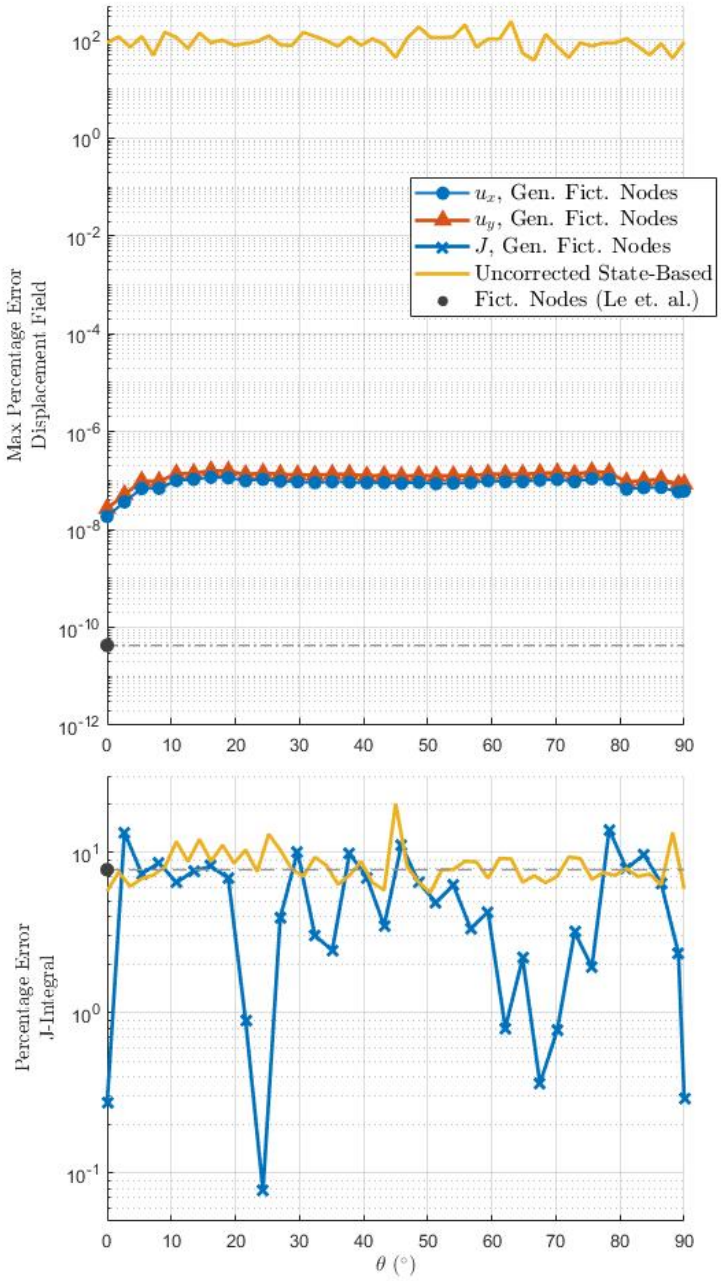

Figure 2.4: Accuracy of simulation results as a function of grid orientation. Top: Horizontal and vertical displacements of the uniaxial tension test, u_x & u_y, are compared to analytical results. Bottom: J-integral results are compared to high-resolution FEM results [39]. Results for the same tests without any surface treatment are included for reference. Le & Bobaru's results for the axis-aligned case are included as a solid dot on the vertical axis.

same is true of the vertical displacements of nodes on the central horizontal line.

To further verify the methodology presented, a problem with a non-homogeneous displacement field should be evaluated; the J-integral test described by Le & Bobaru suits this purpose. A 100mmx100mm square domain with a 50 mm edge crack is subjected to a 1 MPa Mode I loading, as shown in Fig. 2.3. The elastic properties E = 72 GPa and $\nu = 1/3$ and a grid spacing of 1mm are used. The PD expression used by Le & Bobaru to approximate the J-integral is taken from [39]:

$$\begin{aligned} J = & \sum_{i=1}^{n_{boundary}} W n_1 \Delta x - \\ & \frac{1}{2} \sum_{i=1}^{n_{inner}} \sum_{j=1}^{n_{outer}} \left(\mathbf{f}\big(\mathbf{x}_i, \mathbf{x}_j\big) * \Big(\frac{\partial \mathbf{u}_i}{\partial x_1} + \frac{\partial \mathbf{u}_j}{\partial x_1} \Big) \right) A_i A_j \end{aligned} \tag{2.21}$$

where W is the PD strain energy density, n_1 is the first component of the unit normal to the J-integral line's normal vector. $n_{boundary}$ is the number of nodes lying on the contour; n_{inner} and n_{outer} are the number of nodes inside and outside the contour, respectively. However, this approach is only well-posed for nodal grids that have nodes lying either entirely inside, entirely outside, or exactly on the J-integral contour. Clearly, this will only be true of $\theta = 0$ (as per Fig. 2.3) for the proposed test. Therefore, an alternative approach for the J-integral calculation is used [40] which determines the values of traction and strain energy from the displacement field directly rather than PD states, and thus can be readily interpolated to be sampled along the J-integral contour:

$$J = \Delta x \sum_{i}^{n_{sample}} W_i - \sigma_{\mathbf{i}} \mathbf{n_i} \frac{\partial \mathbf{u}_i}{\partial x} \tag{2.22}$$

where n_{sample} is the number of sample points taken along the J-integral contour, Δx is the spacing between those points, and $\mathbf{n}$ is the outward pointing normal vector. σ can be determined from the displacement field by combining Eqs. 2.14 & 2.15. W can also be

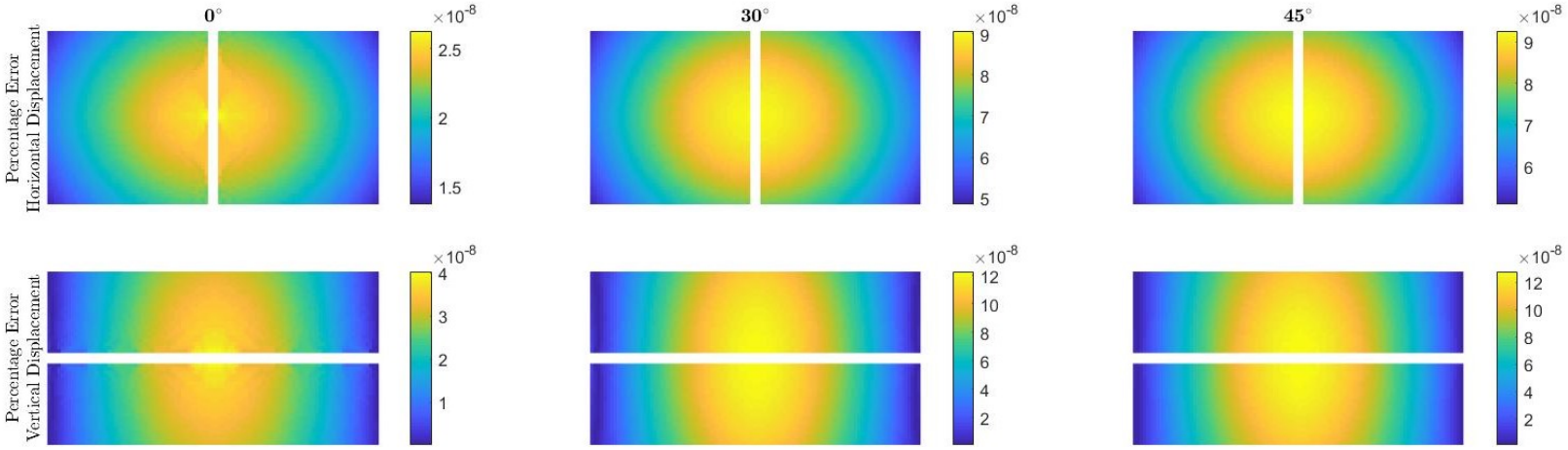

Figure 2.5: Error distribution of uniaxial tension test for selected grid orientations, relative to analytical results.

defined in terms of the partial derivatives of the displacement field:

$$W = \frac{E}{2(1-\nu^2)}\left[\left(\frac{\partial u_1}{\partial x_1}\right)^2 + \left(\frac{\partial u_2}{\partial x_2}\right)^2 + 2\nu\frac{\partial u_1}{\partial x_1}\frac{\partial u_2}{\partial x_2} + \frac{1-\nu}{2}\left(\frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1}\right)^2\right] \tag{2.23}$$

The gradient of the displacement field, needed for W, σ, and again in Eq. 2.22, is estimated at each sample point using a midpoint differencing scheme. Fig. 2.4 shows the results of this computation for $\theta = [0, 2\pi]$. The error show is relative to $J = 19.76N/m^2$, a value obtained from a high-resolution FEM study [39]. Le & Bobaru's result for the same grid refinement at $\theta = 0$ is plotted for comparison.

Fig. 2.4 displays the maximum percentage error in the horizontal and vertical displacement fields of the uniaxial tension test as a function of θ. The results show little dependence on θ; all orientations are able to achieve an accuracy of at least $1 \times 10^{-7}\%$, with the axis-aligned case ($\theta = 0$) showing a slightly better accuracy of $1 \times 10^{-8}\%$. The results for the

same test with no surface treatment are included for reference, as well as the performance of Le & Bobaru's fictitious node method for the axis-aligned case. While Le & Bobaru reported somewhat more accurate results for $\theta = 0$ ($3 \times 10^{-11}\%$, for both horizontal and vertical displacements), the generalized method is far more accurate than the uncorrected simulation, as well as the next-best correction method (3.5% and 6% for horizontal and vertical displacements, respectively [23]). The rotational invariance of the generalized method is further illustrated in Fig. 2.5, which shows the full error distribution for a few selection grid orientations: qualitatively, the distributions are nearly indistinguishable for the different orientations. Furthermore, for all orientations the location of the maximum error occurs at the point nearest the origin, at the center of the bar rather than near its surface, which indicates that the maximum error is now being driven primarily by the division by near-zero displacements near the origin rather than the surface effect.

Fig. 2.4 also displays the results of the J-integral test. The accuracy of this test lies around $\sim 10\%$ for most orientations, but higher accuracy is achieved (nearing $\sim 0.1\%$) for the axis-aligned case and other sections of the θ range. When evaluating these results, it should be noted that the J-integral calculation is highly sensitive to small changes in the displacement field. Additionally, the results are compared to high-resolution FEM results rather than exact analytical results. As such, a larger margin of error than the uniaxial test is to be expected. The orientations yielding $\sim 10\%$ error are on par with the results of Le & Bobaru's method for the axis-aligned case and the uncorrected method, providing a reasonable estimate of the J-integral value. For the axis-aligned case and the other better-performing orientations, the generalized fictitious node method outperforms not only Le & Bobaru's method and uncorrected state-based peridynamics, but also any other method tested in [23].

In conclusion, a generalized fictitious node method has been presented which alleviates the geometric constraints of the precursor presented by Le & Bobaru. The generalized method is able to achieve near-total elimination of the surface effect for a homogeneous displacement field in an arbitrarily-oriented domain, and maintain reasonable performance when evaluating

displacement fields with sharp discontinuities. More generally, this method enables the incorporation of boundary representations into the peridynamic framework, allowing for not only the enforcement of free-surface boundary conditions, but for the precise enforcement of boundary conditions in general. As these features were previously unavailable to peridynamic simulation, this development is a significant novel contribution to the field of peridynamics, allowing direct correspondence to the true boundary value problem methods found in classical continuum mechanics. However, it is important to note that this method bears a large computational expense, particularly in terms of the memory requirements demanded by the storage of interpolation matrices for the fictitious nodes. For this reason, the following chapter, which presents a GPU parallelization of peridynamics that minimizes memory usage, does not incorporate this surface correction method. Revising the computational approach for this method for spatial efficiency remains an open problem for further research.

Chapter 3

GPU PARALLELIZATION OF PERIDYNAMICS

3.1 Background

While peridynamics does have remaining unsolved limitations, as is seen in the improvements of boundary treatments in Chapter 2, the primary lingering issues have viable solutions in development [41]. Nonetheless, widespread use of peridynamics is often inhibited by the computational expense of available software; this chapter aims to address that concern through the development of a high-performance peridynamic simulation tool.

A peridynamic simulation is solved in one of two ways: directly, by assembling the nodal interactions into a stiffness matrix and inverting, much like the finite element method, or iteratively, evaluating the forces on each node, updating their displacements, and repeating until equilibrium is achieved [29]. For the former, due to the nonlocality inherent to the method and the numerous bonds each node has, the stiffness matrix will necessarily be denser than an equivalent FEM problem, and therefore more computationally expensive to solve [42]. Moreover, solving a peridynamic problem in such a way would eliminate much of the method's intended functionality, as physical phenomena like plasticity, large deformations, and crack growth inherently need iteration for proper solution. Therefore, the latter approach of iterative displacement updating is far more practical, but unfortunately the repeated nodal evaluations become quite costly, even with accelerated iteration schemes like dynamic relaxation [28].

Fortunately, the iterative peridynamic solution method is also highly parallelizable, as the nodal evaluations and displacement updates can be performed independently at each timestep [30, 42]. The vast majority of available peridynamics software uses some form of parallelism for acceleration [43, 44]. However, critical as the computational expense is for peridynamics, few comprehensive works are available detailing the current state of the art for the computational aspects of peridynamics. Sandia National Labs, a prominent institution in modern peridynamics research, published in 2015 one of the most detailed descriptions of the computational aspects of peridynamics, breaking down the method into its various elements (establishing nodal neighborhoods, time integration, contact modelling, etc.) and

describing the computational challenges of each component [42]. This work stops short of detailing the parallelization-related aspects of computational peridynamics. Fortunately, a handful of other works go into further detail of parallelizing peridynamics across multiple CPU processors, describing the details of breaking a simulation domain into blocks to send to each processor, how information is partitioned & shared between blocks, and so on [29, 32]. As mentioned above, all peridynamics software with parallelism capabilities publicly available at the writing of this document use this form of parallelism. However, another form of parallelism has the potential to provide far greater acceleration, and yet is far less utilized in the field of peridynamics: GPU-based parallelism.

GPU parallelism is fundamentally different from CPU parallelism, in that a GPU processor, while expensive, can launch several thousands more concurrent computations than a similarly priced multi-CPU setup. In the context of peridynamics, the computations for each node at a given timestep can potentially be performed totally in parallel, which is a significant departure from CPU-based parallelism. While this form of parallelism for peridynamics is underdeveloped, a general description of utilizing CUDA (the API used to execute code on Nvidia GPUs) to run peridynamics simulations on the GPU does exist [30]. However, despite more than halving the computation time of the equivalent CPU implementation, these early works in GPU-based peridynamics conclude that the memory limitations of the GPU processors available at the time limit the utility of a GPU-based approach to peridynamics [32].

However, further exploration is certainly required, as modern GPU resources not only allow larger problems to be simulated with increased GPU memory, but also provide far greater accelerations, upwards of 200 times relative to CPU implementations [45]. Some confirmations of these results on modern computing resources have been confirmed [31], but few go into the detail necessary to replicate the work. Prior to this work, only a single paper could be found with a full implementational description, including details on some optimizations & improvements made specifically for the GPU [46]. More recently, the GPU-based PeriPy software was published, demonstrating excellent acceleration of bond-based

peridynamics. However, further development of GPU peridynamics software is warranted, as several critical areas of investigation require further exploration: development of a GPU solver for state-based peridynamics, examining design choices such as the balancing of speed and memory consumption, and further optimization choices. Therefore, a considerable gap exists in the available work on GPU-parallelized peridynamics; as such, this chapter will focus on expanding the body of work in this field of study.

3.2 GPU Structure & Relevant Terminology

Before discussing the implementation of PD on the GPU, it will be useful to briefly describe the computational structure of the GPU and to identify some relevant terminology. GPUs can have a wide variety of different architectures; this work will focus on implementation for modern Nvidia GPUs, which are the most commonly used for high-performance computation. Additionally, the terminology used here will correspond to that of the CUDA API, used by Python packages such as Numba and PyCuda to access Nvidia GPUs. This section contains a brief overview of the GPU architecture as it pertains to this work; please see the CUDA documentation for more complete descriptions of the various micro-architectures found on different GPUs [47]. See Figure 3.1 for a depiction of the concepts discussed below.

3.2.1 Compute Hierarchy

- **Thread**: A thread is the basic compute unit in GPU software, which independently executes a piece of code in parallel with other threads. During execution, each thread has its own private memory allocation. Modern GPUs can have several thousand threads simultaneously active.

- **Block**: A block is a software-specified grouping of threads that are executed simultaneously. Each block has access to a shared memory allocation, and threads within a block can be synchronized during execution.

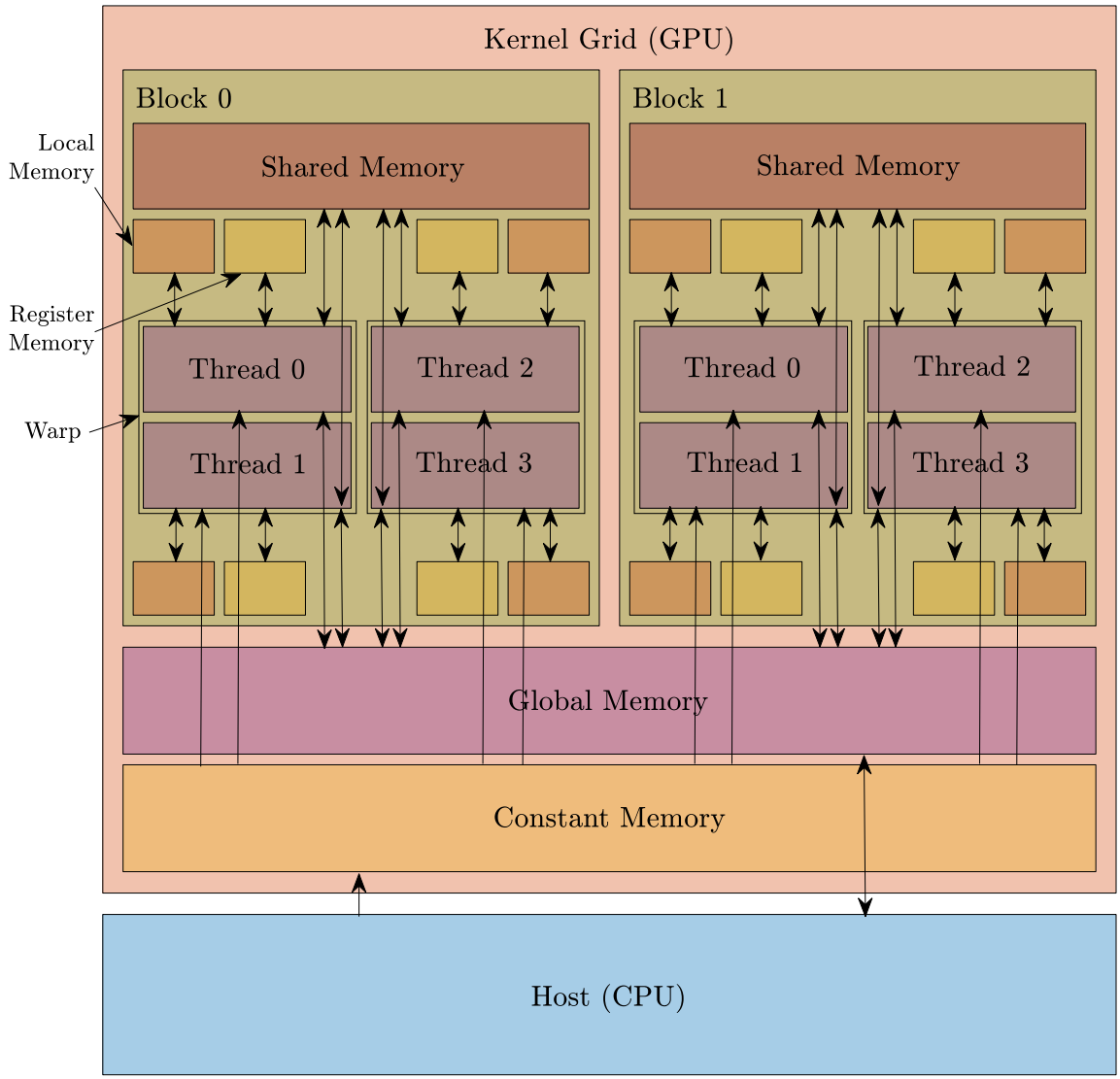

Figure 3.1: Diagram of GPU computational & memory hierarchies. Arrows indicate memory access privileges available to each compute level. Quantities of blocks per grid, threads per warp, and threads per block have been reduced for simplicity.

- **Warp**: A warp is a hardware grouping of threads that are executed simultaneously. For optimal performance (specifically, to maximize GPU *occupancy*), block dimension must be a multiple of warp size for an efficient mapping between the two groupings.

- **Kernel Grid**: The programmed instruction set (or, *kernel*) that a thread executes is launched on an array of blocks (1-, 2-, or 3-Dimensional) called the kernel grid. If a grid is launched containing more threads than a GPU can concurrently run (a hardware-specific limit), the grid forms a queue of blocks awaiting execution. Because of this, synchronization across all threads in a grid can only occur after the execution of the entire kernel.

3.2.2 Memory Hierarchy

- **Global Memory**: Global memory is the largest memory allocation on the GPU, typically several gigabytes. It is a general-purpose allocation that all threads can read from & write to. Global memory size is almost always the limiting factor in determining the problem size a given GPU can represent; the memory-optimized approach this work presents focuses on minimizing the data that must be stored in global memory. Additionally, global memory accesses by a thread are typically the slowest, so reducing the number of accesses a thread must make is critical to optimizing performance.

- **Constant Memory**: Like global memory, constant memory is accessible by all threads. As the name implies, it is read-only; data stored in constant memory must be known a kernel compilation time. As such, it is only useful in specific situations. Accessing constant memory is typically faster than global memory.

- **Shared Memory**: Shared memory is a memory allocation shared by all threads within a block. Accessing shared memory is substantially faster than global or constant memory. Shared memory persists for the duration of a block's execution.

- **Register Memory**: Register memory is allocated by and accessible to each individual thread. It is by far the fastest memory type to read & write into, and is generally used to store temporary variables during calculations within a kernel.

- **Local Memory**: Like register memory, local memory is accessible to individual threads. However, it is far slower, with similar bandwidth to global memory. As such, it is typically only used when register memory is fully utilized.

3.3 Computational Implementation

3.3.1 Basic Computational Representation & Naive Implementation

Algorithm 2 Naive GPU Implementation of Peridynamics

$\xi_{ijk} = E_{C_{ij}k} - E_{ik}$
$x_{ij} = \sqrt{\sum_k \xi_{ijk}^2}$
$m_i = \sum_j x_{ij} V_j$

for tt = 1:NT **do**
    $\eta_{ijk} = u_{C_{ij}k} - u_{ik}$
    $e_{ij} = \sqrt{\sum_k (\xi_{ijk} + \eta_{ijk})^2} - x_{ij}$
    $\phi_{ij} = \frac{x_{ij}+e_{ij}}{x_{ij}} < s_c \,\&\, \phi_{ij}$
    $\theta_i = \frac{3}{m_i} \sum_j e_{ij} \phi_{ij} V_j$
    $e_{ij}^d = e_{ij} - \theta_i x_{ij}/3$
    $t_{ij} = \frac{2k\theta_i}{m_i} + \frac{15\mu e_{ij}^d}{m_i x_{ij}}$
    $f_{ik} = b_{ik} + \sum_j (t_{ij} + t_{C_{ij}I_{ij}}) \frac{\xi_{ijk}+\eta_{ijk}}{e_{ij}+x_{ij}} \phi_{ij} V_j$
    $\dot{u}_{ik} += \frac{\Delta_t}{2}(f_{ik}^{t-1}/\rho - \zeta \dot{u}_{ik})$
    if S_{ik} **then**
        $u_{ik} = D_{ik}$
    else
        $u_{ik} += \Delta_t \dot{u}_{ik}$
    end if
    $\dot{u}_{ik} += \frac{\Delta_t}{2}(f_{ik}/\rho - \zeta \dot{u}_{ik})$
end for

A precursor to this work [45] introduces the approach towards parallelizing PD detailed in Algorithm 2. This method discretizes each mathematical step described in Section

1.2 into a tensor operation. Indices i, j, and k correspond to node index, bond index, and spatial dimension index respectively. Additional notation introduced by Algorithm 2 includes:

- E_{ik}, a matrix containing the spatial coordinates of each node
- C_{ij}, a connectivity matrix indicating the nodal indices of all the neighbors of each node
- I_{ij}, an inverse connectivity matrix indicating the bond indices a node occupies in the neighborhoods of its various neighbors (necessary for the determination of $\underline{t}[\hat{\mathbf{x}}, \mathbf{x}]$ in Eq. 1.12)
- S_{ik}, a boolean array indicating whether a given node has an applied displacement in a given direction
- D_{ik}, an array containing the applied displacement boundary condition values.

Finally, an influence function $\omega = \frac{1}{\underline{x}}$ has been selected and the relevant computations have been simplified accordingly.

The parallel implementation of Algorithm 2 is very straightforward, with each line corresponding to a kernel function, and the associated grid launched with dimensionality corresponding to that of the output tensor. As such, each thread in the grid fills a single element of the output tensor. Each of these output tensors must be stored in global memory so that successive kernels can utilize the data they contain.

As Section 3.4 will explore, this approach is able to achieve good computational efficiency for large problems, as it performs very few redundant calculations. However, this efficiency is achieved by using large amounts of global memory to store re-used values, which entails a computational overhead, and also strongly limits the problem size a given GPU can manage.

Due to the straightforward methodology used to formulate this method and the lack of optimizations it entails, we refer to this as the "Naive" approach.

Algorithm 3 Extension of PeriPy Method to State-Based Peridynamics

function UPDATEDILATION(C, E, u, m, ϕ, V)
    $\xi_{ijk} = E_{C_{ij}k} - E_{ik}$
    $x_{ij} = \sqrt{\sum_k \xi_{ijk}^2}$
    $\eta_{ijk} = u_{C_{ij}k} - u_{ik}$
    $e_{ij} = \sqrt{\sum_k (\xi_{ijk} + \eta_{ijk})^2} - x_{ij}$
    $\phi_{ij} = \frac{x_{ij}+e_{ij}}{x_{ij}} < s_c \,\&\, \phi_{ij}$
    $\theta_i = \frac{3}{m_i} \sum_j e_{ij} \phi_{ij} V_{C_{ij}}$
    return ϕ, θ
end function
function UPDATEFORCE(C, E, u, θ, m, ϕ, V, b)
    $\xi_{ijk} = E_{C_{ij}k} - E_{ik}$
    $x_{ij} = \sqrt{\sum_k \xi_{ijk}^2}$
    $\eta_{ijk} = u_{C_{ij}k} - u_{ik}$
    $e_{ij} = \sqrt{\sum_k (\xi_{ijk} + \eta_{ijk})^2} - x_{ij}$
    $f_{ik} = b_{ik} + \sum_j \left(\left(2k + 15\mu\right) \left(\frac{\theta_i}{m_i} + \frac{\theta_{C_{ij}}}{m_{C_{ij}}} \right) + \left(\frac{1}{m_i} + \frac{1}{m_{C_{ij}}} \right) \frac{15\mu e_{ij}}{x_{ij}} \right) \frac{\xi_{ijk}+\eta_{ijk}}{e_{ij}+x_{ij}} \phi_{ij} V_{C_{ij}}$
    return f
end function
function UPDATEDISPLACEMENT(u, $\dot{u}$, f^{t-1}, f^t, S, D)
    $\dot{u}_{ik} += \frac{\Delta_t}{2}(f_{ik}^{t-1}/\rho - \zeta \dot{u}_{ik})$
    if S_{ik} **then**
        $u_{ik} = D_{ik}$
    else
        $u_{ik} += \Delta_t \dot{u}_{ik}$
    end if
    $\dot{u}_{ik} += \frac{\Delta_t}{2}(f_{ik}^{t}/\rho - \zeta \dot{u}_{ik})$
    return u, $\dot{u}$, f^{t-1}
end function

for tt = 1:NT **do**
    ϕ, θ = UPDATEDILATION(C, E, u, m, ϕ, V)
    f^t = UPDATEFORCE(C, E, u, θ, m, ϕ, V, b)
    u, $\dot{u}$, f^{t-1} = UPDATEDISPLACEMENT(u, $\dot{u}$, f^{t-1}, f^t, S, D)
end for

3.3.2 Extension of PeriPy to State-Based Peridynamics

As discussed in Sec. 1.1, the PeriPy package is the most recent & comprehensive open-source software currently available for GPU-based PD. However, the package is an implementation of bond-based PD; this section will detail how Algorithm 2 may be optimized & modified to fit within the PeriPy framework. This adaptation is summarized in Algorithm 3.

The primary optimization the PeriPy package makes, relative to the approach taken in Algorithm 2, is a kernel fusing which reduces the amount of data that needs to be stored in global memory, and therefore also the number of accesses to global memory that need to be made. Values stored in global memory, particularly those associated with bonds like ξ and η, are replaced with on-the-fly calculations that only require register memory.

In the state-based equivalent of this kernel fusing & on-the-fly computation, ξ and x are no longer pre-computed but recalculated at every iteration. These calculations as well as those determining η, e, ϕ, and θ may then be combined into a single kernel, `updateDilation`. The only bondwise arrays that are referenced in global memory are the connectivity array C and damage state ϕ. ξ, η, x, and e will be recalculated in the next kernel, `updateForce`, which is preferable to storing these values in global memory.

With proper care, calculation of e^d, t, and f may all be performed within a single `updateForce` kernel, storing only f in global memory. The difficulty here rises from the fact that f_{ik} depends on both t_{ij} and t_{ji}. A naive approach (such as that of Algorithm 2) would require a separate kernel for the calculation of t and for the output to be stored in global memory. However, since $e_{ij} = e_{ji}$ and $x_{ij} = x_{ji}$, f_{ik} may be written instead as a function θ_i and θ_j by decomposing e^d:

$$\begin{aligned} f_{ik} = b_{ik} + \sum_j \Bigg(\Big(2k + 15\mu\Big)\Big(\frac{\theta_i}{m_i} + \frac{\theta_j}{m_j}\Big) + \\ \Big(\frac{1}{m_i} + \frac{1}{m_j}\Big)\frac{15\mu e_{ij}}{x_{ij}}\Bigg)\frac{\xi_{ijk} + \eta_{ijk}}{e_{ij} + x_{ij}}\phi_{ij}V_j \end{aligned} \tag{3.1}$$

$$\theta_i = \sum_j f(\mathbf{x}_i, \mathbf{x}_j, \mathbf{u}_i, \mathbf{u}_j) \tag{3.2}$$

The `updateDisplacement` kernel performs the remaining computations for the Velocity-Verlet integration. The set of arrays that must be stored in global memory has thus been reduced to C_{ij}, E_{ik}, u_{ik}, m_i, ϕ_{ij}, V_i, θ_i, $\dot{u}_{ik}$, f_{ik}^{t-1}, f_{ik}^{t}, b_{ik}, S_i, and D_{ik}. Of these, only C and ϕ scale with horizon size, and therefore typically comprise the majority of the global memory allotment.

The other major change PeriPy makes is to the kernel grid structure, along with use of shared memory, for the evaluation of sum reductions within kernels. In this state-based adaption of PeriPy, both the `updateDilation` and `updateForce` kernels are launched on a 1-dimensional grid, where each block corresponds to a node (i) and each thread corresponds to a neighbor (j) of the node its block represents. Each of the bondwise memory reads & calculations across the neighbors j can then be performed in parallel, and a parallel sum-reduction over the bonds using shared memory can be used to evaluate θ and f [48]. The effects of this change are discussed in further detail in Sec. 3.3.3. The `updateDisplacement` kernel grid is the same as in Algorithm 2, where a 2-dimensional grid is launched with each thread corresponding to an element of the u_{ik} output.

3.3.3 Memory-Optimized Approach

As Section 3.4 will explore, the adaptation of the approach used by the PeriPy software to state-based PD significantly reduces the amount of memory a problem size uses, relative to the naive approach. However, further optimizations are certainly possible. In particular, if the undeformed node layout is restricted to a regular grid, the memory requirements of the method can be substantially reduced. The following section described this novel approach, termed the "Memory-Optimized" method, which as the name suggests uses this assumption of a regular grid to minimize the memory the process consumes. This alteration, in addition to a few other performance-critical optimizations discussed below, substantially increases the

Algorithm 4 Memory-Optimized Approach

function UPDATEDILATION(u, ϕ, χ)
  if χ_i **then**
    $C_{ij} = i + J_j$
    $\eta_{ijk} = u_{kC_{ij}} - u_{ki}$
    $e_{ij} = \sqrt{\sum_k (\xi_{jk} + \eta_{ijk})^2} - x_j$
    $\phi_{ji} = \frac{x_{ij}+e_{ij}}{x_j} < s_c \,\&\, \phi_{ji}$
    $\theta_i = \frac{3}{m} \sum_j e_{ij}\phi_{ji} V$
  end if
  return ϕ, θ
end function
function UPDATEFORCE(u, θ, ϕ, χ, N, $\dot{u}$, f^{t-1}, S)
  if χ_i **then**
    $C_{ij} = i + J_j$
    $\eta_{ijk} = u_{kC_{ij}} - u_{ki}$
    $e_{ij} = \sqrt{\sum_k (\xi_{jk} + \eta_{ijk})^2} - x_j$
    $f_{ki} = B_{N_i} + \sum_j \left(\left(\frac{2k+15\mu}{m} \right) \left(\theta_i + \theta_j \right) + \frac{30\mu e_{ij}}{m x_j} \right) \frac{\xi_{ijk}+\eta_{ijk}}{e_{ij}+x_j} \phi_{ji} V$
    $\dot{u}_{ki} += \frac{\Delta_t}{2}(f_{ki}^{t-1}/\rho - \zeta \dot{u}_{ki})$
    if $S_{ki} >= 0$ **then**
      $u_{ki} = D_{S_{ki}}$
    else
      $u_{ki} += \Delta_t \dot{u}_{ki}$
    end if
    $\dot{u}_{ki} += \frac{\Delta_t}{2}(f_{ki}^{t}/\rho - \zeta \dot{u}_{ki})$
  end if
  return u, $\dot{u}$, f^{t-1}
end function

for tt = 1:NT **do**
  ϕ, θ = UPDATEDILATION(u, ϕ, χ)
  u, $\dot{u}$, f^t = UPDATEFORCE(u, θ, ϕ, χ, N, $\dot{u}$, f^{t-1}, S)
end for

problem size a given GPU can manage, while also significantly improving performance for a given problem size. This approach is summarized in Algorithm 4.

Global Memory Reduction

The primary advantage of performing PD simulation on a regular grid is that it allows the evaluations over nodal neighborhoods to become a stencil operation, rather than a scatter-gather process. Instead of storing an entire connectivity matrix C_{ij} in global memory, a small connectivity stencil J_j can be defined such that $C_{ij} = i + J_j \forall i$ and stored in constant memory. Similarly, ξ also no longer has any dependence on i and may be stored in constant memory as a stencil, eliminating the need to store the coordinate matrix E_{ik} in global memory. The only global-memory cost of this modification is the need for a boolean occupancy array χ_i indicating whether a node on the regular grid lies within the material domain.

The introduction of a regular grid also removes the need for the storage of a V_i array indicating nodal volumes; all nodes have the same volume so V may be precomputed as a single value, stored in constant memory. Similarly, m is constant for all nodes within the bulk of the material. Since the PD surface effect [23, 49] is being neglected in each of the approaches described here, assuming m to be constant throughout the entire domain does not significantly affect the accuracy of the simulation. Future work may address surface effect correction within this implementation, and would need to account for a lower value of m near material surfaces.

Finally, the global memory requirements for the boundary condition arrays may be reduced relative to Algorithm 3. The body force matrix b_{ik} can be replaced with an index array N_i, which stores a negative integer for all nodes with no applied body force and a positive index n for nodes with applied body forces, which indexes into an array $B_{n,k}$ containing a list of all distinct body forces. Since for most problems the number of distinct body forces is far less than the number of nodes, this reduces the per-node memory requirement for body force storage by a factor corresponding to the number of spatial dimensions. Similarly, for displacement boundary conditions, the boolean array S_{ik} (indicating whether a node has an applied displacement in a given dimension) can be replaced with an integer array of the same size, returning a positive integer m for nodes with applied displacements that indexes

into a list of distinct displacements, D_m. As with applied forces, the number of distinct applied displacements is typically much less than the number of nodes, so for most problems this modification significantly reduces the per-node storage requirements for displacement boundary conditions.

Bit-Packing & Data Type Management

After performing the optimizations described in Sec. 3.3.3, the only remaining arrays that scale with problem size & must be stored in global memory are u_{ik}, ϕ_{ij}, χ_i, θ_i, N_i, $\dot{u}_{ik}$, f_{ik}^{t-1}, f_{ik}^{t}, and S_{ik}. The actual memory requirements of these arrays can be further reduced by careful selection of data type. If a user wishes to run a high-precision simulation with double precision, u_{ik}, θ_i, $\dot{u}_i$, f_i^{t-1}, and f_i^t must all be double-precision floating point arrays to achieve maximum simulation precision (8 bytes per node and dimension). N_i and S_{ik} need to be able to contain unique indices for each distinct applied force and displacement, respectively. Single-precision integers for these arrays would be more than sufficient for any simulation, but for the majority of problems that have only a few distinct boundary conditions, a single-byte integer for each element is usually acceptable. Finally, ϕ_{ij} and χ_i are boolean arrays, which consume one byte per element. However, a boolean value can be represented with a single *bit*, so to minimize memory usage, a bit-packing [50] function is used to only require 1 byte per 8 elements of ϕ_{ij} and χ_i. This is a critical operation as ϕ_{ij} is the only global memory allocation that scales with horizon size, and bit-packing substantially reduces its memory consumption.

Thread Workload Optimization

PeriPy aims its thread workload allocation at maximizing potential parallelism. By parallelizing the summations over neighbors in `updateDilation` and `updateForce`, and parallelizing by both node and dimension in `updateDisplacement`, PeriPy ensures that if an infinite number of threads can be concurrently launched, maximum parallelization and therefore optimal performance will be achieved. However, the best GPUs available cannot launch

much more than 100,000 concurrent threads, which is fewer than the number of nodes in most 3D simulations. When the GPU runs out of available active threads, it serially loops over queued blocks, as discussed in Sec. 3.2. In such cases, maximum parallelism can also be achieved with a simpler grid layout, with each thread representing a single node, and looping over neighbors serially withing the kernels to perform the neighborhood summations. This layout will be referred to as Node-Per-Thread (NPT).

In `updateDisplacement`, the PeriPy and NPT approaches achieve very similar results, as theoretically expected: the PeriPy approach fills one output per thread, and each hardware core must process approximately D*N/C threads, where D is the number of spatial dimensions, N is the number of nodes, and C is the number of threads that can be run concurrently (roughly proportional to the number of hardware cores a GPU has). The NPT approach fills D outputs per thread, and each hardware core processes N/C threads. This results in each approach performing an equal number of concurrent computations, and thus achieving very similar performance. The NPT approach turns out to be slightly faster, because the longer kernel functions and fewer kernel launches compound less scheduler overhead and provide more opportunity for the scheduler to hide latencies.

Similar logic applies to the bondwise-parallelization of PeriPy in `updateDilation` and `updateForce`: if the number of threads launched exceeds the number the GPU can concurrently run, PeriPy does not achieve a higher degree of parallelism than the NPT approach. However, the parallel reductions have a secondary effect that causes PeriPy's performance to lag significantly behind NPT: while the binary sum reduction within a block is being performed, after a thread has completed is contribution to the sum it sits idle until the entire sum is completed. With NPT, no thread inherently sits idle at any point, which yields a substantial performance increase. This gap is narrowed when damage is present, as the binary sum reduction mitigates the thread divergence caused by damage, but the NPT remains significantly faster.

Additionally, with the simpler NPT kernel grid layout used for all kernels, the `updateDisplacement` kernel can be fused into the `updateForce` kernel, which removes the need to store f^t in global

memory.

Memory coalescence

GPU latency during memory reads & writes can be greatly reduced by coalescing memory accesses. Each warp on the GPU can load a total of 128 successive bytes (32 single-precision values) from memory in a single transaction. Therefore, if each of the 32 threads in a warp makes a simultaneous, consecutive read from global memory, this occurs as a single load from global memory in what is known as a fully coalesced memory access. Similarly, if all threads in a warp access the same memory address at the same time, this occurs as a single read that is broadcast to all threads in the warp.

Fortunately, Algorithm 4 is very well-posed for memory coalescing. In particular, within a given warp and in the bulk of the material, consecutive threads in a warp will simultaneously access values of u and θ for consecutive neighbors, allowing for fully coalesced memory accesses everywhere except for small sections near the boundary of the domain. If the domain dimensions are divisible by 32, memory accesses can be fully coalesced. The only modification relative to Algorithm 3 that must be made to achieve this is transposing multidimensional arrays: the GPU represents all arrays as 1D, and when multidimensional arrays (such as u, f, etc.) are initialized their dimensions must be ordered such that when they are flattened to 1D, spatially consecutive points are still consecutive in memory. Using the Python package Numpy to initialize these arrays and PyCuda to send them to the GPU, the array orientation necessary for flattening into consecutive memory is achieved by specifying the nodal index to be the final dimension. The orders of array indices in Algorithm 4 have been updated to reflect this.

Shared Memory Usage

As discussed in Section 3.2, shared memory has significantly higher access speed than global or constant memory, and can therefore provide substantial accelerations if it can be effectively utilized. However, the use cases for shared memory are restricted by the

fact that shared memory only persists for the lifetime of the block that allocates it. PeriPy utilizes shared memory in its bondwise sum reductions, storing the summation data in shared memory so all threads on a block (corresponding to the neighbors of a node) can contribute to the summation with low overhead. While this is certainly faster than using global memory for the same process, as explored in Sec. 3.3.3, the block sum reduction in general is not an optimal approach.

A variety of alternative approaches were attempted to utilize shared memory for performance acceleration with the NPT kernel grid layout, most of which were unsuccessful due to decreased memory coalescence, increased thread latency while waiting for blocks to synchronize, and so on. The only approach that successfully accelerates the process is to load the directional bond length stencil ξ_{jk} into shared memory from constant memory at the start of the `updateDilation` and `updateForce` kernels, then have each thread access the shared memory allocation throughout the kernel. Since the block can be sized such that each thread only has to fill one element of the shared memory array, this significantly reduces the number of constant memory accesses each thread makes, resulting in a moderate performance improvement.

Expensive Math Operations

With the optimizations described above, most of the major bottlenecks of this GPU implementation have been removed. As such, one of the most expensive remaining steps is the evaluation of square roots required by both the `updateDilation` and `updateForce` kernels. In Algorithm 3, each of these kernels must compute 2 square roots, computing original & deformed bond lengths. In Algorithm 4, the former is turned into a precomputed stencil x_j, as the regular grid of this implementation means the undeformed length of each bond j will be constant for all nodes i. The second square root, determining deformed bond length, cannot be removed. However, if single-precision arithmetic is acceptable in a user's application, this square root can be greatly accelerated with CUDA's fast math functionality for single precision. This is a major source of the speedup between double- and

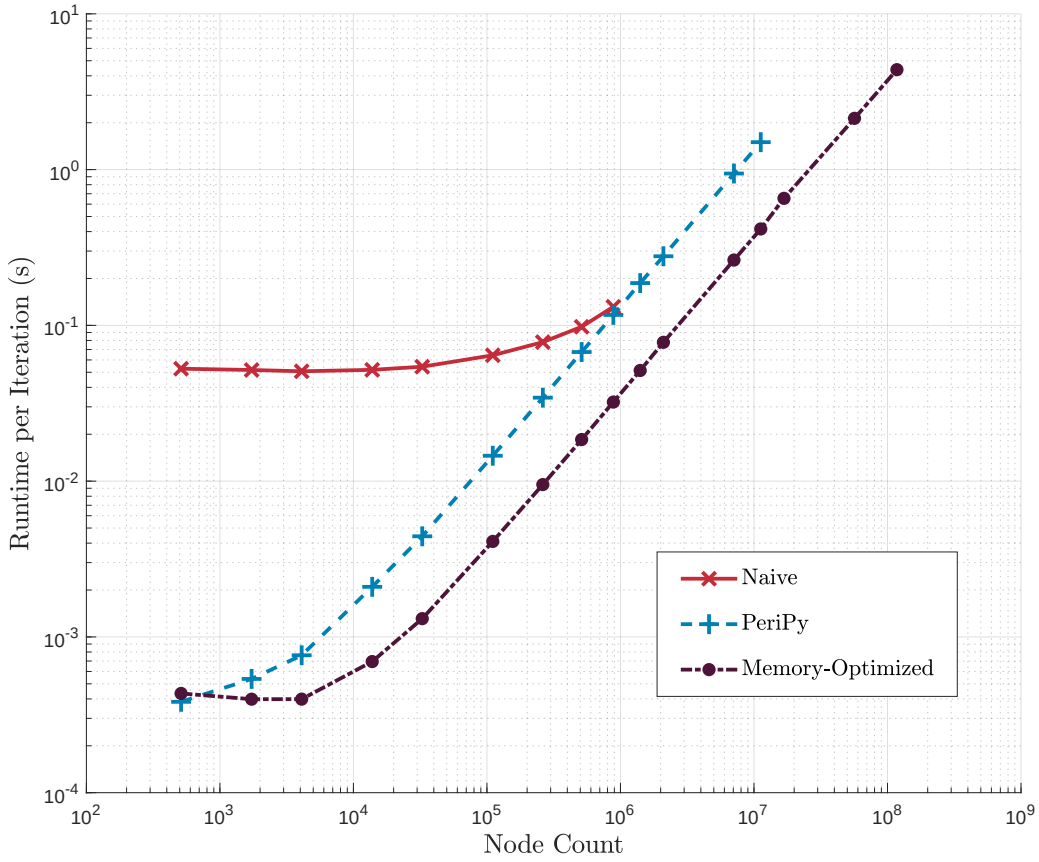

Figure 3.2: Benchmarking results are depicted for each of the 3 presented implementations, tested on problem sizes from $\sim$ 1000 - 100,000,000 nodes with a Nvidia GeForce RTX 2080 Ti GPU. The specific upper limit of each curve is indicative of the maximum problem size the corresponding implementation can represent on this GPU used for the benchmark experiment.

single- precision that is explored in Sec. 3.4.

3.4 Results

The two primary features of the 3 implementations presented above are computational efficiency (how fast a given GPU can simulate a specified problem size) and memory efficiency (how large of a problem a given GPU can simulate).

Memory efficiency can be evaluated by examining the amount of memory used per node.

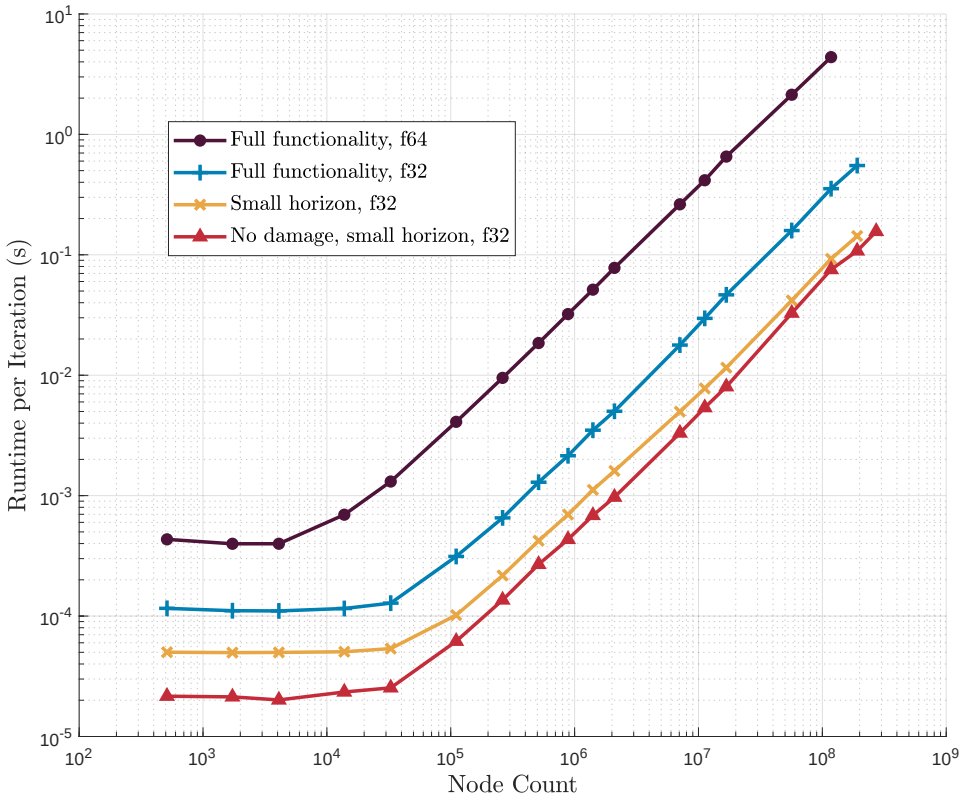

Figure 3.3: The memory-optimized approach is benchmarked on problem sizes from $\sim$ 1000 - 300,000,000 nodes with varying simulation parameters. Specifically, the curves depict the performance improvements yielded by reducing precision from double to single, reducing horizon size, and removing damage modelling, successively.

Based on the dimensions & types of the global memory arrays used by each implementation in Section 3.3, the per-node memory requirements in bytes for each are

$$\text{Naive: } 2MDP + 4MP + 5M + 7DP + 3P + D \tag{3.3a}$$

$$\text{PeriPy: } 5M + 8DP + 3P + D \tag{3.3b}$$

$$\text{Memory-Optimized: } \frac{1}{8}M + 3DP + P + D + \frac{9}{8} \tag{3.3c}$$

where P is the size in bytes of the problem's precision (4 for single-precision, 8 for double), M is the number of neighbors per node, and D is the number of spatial dimensions. For a double-precision, 3D problem with 128 neighbors per node (which corresponds to $\delta = 3L$ on a rectangular grid of points with a spacing of L), a GPU with 11 GB of memory (such as the NVIDIA GeForce RTX 2080 Ti used in the benchmarking experiment below) is capable of evaluating up to approximately 1 million nodes with Algorithm 2, 12.8 million nodes with Algorithm 3, and 110 million with Algorithm 4. These theoretical calculations are reflected in the upper domain limit of each curve in Figure 3.2.

To evaluate performance, a benchmarking experiment is necessary. Fortunately, most problem-specific simulation features (geometry, material properties, etc.) don't affect the per-iteration performance of each method, so a simple test problem consisting of a 3D, rectangular, linearly elastic medium under uniaxial tension can be used to characterize the performance of each method. One simulation feature which may impact performance is extensive crack propagation in the simulated domain, which causes threads within a block to desynchronize, leading to the thread divergence discussed in Sec. 3.3.3. However, the effect of thread divergence can be minimized by specifying small block sizes, which decreases the amount of time a thread sits idle while it waits for other threads in its block to complete execution. Moreover, if the number of blocks containing nodes with damaged bonds is small relative to the total number of blocks, the effect of thread divergence is negligible. Therefore, a simple benchmark without crack propagation remains an appropriate choice for performance evaluation.

The results of this benchmark for each implementation are depicted in Figure 3.2, evaluating a range of problem sizes for a double-precision simulation on a Nvidia GeForce RTX 2080 Ti, with a horizon size of 3L. As expected, the naive approach is outperformed by both of the other implementations. However, as discussed above this approach does achieve rea-

sonable performance in large scale, on this GPU nearly breaking even with PeriPy around 1,000,000 nodes. This problem size limit is indicated by the upper domain bound of the curve for the naive approach in Fig. 3.2. It is possible that the naive approach may even slightly outperform PeriPy at larger scales; however, it is strongly memory-limited and cannot simulate more than 1,000,000 nodes on this GPU. As such, for problems where irregular nodal grids are necessary, PeriPy is generally the optimal choice, particularly for small problem scales or if GPU computing resources are limited. However, if a regular grid of points is acceptable, the memory-optimized approach yields significantly faster and more memory-efficient results than the other methods for all problem sizes. For large-scale simulations, the memory-optimized approach yields a 3.6x acceleration over PeriPy, increases the problem size a given GPU can handle by 10.4x relative to PeriPy.

Besides the implementation a user might choose, a number of other simulation parameters have a strong effect on performance. Specifically, floating point precision, neighbor number, and inclusion of damage modelling each have a particularly strong effect on both simulation speed and memory-efficiency. For the benchmark performed in Figure 3.2, all methods use double precision, a reasonably large number of neighbors, and include damage modeling. Figure 3.3 depicts the changes in performance in the memory-optimized approach by successively reducing precision to single, reducing neighbor number to 32 (corresponding to a horizon size of double the grid spacing), and removing the damage modelling feature. Precision reduction yields an acceleration of 12.4x and a 65% increase in memory efficiency. Neighbor number provides a further 4x acceleration, and removal of damage modelling yields a modest 1.2x acceleration and an additional 42% memory-efficiency increase. Combined, these changes yield a 63x acceleration and a 3.2x increase in the problem size a given amount of memory can represent.

Finally, it should be noted that the implementations presented and characterized here do not contain some features that a user might be interested in, such as plasticity or surface effect correction. While these features can certainly be incorporated into each approach, they will require additional memory allocations and accesses, and therefore produce some

reduction in computational and memory efficiency. Developing efficient implementations of these features that minimizes their impact will be the focus of future work.

3.5 Discussion

This work has presented and compared three separate implementations of state-based PD on the GPU. The naive approach, based off an earlier work [45], minimizes redundant computations and achieves reasonable performance at large scale. However, this approach is very memory-expensive and is therefore not capable of large-scale simulation if computing resources are limited.

The second approach is an extension of the open-source project PeriPy [51], expanded to represent state-based PD. We have demonstrated that PeriPy may be readily adapted to state-based PD with minor modification. As predicted by [51], the state-based benchmarking in Section 3.4 is almost exactly twice as slow as the bond-based approach of the original project. PeriPy has the flexibility of being able to be applied to an irregular grid of points, and outperforms the naive approach on small-scale problems and in memory-efficiency for large-scale problems. As such, this method is the optimal choice for cases where an irregular grid is required.

Finally, a novel, memory-optimized implementation is developed which aims to minimize the global memory requirements of the program and thereby maximize the problem size a single GPU can manage. While this method can only be used on regular grids of points, it outperforms PeriPy at all problem scales. Moreover, it achieves its purpose by increasing the manageable problem size by an order of magnitude relative to PeriPy, and two orders of magnitude relative to the naive approach. Therefore, this memory-optimized approach is a significant advancement in computational peridynamics, and enables extremely efficient simulation of large-scale problems with limited GPU resources.

Chapter 4

APPLICATIONS: TOPOLOGY OPTIMIZATION & NONLINEAR ELASTICITY

4.1 Background

As mentioned earlier, the computational expense of peridynamics is one of its greatest limitations, and the substantial reduction in this expense via GPU parallelism has the potential to greatly expand the utility of peridynamics. The vast majority of existing works on peridynamics focus on its application to problems dealing with crack propagation & other damage-related phenomena, as such problems are among the few varieties for which peridynamics yields enough of a benefit to warrant the computational expense [5–7]. However, with a GPU-based approach greatly reducing this expense, other applications suddenly become far more feasible alternatives to traditional FEM-based approaches. Of particular interest are applications in topology optimization, and structural analysis problems dealing with nonlinearities. These two use cases have been selected to highlight the significance of the parallelization developed in Chapter 3, as it is this parallelization which makes solution with peridynamics tractable.

Nonlinearities in structural analysis can arise from a variety of sources, such as nonlinear material constitutive relationships [52] and the nonlinear strain-displacement relationships exhibited by large deformations [53]. Typically, however, the finite element method ignores these nonlinearities, as linearizing the governing equations allows a problem to be assembled into a system of linear algebraic equations, for which efficient and well-established solvers exist [54]. If one wishes to use FEM to evaluate nonlinear deformation problems, more complex analysis is required. Typically, some incremental and/or iterative approach is used to identify a solution to the nonlinear governing equations, where each iteration requires solving a linearized system of equations, thus quickly accumulating a large computational expense [53].

Peridynamics offers an attractive alternative to FEM for this use case, as the nodal force evaluation process peridynamics uses can be very naturally extended to represent a nonlinear relationship between nodal displacements and the corresponding forces. However, exploration of this application of peridynamics has thus far been limited due to the his-

torical computational expense of peridynamic simulation. In 2005, Weckner & Abeyaratne demonstrated that, in contrast to linear FEM, peridynamics naturally produces nonlinear dispersion relations in a vibrating beam [55]. More recently, Xu et al. demonstrated a peridynamic formulation incorporating nonlinear hyperelastic material models [56]. Finally, in 2021, Nguyen & Oterkus demonstrated agreement between peridynamic and FEM results for large deformation problems producing geometric nonlinearities [17]. This section will similarly focus on applications to geometrically nonlinear problems, as the work published by Nguyen & Oterkus only covers a few basic validation problems, and does not evaluate whether peridynamics demonstrates an advantage over FEM-based approaches. Further development and exploration of peridynamics for large deformation problems is warranted, particularly with the introduction of an accelerated peridynamic solver. Chapter 4 will explore the computational benefits yielded by the GPU peridynamic framework in geometrically nonlinear analysis, providing comparisons with various state-of-the-art FEM-based solutions to contextualize the advantage peridynamics provides.

Topology optimization is the systematic modification of the layout of material within a design domain to optimize some performance criteria [57]. The rapid development of additive manufacturing technologies in recent decades and the flexibility it allows for design as brought new relevance to this field, as the intricate designs topology optimizations often yield can now be far more readily fabricated. However, the computational complexity of optimization methods remains a major obstacle in their practical, widespread application towards engineering problems [58]. Specifically, topology optimization typically involves iteratively simulating the behavior of the design domain then modifying the topology, repeating until an optimal topology is achieved [57]. The finite element method is the most common method used for the simulation step, and a single FEM simulation can be quite expensive; for large domains, the need for numerous FEM simulations for an optimization problem can be prohibitively expensive [54]. This computational cost is compounded for situations dealing with more complicated physics, such as crack propagation & nonlinear elasticity problems, which can require numerous iterations of FEM solution within a single topology optimization

iteration.

An efficient peridynamics solver, such as the GPU-based implementation developed in Chapter 3 of this work, could be an attractive solution to this problem, perhaps in general cases, but certainly for the classes of problems for which peridynamics is already better-suited than FEM. Recent work has shown that peridynamics can be ported into the topology optimization process relatively easily, replacing the finite element step. Additionally, peridynamics has been proven successful for topology optimization in more complex cases involving cracking [33] and multi-material bodies [34]. This chapter will expand upon these promising works by demonstrating for the first time that peridynamics automatically produces the results of geometrically nonlinear topology optimization, comparing to solutions obtained with nonlinear FEM-based solution methods [59].

In addition to validating new use cases of peridynamics in nonlinear topology optimization, this chapter characterizes the computational benefits yielded by the GPU peridynamic framework, both in the context of nonlinear optimization, and for topology optimization in general. The existing published works on peridynamic topology optimization do not explore the computational aspects of their work, and as such do not make the argument that peridynamics offers an advantage over FEM-based methods for the general topology optimization use case [33, 34]. Prior to this work, the state-of-the-art for high-performance topology-optimization involves GPU-based FEM solutions in the structural evaluation step [60, 61]. This chapter will demonstrate that GPU-based peridynamic solutions for the structural evaluation step reach parity with and even surpass state-of-the-art FEM solution methods. Additionally, in cases involving more complex phenomena like nonlinear deformations, the FEM-based state-of-the-art has not advanced to include high performance computing techniques. The enormous benefit yielded by this novel GPU-based peridynamics framework over serial FEM approaches in such use cases will be demonstrated here as well.

4.2 Nonlinear Analysis

In order to solve problems involving nonlinear deformations with the finite element method, an inherently linearized process, solutions are obtained incrementally, which when coupled with the large matrix inversions used by FEM can end up being very computationally expensive. Point-based methods like peridynamics however can naturally incorporate nonlinear relationships between displacement and force with little to no additional computational expense. With the acceleration of peridynamics yielded by the GPU framework of Chapter 3 of this work bringing the computational expense of peridynamics closer to parity with that of a basic, linear FEM simulation, it is possible for peridynamic solvers to outperform a comparable nonlinear FEM simulation.

In their recent work, Nguyen & Oterkus provide a basic demonstration of the application of peridynamics towards large-deformation problems, showing good agreement with nonlinear FEM results [17]. Specifically, their work provides 1-, 2- and 3-D formulations of peridynamics, introducing the peridynamic equivalents of true strain & true stress and describing a method of recalculating nodal volumes under the applied large displacements. While these developments are important advancements pioneering the use of peridynamics in large deformation analysis, further refinement and demonstration is required. In particular, this work seeks to advance Nguyen & Oterkus' studies in two key ways: a) by resolving a minor error in their large-deformation formulation, described in further detail below, and b) by demonstrating the computational advantage yielded by high-performance peridynamics compared with the state-of-the-art for large-deformation analysis using FEM.

4.2.1 Formulation Modification

Nguyen & Oterkus provide the following discrete formulation of bond-based peridynamics for large-deformation, 1-D uniaxial tension problems:

$$\rho\ddot{u}_i = \sum_j \frac{2E}{A\delta_H^2} ln(1+s)\frac{\underline{x}}{\underline{e}}\frac{\underline{\xi}_x}{\underline{e}}dV + b_i \tag{4.1}$$

where δ_H is the radius of the nodal horizon $\mathcal{H}$, A is the cross-sectional area of the bar, and s is the bond stretch $\frac{\underline{\underline{e}}-\underline{\underline{x}}}{\underline{\underline{x}}}$ (all other symbols here are defined in Section 1.2). The classical formulation of the same problem is given by Hooke's Law

$$\sigma_t = E\epsilon_t \tag{4.2}$$

where σ_t is true stress, related to engineering stress σ by $\sigma_t = \sigma(1+\epsilon)$, and ϵ_t is true strain, related to engineering strain ϵ by $\epsilon_t = ln(1+\epsilon)$. So, the resulting classical governing equation for this problem in terms of the familiar engineering stress & strain becomes

$$\sigma = E\frac{ln(1+\epsilon)}{1+\epsilon} \tag{4.3}$$

where Eq. 4.1 and Eq. 4.3 should become equal at equilibrium, where $\ddot{u}_i = 0$. In Eq. 4.1, it can be seen that s is equivalent to the engineering strain ϵ, and the $\frac{\underline{\underline{x}}}{\underline{\underline{e}}}$ term is equal to $\epsilon+1$. Examining the node i at the end of the uniaxial geometry where the loading force F is applied, the body force density b_i is F/dV. Since ξ_x is the only component of the deformed bond vector ξ, the $\frac{\xi_x}{\underline{\underline{e}}}$ term for all neighbors j of this end node i becomes -1. Eq. 4.1 becomes

$$\frac{F}{dV} = \sum_j \frac{2E}{A\delta_H^2}\frac{ln(1+\epsilon)}{1+\epsilon}dV \tag{4.4}$$

If the undeformed bond length is L, dV can be written as $A*L$, and the summation can become a simple multiplication by a factor of δ/L, as all other terms within the summation should be the same for all bonds (only for this simple uniaxial tension case). The resulting large-deformation constitutive relationship is

$$\frac{F}{A} = \frac{2L}{\delta}E\frac{ln(1+\epsilon)}{1+\epsilon} \tag{4.5}$$

where of course $\frac{F}{A}$ is equivalent to σ. Therefore, for this formulation to properly correspond to the classical constitutive relationship in Eq. 4.3, a constraint is imposed on the

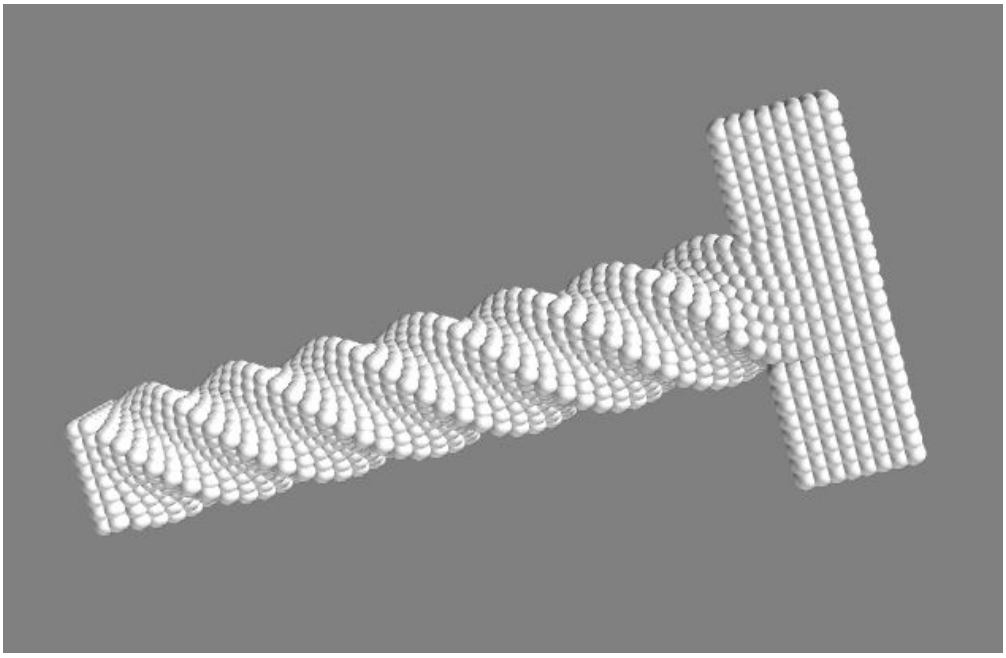

Figure 4.1: Simulation results for a T-shaped beam undergoing a large torsional force, exhibiting large-deformation nonlinear behavior.

horizon radius; $\delta = 2L$. Such a constraint should not be necessary, and would be undesirable in many peridynamics use cases. Moreover, Nguyen & Oterkus report using the typical $\delta = 3.015L$ in their numerical experiments, hence indicating a discrepancy between the results they report (which match the classical solution) and the formulation they provide.

To produce proper constitutive correspondence without restricting the horizon, an alternative bond constant can be formulated that eliminates this issue by removing the $\frac{2L}{\delta}$ term from Eq. 4.5 and backtracking to Eq. 4.4, producing the following governing equation:

$$\rho\ddot{u}_i = \sum_j \frac{E}{AL\delta_H} ln(1+s)\underline{\underline{\frac{x}{e}}}\,\underline{\underline{\frac{\xi_x}{e}}}\,dV + b_i \tag{4.6}$$

4.2.2 Comparison with Nonlinear FEA State-of-the-Art

Despite recent advances in the computational state-of-the-art of FEM, incorporating conjugate-gradient linear solvers to invert stiffness matrices in parallel on the GPU (which will be discussed further in Section 4.3), at the time this document was written, no GPU-based solvers have been published for application to nonlinear structural analysis [62]. Instead, the state-of-the-art for problems dealing with nonlinearity generally focuses on accelerating computation via order-reduction [62, 63]. In [63], Maksum et al. evaluate both a compact and efficient, albeit serial, full-order nonlinear FEM model alongside a reduced-basis parametrized nonlinear structural analysis undergoing large deformations.

While such reduced-order models are extremely computationally efficient, they do not characterize the full behavior of a model. Additionally, reduced-order models typically require tailoring to each specific problem they are used in, and pose difficulty for use in cases with complex geometries. As such, despite the success of such methods, efficient methods for producing full-fidelity simulations of nonlinear behavior remains an open problem. Here we compare the state-of-the-art nonlinear solver results presented by Maksum et al. for a 3-dimensional test problem consisting of a T-beam under a large torsional load with the results of a corresponding peridynamic model run under the GPU framework described in Chapter 3. Figure 4.1 depicts the deformed geometry obtained in peridynamic simulation, and Table 4.1 contains the timing information presented by Maksum et al. as well as for the peridynamic simulation.

	Full-Order Nonlinear FEM	Reduced-Order Model	Peridynamics
Computation Time (s)	1.14E+03	4.30E-01	4.77E+00

Table 4.1: Computation of state-of-the-art processes for Nonlinear FEM and Reduced Order Modeling, in comparison with Peridynamics.

4.2.3 Handed-Shearing Auxetics

Handed-Shearing Auxetics (HSAs) are a very recent emergent technology in the field of meta-materials for engineering applications, with a wide variety of use cases. These structures demonstrate unique mechanical behavior, undergoing longitudinal expansion or contraction under the application of radial twist. Additionally, the radial expansion or contraction of the structure is a function of the direction of the applied twist. These distinctive properties make HSAs particularly interesting for applications in soft robotics gripping applications, where they have been demonstrated to yield substantial utility [64].

For such applications in robotics, having an effective model of the structural behavior is highly desirable for the developing novel actuator designs utilizing the HSAs. Very recently, researchers developed a reduced-order model which accurately describes the macroscopic positional behavior of HSA structures, mapping applied rotation to axial extension [65]. However, efforts to develop a full-fidelity FEM model of these structures, which would provide crucial insight into the complex forces and stresses within the HSAs as well as how they behave under arbitrary loading conditions, have failed to produce reasonable results. These structures pose difficulties for analysis with FEM not only because of the highly nonlinear large deformations they undergo in the relevant use cases, but also because of the numerous fine connections and sharp geometric discontinuities inherent to their design. As shown in Figure 4.2, linear FEM analysis of these structures produces clearly erroneous results which violate basic symmetries of the design domain. Nonlinear FEM analysis does resolve these issues, but at a steep computational cost, as summarized in Table 4.2.

As has been reiterated numerous times throughout this work, peridynamics offers the unique advantage of readily accounting for both the existence of discontinuities and the stress concentrations they cause, as well as geometrically nonlinear deformations. As such, the final component of this section is the demonstration of this peridynamic framework in the simulation of a test set of HSA structures. Figure 4.2 depicts the peridynamic model of a sample HSA cylinder undergoing a radial twist that produces radial expansion and

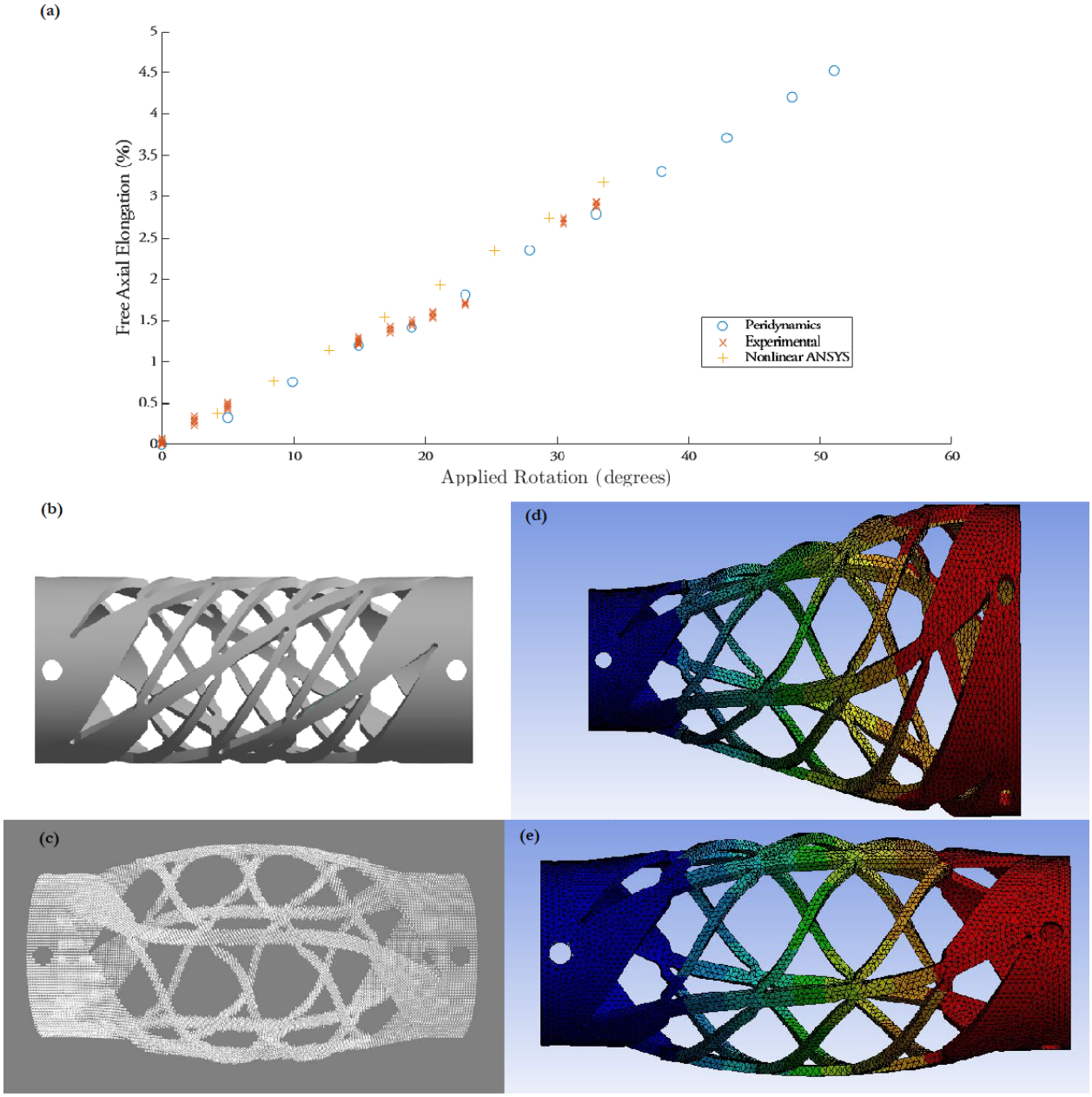

Figure 4.2: Simulation results for a selected HSA undergoing a 90-degree axial twist. Panel (a) shows a comparison of numerical results for the elongation of an HSA structure undergoing an applied twist while unconstrained in the axial direction. Experimental results are compared with simulation results of both ANSYS Nonlinear FEM and peridynamics. (b) depicts an undeformed HSA structure. (c) depicts the peridynamic simulation result for that structure with an applied twist. (d) shows the linear FEM results from ANSYS for the same structure, and (e) shows the nonlinear ANSYS results.

Solution Method	Linear ANSYS, CPU	Linear ANSYS, GPU	Nonlinear ANSYS, CPU	Nonlinear ANSYS, GPU	Peridynamics (GPU)
Computation Time (s)	10.5	10.4	1741	1563	1.25

Table 4.2: Computation times for the results depicted in Figure 4.2. CPU processor was an Intel Core i7, 1.86 GHz; GPU processor was a Nvidia RTX 3090.

axial extension, alongside the results obtained with ANSYS linear and nonlinear solution methods. Additionally, Figure 4.2 depicts the relationship between applied twist and the free axial extension of the structure, comparing the results of the peridynamic model with those from nonlinear FEM evaluation with ANSYS, as well as experimentally determine values. Finally, Table 4.2 summarizes the simulation speeds of the various solvers used in a 75000 node simulation. Note that, as there were no open-source GPU-based nonlinear FEM solvers available at the time this document was prepared, the best available comparison for the GPU peridynamic framework is the GPU-supported solution method available within the ANSYS software.

4.3 Topology Optimization

Topology optimization of structural designs is a broad field for which a wide variety of methodologies have been developed. When the concept of topology optimization was first introduced in the late 1980s, a homogenization-based method was proposed [66]. Since then, various other techniques have been demonstrated, such as genetic algorithms [67], solid isotropic material with penalization (SIMP) [68], level-set methods [69], and evolutionary structural optimization (ESO) [70]. Virtually all of these approaches incorporate iteratively performing structural analysis on an updating design space, with FEM being by far the most common tool used for that step. In principle, FEM can be replaced with peridynamics in any of these methods; however, careful analysis of the use of peridynamics in the entire collection of topology optimization methodologies is beyond the scope of this work. Instead,

the following section selects a version of the SIMP approach for thorough exploration based on its widespread use in large-scale engineering problems.

SIMP is a continuous density-based optimization approach in which the densities of all units of the discretized domain (elements in FEM; nodes in peridynamics) are treated as design variables, where the base material density of each node is multiplied by a value between 0 and 1 to produce a new topology. A number of variations of this method exist describing the specific heuristics by which the density is updated to minimize an objective function. For example, the optimality criterion (OC) method uses the gradient of the objective function with respect to the design variables to determine the direction in which the design variables are stepped [71]. Alternatively, the proportional optimization (PO) method assigns density values for a node based on its proportion of the objective function in the previous design iteration [72]. This work will focus on the PO method for its computational and conceptual simplicity, but the methods described here may be easily adapted to other variations of SIMP topology optimization.

Pioneering work demonstrating peridynamics for use in SIMP topology optimization has recently been performed by Sohouli et al. [33, 34]. In these works, the authors develop formulations of both OC and PO methods for peridynamics, and show that results produced for canonical 2D optimization problems match those obtained with traditional FEM-based methods. Moreover, Sohouli et al. demonstrate that peridynamics readily allows for topology optimization in problems dealing with cracks and discontinuities, noting that due to the cumbersome and bespoke nature of XFEM modelling, little exploration of optimization in such problems is available, therefore making a case for the use of peridynamics. However, these works do not explore the computational demands of peridynamic topology optimization, and prior to the GPU framework presented here, the numerous simulations needed for optimization incur a heavy expense. As such, the focus of this section is porting the proportional topology optimization process into this GPU peridynamics framework and evaluating the speed relative to state-of-the-art FEM methods. Additionally, beyond the crack problems evaluated by Sohouli et al., this optimization technique will be validated on nonlinear

optimization problems to further demonstrate the unique versatility peridynamics lends over FEM-based solutions.

4.3.1 Proportional Optimization Formulation for Peridynamics

The general principle of proportional topology is simple: with each optimization iteration, the material of the domain is distributed proportionally to the distribution of elastic strain energy density throughout the domain. Mathematically speaking, each material point $\hat{\mathbf{x}}_i$ has a corresponding design variable κ_i relating to the density of the point. Specifically, in each optimization iteration the stiffness used for each material point is determined as

$$E_i = (\kappa_i)^p E_s \tag{4.7}$$

where E_s is the Elastic Modulus of the material, and p is a penalization parameter, typically in the range $1 \leq p \leq 5$.

The most common optimality criteria used in structural engineering design is compliance, C, which can be defined in terms of the peridynamic elastic strain energy density given by Eq. 1.8:

$$C = \int_{\Omega} W dV_{\hat{\mathbf{x}}} = \sum_i W(\hat{\mathbf{x}}_i) V_i \tag{4.8}$$

Finally, a volume constraint is typically applied, limiting the total material used to a set fraction $\overline{V}$ of the total volume of the design space. The resulting minimization problem for the topology optimization can be summarized as

$$\min C(\kappa_i) = \sum_i W(\hat{\mathbf{x}}_i) V_i \text{ s.t. } \begin{cases} \frac{\sum_i \kappa_i V_i}{\sum_i V_i} = \overline{V} \\ \kappa_{min} \leq \kappa_i \leq 1 \end{cases} \tag{4.9}$$

Note the bounds on κ, particularly the lower bound κ_{min}, which is typically set to some small value slightly greater than 0 to ensure that the stiffness matrix of the peridynamic

problem never becomes singular.

In each iteration, after the peridynamic simulation is completed, a new value is computed for each design variable κ_i^{opt} proportional to each material point's contribution to the total compliance:

$$\kappa_i^{opt} = RM \frac{W(\hat{\mathbf{x}}_i)V_i}{C} \tag{4.10}$$

where RM is the remaining material to be distributed. To maintain stability in the overall topology modification, the design variable used in the subsequent iteration is blended with the previous value according to a history parameter, α:

$$\kappa_i^{n+1} = \alpha\kappa_i^{n-1} + (1-\alpha)\kappa_i^{opt} \tag{4.11}$$

Initially, RM is set to be the fraction $\overline{V}$ of the total design space volume. However, after the material is distributed, κ_i must be bounded by κ_{min} and 1. Additionally, the following filter is applied to smooth the result and eliminate the impractical fractal support structures that can easily arise in topology optimizers:

$$\psi(||\boldsymbol{\xi}_{ij}||) = \begin{cases} r_{min} - ||\boldsymbol{\xi}_{ij}|| & \text{for } ||\boldsymbol{\xi}_{ij}|| < r_{min} \\ 0 & \text{for } ||\boldsymbol{\xi}_{ij}|| \geq r_{min} \end{cases} \tag{4.12}$$

where r_{min} is the radius of some neighborhood around each material point (the peridynamic neighborhood $\mathcal{H}$ is a convenient choice). Finally, a new design variable is obtained:

$$\overline{\kappa_i} = \frac{\sum_j \psi(||\boldsymbol{\xi}_{ij}||)\kappa_j}{\sum_j \psi(||\boldsymbol{\xi}_{ij}||)} \tag{4.13}$$

After these limits and filter are applied, the actual material used will be less than the desired volume fraction $\overline{V}$. This remainder becomes RM, and Eqs. 4.10 - 4.13 are repeated until this remained is sufficiently small.

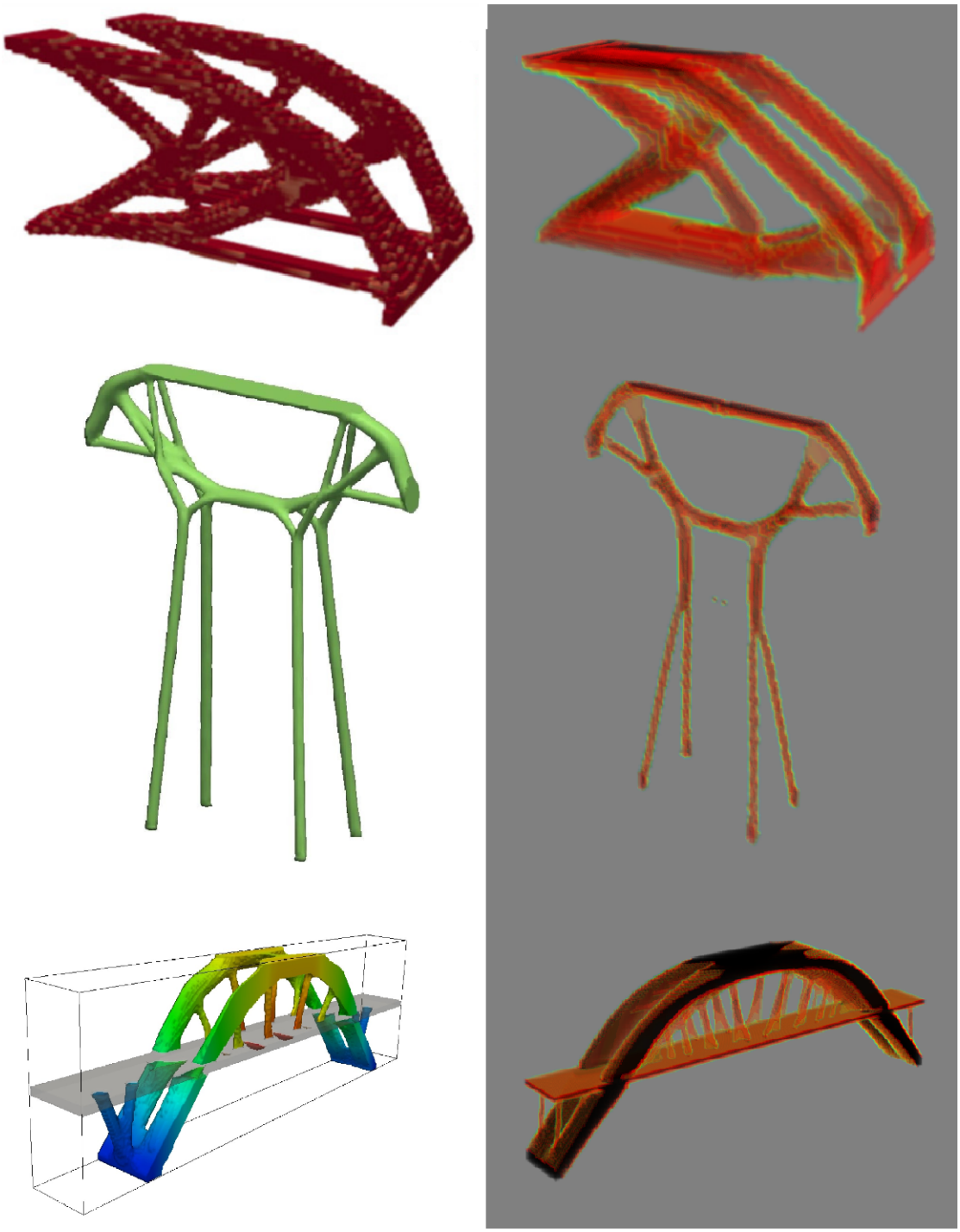

Figure 4.3: Topology optimization results for the cantilevered beam, electric mast, and tiered bridge optimization problems (from top to bottom). The left column shows the results of FEM-based solution from literature (Cantilevered beam courtesy of Ratnakar et al. [73]; electric mast and tiered bridge courtesy of Martinez-Frutos et al. [74]), and the right column shows the results of the peridynamic framework presented in this work.

4.3.2 Validation

Large-Scale Optimization

While the results for a collection of canonical 2D problems presented in [33] are important proofs of concept for peridynamic topology optimization, most real-world engineering problems deal with large, 3-dimensional structures. As such, demonstrating the scalability of peridynamic optimization to large-scale problems is crucial in realizing its widespread adoption. To that end, three numerical test problems are selected here for which 3D, high-resolution results are available in recent literature for state-of-the-art, parallelized FEM-based solvers. Specifically, [73] describes a 3D cantilevered beam optimization, and provides timing results for optimization with both a serial solver on the CPU, and a parallelized conjugate gradient solver on the GPU. Similarly, [74] describes the optimization of a 3D T-shaped design space with vertical loads acting on the crossbar (termed the "Electric Mast" problem), and a 3D rectangular domain experiencing a vertical distributed load through its midplane (termed the "Tiered Bridge" problem), and includes the serial and parallel timing results. Please see these references for a complete description of the problem geometries, materials, applied loads, and optimization settings.

Figure 4.3 depicts the results obtained with peridynamic structural analysis, in comparison with results from literature obtained with FEM-based approaches. Figure 4.4 contains the timing results for both serial and parallel optimizations reported in the corresponding literature, in comparison with those obtained with the peridynamic GPU framework. Note that the most direct comparison in Figure 4.4 is between the Nvidia Tesla k40 numerical experiments, as both FEM- and peridynamic-based methods have been evaluated on that hardware. Intel CPU results for FEM approaches and Nvidia RTX 3090 results for peridynamic approaches are included for further reference.

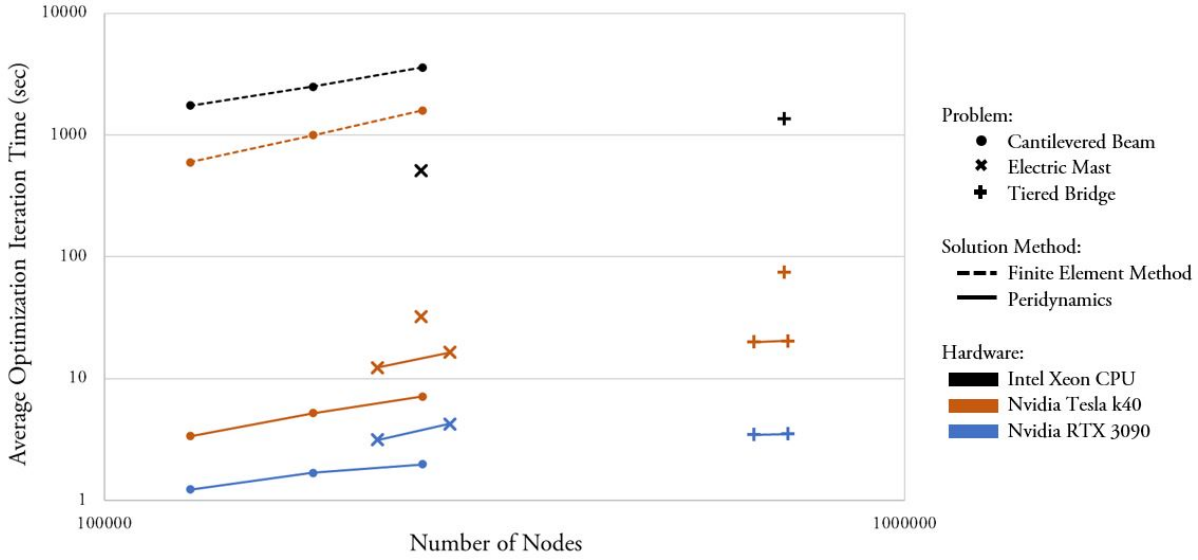

Figure 4.4: Average execution time for a single optimization iteration for a variety of approaches in large-scale topology optimization. Dashed lines and standalone points indicate state-of-the-art FEM-based results reported in literature; solid lines indicate peridynamics-based results original to this work. Round, X, and Plus markers indicate the cantilevered beam, electric mast, and tiered bridge problems, respectively. Color indicates the hardware each problem runs on, with black, red, and blue indicating Intel Xeon, Nvidia Tesla k40, and Nvidia RTX 3090, respectively.

Nonlinear Optimization

While the ability of the peridynamic topology optimizer to outperform state-of-the-art FEM-based topology optimizers for simple linear elastic test problems is an important advancement, peridynamic topology optimization offers even further advantage: the ability to optimize designs dealing with more complex physical phenomena. Sohouli et al. explore using peridynamics for the optimization of cracked structures [33] and multi-material design domains [34]; however, optimization of structures exhibiting nonlinear behavior is one use case which has yet to be explored. Given the groundwork laid in the study of peridynamics in nonlinear analysis described in Section 4.2, such a use case is a logical extension of this work.

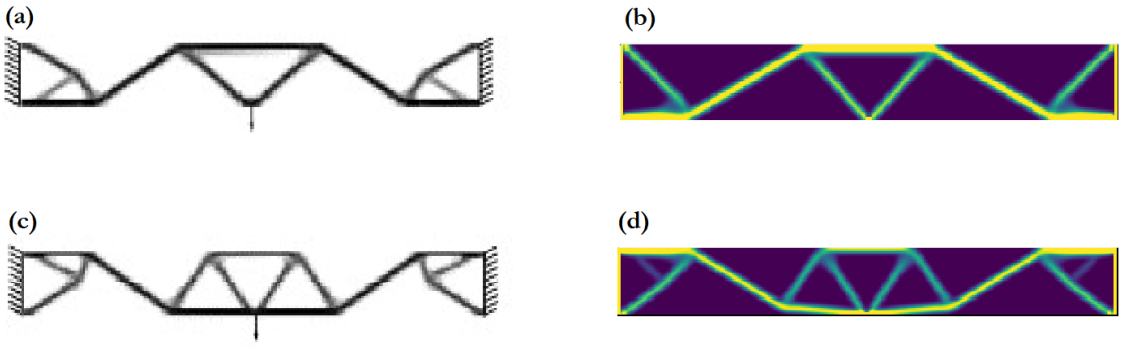

Figure 4.5: Small- and large-deformation results for a centrally-loaded beam. Panel (a) depicts optimization results with a standard (small-deformation) FEM solver. Panel (b) depicts results of the peridynamic optimizer with a small load applied. Panel (c) depicts results of a nonlinear FEM solver utilizing the Newton-Raphson method to evaluate large displacements. Panel (d) depicts results of the peridynamic solver with a large load applied. Note that no formulational modification is required to produce the bifurcation between panels (b) and (d). Images in panels (a) and (c) are courtesy of Jung & Gea [59].

A number of works published in the past 2 decades have studied the optimization of nonlinear structures, typically using nonlinear FEM techniques for the structural analysis stages, and almost exclusively restricting analysis to simple, 2D structures due to the large computational expense of iteratively performing such nonlinear FEM analysis [59]. However, despite the significant research interest in nonlinear optimization, little work is available discussing the computational aspects of the process, with only a small handful of programs for nonlinear optimization being publicly available [75]. To date, none of these programs make use of modern high-performance computing tools, and as such the state-of-the-art of nonlinear topology optimization lags behind that of simpler linear optimizers. As was explored in Section 4.2, peridynamics naturally reproduces elements of large-deformation behavior due to the reference frame agnosticism of bond forces, and can be readily extended to fully represent geometric nonlinearity through insertion of nonlinear bond-strain measures, which introduces little to no inherent additional computational cost. This fact, in combination

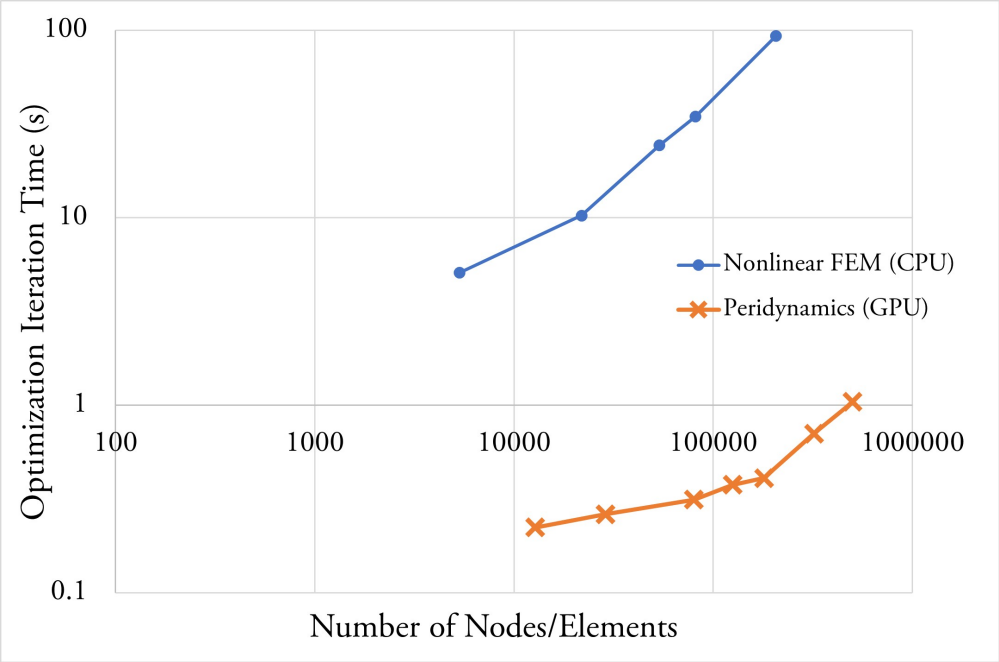

Figure 4.6: Comparison of the most recent publicly available nonlinear topology optimization tool, based on FEM (courtesy of Zhu et al. [75]), with nonlinear topology optimization performed with the GPU peridynamics framework. Note that the nonlinear FEM solutions are run on the CPU as there are no currently available nonlinear FEM-based optimizer utilizing the GPU.

with the GPU framework and peridynamic topology optimizer presented in this work, makes peridynamics an exceptional choice for the structural analysis step in nonlinear topology optimization.

Figure 4.5 shows a basic demonstration of the peridynamic topology optimizer in a problem in which large deformation is known to tangibly affect the optimizer topology. The problem consists of a low-stiffness elastic beam, fixed on the ends and loaded with a point force at the center of the bottom edge. Jung et al. [59] provide SIMP optimization results utilizing FEM-based structural analysis, both for a linear solution, and for large-deformation behavior evaluated with the Newton-Raphson method. Figure 4.6 depicts the timing results for an average optimization iteration of the most recently published FEM-based nonlinear

topology optimization code, based on the FreeFEM package and provided by Zhu et al. [75], alongside the corresponding timings of the peridynamic topology optimizer, as a function of problem size.

4.4 Discussion

In Chapter 3, the GPU framework for peridynamic simulation that is developed was shown to yield a substantial advantage in performance and representable problem size over the prior state-of-art peridynamics software, therefore comprising a substantial advancement for the field of peridynamics. Chapter 4 expands on the significance of this development by demonstrating that the framework yields an advantage over the state-of-the-art of not only peridynamics, but of structural analysis tools in general. Specifically, clear use cases for the GPU peridynamics framework are established in the domains of both nonlinear structural analysis and topology optimization.

Section 4.2.2 explores the application of the GPU peridynamic framework to general nonlinear large-deformation problems, and compares the results with the state-of-the-art in FEM-based nonlinear analysis. Note that modern GPU-based solvers for nonlinear FEM are not available in present literature, and the CPU-based results selected for comparison are the fastest FEM-based nonlinear solutions available at the time of writing. For the selected test problem of a T-shaped beam undergoing large-angle torsion, the peridynamic modeler outperforms the full-fidelity FEM solver by a factor of 465x, as given in Table 4.1. While the peridynamic solution remains an order of magnitude slower than the reduced-order nonlinear solver presented by [63], it is able to vastly narrow the gap between a full-fidelity model and the reduced-order model, while providing the generality yielded by maintaining a full geometric representation.

Section 4.2.3 focuses on a specific application of nonlinear structural analysis, handed-shearing auxetics. HSAs are an emergent technology in mechanical metamaterials which use complex geometric features to produce unusual structural behavior. Both large deformations and geometric discontinuities are often intrinsic to the designs of these structures, therefore

posing difficulty for typical FEM modelling methods. The GPU peridynamic framework is validated on these HSA structures, comparing against both experimentally obtained results and simulation results produced with ANSYS, a leading modern software package for FEM analysis. As is shown in Figure 4.2, linear FEM analysis of HSAs produces erroneous results; for a cylindrical structure experiencing a torsional twist, the resulting radial deformation should be symmetric in the axial direction, but the FEM produces an exponentially increasing radial deformation. Nonlinear analysis with ANSYS resolves this issue and provides a reasonable fit for the experimentally determined data, but demands a dramatic increase in computational load, even when ANSYS's GPU accelerator is utilized, as summarized in Table 4.2. Peridynamic simulation, on the other hand, automatically produces reasonable behavior, which also matches experimental data for HSA deformation more closely than nonlinear FEM. Moreover, the GPU framework is able to achieve this result 1250x faster than the GPU-accelerated solver available in the ANSYS software package.

Finally, Section 4.3 explores the advantages yielded by the GPU peridynamic framework in both linear and nonlinear topology optimization. Figure 4.3 shows the reproduction of large-scale topology optimization problems with the peridynamics solver, showing very good agreement with FEM-based results available in literature. The minor differences that do exist are likely the result of variations in the specifics of the implementations of the SIMP method used, as well as slight differences in optimization parameters not explicitly stated in literature.

In linear structural analysis, the GPU framework brings peridynamics to approximate computational parity with state-of-the-art GPU-parallelized FEM solvers for a single structural evaluation of a body. For topology optimization however, the use of peridynamics yields a further computational advantage: the semi-dynamic nature of the peridynamic formulation means that if a simulation is provided with a reasonable estimate of the solution field, the time required to obtain the true solution is substantially reduced. Therefore, in applications like topology optimization where numerous analysis iterations of similar topologies are performed, the total runtime with peridynamics is significantly reduced by seeding

a given analysis with the results of the previous iteration. The benefits yielded by this approach are summarized in Figure 4.4, which show the peridynamic solver yielding a speed-up of anywhere from 2.2 to 223x over state-of-the-art FEM solvers running on the same GPU hardware.

Along with a computational advantage, peridynamics offers greater versatility in the class of physical phenomena it can represent over basic FEM. Peridynamic topology optimization involving certain examples of such phenomena, such as cracked bodies and multimaterial domains, have been recently explored by other researchers. Section 4.3.2 expands on these works by applying the peridynamic topology optimizer to nonlinear deformation problems. Figure 4.5 shows an example of a slender beam undergoing bending, which produces a significantly different optimized design from linear analysis when large deformations are considered. In this example, peridynamics naturally produces this design bifurcation as the applied load transitions from the linear to nonlinear regime. This demonstration presents a significant advancement of the state-of-the-art of nonlinear topology optimization; as depicted in Figure 4.6, the GPU peridynamic solver yields an acceleration of two orders of magnitude over available nonlinear FEM-based optimizers. This advancement greatly increases the tractable problem size manageable in nonlinear topology optimization.

BIBLIOGRAPHY

1. Rudnicki, J. W. *Fundamentals of Continuum Mechanics* (John Wiley & Sons, 2015).
2. Hrennikoff, A. Solution of Problems of Elasticity by the Framework Method. *Journal of Applied Mechanics* **8,** A169–A175. ISSN: 0021-8936. eprint: https://asmedigitalcollection.asme.org/appliedmechanics/article-pdf/8/4/A169/6744200/a169_1.pdf. https://doi.org/10.1115/1.4009129 (Mar. 2021).
3. Groetsch, C. in *Encyclopedia of Physical Science and Technology* (ed Meyers, R. A.) 3rd, 337–353 (Academic Press, New York, 2003). ISBN: 978-0-12-227410-7.
4. Silling, S. Reformulation of elasticity theory for discontinuities and long-range forces. *Journal of the Mechanics and Physics of Solids* **48,** 175–209 (2000).
5. Silling, S. A., Weckner, O., Askari, E. & Bobaru, F. Crack nucleation in a peridynamic solid. *IUTAM Symposium on Dynamic Fracture and Fragmentation,* 219–227 (2010).
6. Kilic, B. & Madenci, E. *Prediction of crack paths in a quenched glass plate by using peridynamic theory* May 2009. https://link.springer.com/article/10.1007/s10704-009-9355-2.
7. Agwai, A., Guven, I. & Madenci, E. Predicting crack propagation with peridynamics: A comparative study. *International Journal of Fracture* **171,** 65–78 (2011).
8. Sau, N., Medina-Mendoza, J. & Borbon-Almada, A. C. Peridynamic modelling of reinforced concrete structures. *Engineering Failure Analysis* **103,** 266–274. ISSN: 1350-6307. https://www.sciencedirect.com/science/article/pii/S1350630718313530 (2019).
9. Datta, D. Introduction to eXtended Finite Element (XFEM) Method (Aug. 2013).

10. Madenci, E. & Oterkus, E. *Peridynamic Theory and its Applications* (Springer-Verlag, 2016).

11. Foster, J. T., Silling, S. A. & Chen, W. W. Viscoplasticity using peridynamics. *International Journal for Numerical Methods in Engineering* **81,** 1242–1258. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2725`. `https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2725`.

12. Madenci, E. & Oterkus, E. Fully coupled peridynamic thermomechanics. *Peridynamic Theory and Its Applications,* 245–278 (2013).

13. Bobaru, F. & Duangpanya, M. The peridynamic formulation for transient heat conduction. *International Journal of Heat and Mass Transfer* **53,** 4047–4059 (2010).

14. Nishawala, V. V., Ostoja-Starzewski, M., Leamy, M. J. & Demmie, P. N. Simulation of elastic wave propagation using cellular automata and peridynamics, and comparison with experiments. *Wave Motion* **60,** 73–83 (2016).

15. Bažant, Z. P., Luo, W., Chau, V. T. & Bessa, M. A. Wave dispersion and basic concepts of peridynamics compared to classical nonlocal damage models. *Journal of Applied Mechanics* **83** (2016).

16. Tong, Q. & Li, S. Multiscale coupling of molecular dynamics and peridynamics. *Journal of the Mechanics and Physics of Solids* **95,** 169–187 (2016).

17. Nguyen, C. T. & Oterkus, S. Ordinary state-based peridynamic model for geometrically nonlinear analysis. *Engineering Fracture Mechanics* **224,** 106750 (2020).

18. Tupek, M. & Radovitzky, R. An extended constitutive correspondence formulation of peridynamics based on nonlinear bond-strain measures. *Journal of the Mechanics and Physics of Solids* **65,** 82–92 (2014).

19. Wang, B., Oterkus, S. & Oterkus, E. Determination of horizon size in state-based peridynamics. *Continuum Mechanics and Thermodynamics* (2020).

20. Ostoja-Starzewski, M., Demmie, P. & Zubelewicz, A. On Thermodynamic Restrictions in Peridynamics. *Journal of Applied Mechanics* **80,** 4502– (Jan. 2013).

21. Silling, S., Epton, M., Weckner, O., Xu, J. & Askari, E. Peridynamic States and Constitutive Modeling. eng. *Journal of Elasticity* **88,** 151–184. ISSN: 0374-3535 (2007).

22. Mitchell, J. A. & Silling, S. A. *Towards Addressing Surface Effects in Ordinary Isotropic Peridynamic Models Position Aware Linear Solid (PALS)* eng. in (United States, 2013).

23. Le, Q. V. & Bobaru, F. Surface corrections for peridynamic models in elasticity and fracture. *Computational Mechanics* **61,** 499–518 (2018).

24. Bobaru, F. *Handbook of peridynamic modeling* (CRC Press, 2017).

25. Madenci, E. & Oterkus, E. Coupling of the peridynamic theory and finite element method. *Peridynamic Theory and Its Applications,* 191–202 (2013).

26. Oterkus, E. *Peridynamic theory for modeling three-dimensional damage growth in metallic and composite structures* in (2010).

27. Macek, R. W. & Silling, S. A. Peridynamics via finite element analysis. *Finite Elements in Analysis and Design* **43,** 1169–1178 (2007).

28. Kilic, B. & Madenci, E. An adaptive dynamic relaxation method for quasi-static simulations using the peridynamic theory. *Theoretical and Applied Fracture Mechanics* **53,** 194–204. ISSN: 0167-8442. https://www.sciencedirect.com/science/article/pii/S0167844210000479 (2010).

29. Gerstle, W. *Introduction To Practical Peridynamics: Computational Solid Mechanics Without Stress And Strain* ISBN: 9789814699563. https://books.google.com/books?id=WDc8DQAAQBAJ (World Scientific Publishing Company, 2015).

30. Diehl, P. & Schweitzer, M. A. Efficient Particle-Based Simulation of Dynamic Cracks and Fractures in Ceramic Material (2014).

31. Ebrahimi, S. *Mechanical Behavior of Materials at Multiscale Peridynamic Theory and Learning-based Approaches* PhD thesis (2020).

32. Kaiser, H. *et al.* Hpx-the c++ standard library for parallelism and concurrency. *Journal of Open Source Software* **5,** 2352 (2020).

33. Kefal, A., Sohouli, A., Oterkus, E., Yildiz, M. & Suleman, A. Topology optimization of cracked structures using peridynamics. *Continuum Mechanics and Thermodynamics* **31,** 1645–1672 (2019).

34. Habibian, A. *et al.* Multi-material topology optimization of structures with discontinuities using Peridynamics. *Composite Structures* **258,** 113345 (2021).

35. Mikata, Y. Peridynamics for Fluid Mechanics and acoustics. *Acta Mechanica* **232,** 3011–3032 (2021).

36. Silling, S. A. Linearized theory of peridynamic states. *Journal of Elasticity* **99,** 85–111 (2010).

37. Courant, R., Friedrichs, K. & Lewy, H. Uber Die Partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen* **100,** 32–74 (1928).

38. Oñate, E., Perazzo, F. & Miquel, J. A finite point method for elasticity problems. *Computers & Structures* **79,** 2151–2163. ISSN: 0045-7949 (2001).

39. Hu, W., Ha, Y., Bobaru, F. & Silling, S. The formulation and computation of the nonlocal J-integral in bond-based peridynamics. eng. *International Journal of Fracture* **176,** 195–206. ISSN: 0376-9429 (2012).

40. Stenstrom, C. & Eriksson, K. The J-contour integral in peridynamics via displacements. *International Journal of Fracture* **216,** 173–183 (2019).

41. Diehl, P., Prudhomme, S. & Lévesque, M. A Review of Benchmark Experiments for the Validation of Peridynamics Models. *Journal of Peridynamics and Nonlocal Modeling* **1,** 14–35 (2019).

42. Littlewood, D. Roadmap for Peridynamic Software Implementation. *Sandia Report* (2015).

43. Parks, M. *et al. Peridigm* `https://github.com/peridigm/peridigm`. 2012.

44. Birkey, J. *Development of Visual EMU, a graphical user interface for the peridynamic EMU code* PhD thesis (Kansas State University, 2007).

45. Bartlett, J. D. & Storti, D. A SINGLE-CARD GPU IMPLEMENTATION OF PERIDYNAMICS. *Proceedings of the ASME 2021 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference* (Aug. 2021).

46. Li, X., Ye, H. & Zhang, J. *Redesigning Peridigm on SIMT Accelerators for High-performance Peridynamics Simulations* in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), 433–443.

47. *Cuda C++ Programming Guide* `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

48. Harris, M. *et al.* Optimizing parallel reduction in CUDA. *Nvidia developer technology* **2,** 70 (2007).

49. Bartlett, J. & Storti, D. A Generalized Fictitious Node Approach for Surface Effect Correction in Peridynamic Simulation. *Journal of Peridynamics and Nonlocal Modeling* (2021).

50. Bailey, T. An Introduction to the C Programming Language and Software Design. *July-2005* (2005).

51. Boys, B., Dodwell, T., Hobbs, M. & Girolami, M. PeriPy - A high performance OpenCL peridynamics package. *Computer Methods in Applied Mechanics and Engineering* **386,** 114085. ISSN: 0045-7825. `https://www.sciencedirect.com/science/article/pii/S0045782521004163` (2021).

52. Mihai, L. A. & Goriely, A. How to characterize a nonlinear elastic material? A review on nonlinear constitutive parameters in isotropic finite elasticity. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* **473,** 20170607 (2017).

53. Hibbitt, H., Marcal, P. & Rice, J. A finite element formulation for problems of large strain and large displacement. *International Journal of Solids and Structures* **6,** 1069–1086 (1970).

54. Groetsch, C. in *Encyclopedia of Physical Science and Technology* (ed Meyers, R. A.) 3rd, 337–353 (Academic Press, New York, 2003). ISBN: 978-0-12-227410-7.

55. Weckner, O. & Abeyaratne, R. The effect of long-range forces on the dynamics of a bar. *Journal of the Mechanics and Physics of Solids* **53,** 705–728. ISSN: 0022-5096. https://www.sciencedirect.com/science/article/pii/S0022509604001449 (2005).

56. Xu, L., He, X., Chen, W., Li, S. & Wang, G. Reformulating Hyperelastic Materials with Peridynamic Modeling. *Computer Graphics Forum* **37,** 121–130 (Oct. 2018).

57. Eschenauer, H. A. & Olhoff, N. Topology optimization of continuum structures: A review*. *Applied Mechanics Reviews* **54,** 331–390. ISSN: 0003-6900. eprint: https://asmedigitalcollection.asme.org/appliedmechanicsreviews/article-pdf/54/4/331/5438226/331_1.pdf. https://doi.org/10.1115/1.1388075 (July 2001).

58. ZHU, J. *et al.* A review of topology optimization for additive manufacturing: Status and challenges. *Chinese Journal of Aeronautics* **34,** 91–110. ISSN: 1000-9361. https://www.sciencedirect.com/science/article/pii/S1000936120304520 (2021).

59. Jung, D. & Gea, H. C. Topology optimization of nonlinear structures. *Finite Elements in Analysis and Design* **40,** 1417–1427. ISSN: 0168-874X. https://www.sciencedirect.com/science/article/pii/S0168874X03002117 (2004).

60. Maksum, Y. *et al.* Computational Acceleration of Topology Optimization Using Parallel Computing and Machine Learning Methods – Analysis of Research Trends. *Journal of Industrial Information Integration* **28,** 100352. ISSN: 2452-414X. https://www.sciencedirect.com/science/article/pii/S2452414X22000231 (2022).

61. Mukherjee, S. *et al.* Accelerating large-scale topology optimization: state-of-the-art and challenges. *Archives of Computational Methods in Engineering* **28,** 4549–4571 (2021).

62. Marinkovic, D. & Zehn, M. Survey of Finite Element Method-Based Real-Time Simulations. *Applied Sciences* **9.** ISSN: 2076-3417. https://www.mdpi.com/2076-3417/9/14/2775 (2019).

63. Maksum, Y. *et al.* Computational Acceleration of Topology Optimization Using Parallel Computing and Machine Learning Methods – Analysis of Research Trends. *Journal of Industrial Information Integration* **28,** 100352. ISSN: 2452-414X. https://www.sciencedirect.com/science/article/pii/S2452414X22000231 (2022).

64. Chin, L. *et al. A Simple Electric Soft Robotic Gripper with High-Deformation Haptic Feedback* in *2019 International Conference on Robotics and Automation (ICRA)* (2019), 2765–2771.

65. Garg, A., Good, I., Revier, D., Airis, K. & Lipton, J. I. Kinematic Modeling of Handed Shearing Auxetics via Piecewise Constant Curvature. *CoRR* **abs/2112.04706.** arXiv: 2112.04706. https://arxiv.org/abs/2112.04706 (2021).

66. Bendsøe, M. P. & Kikuchi, N. Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering* **71,** 197–224. ISSN: 0045-7825. https://www.sciencedirect.com/science/article/pii/0045782588900862 (1988).

67. Jenkins, W. Towards structural optimization via the genetic algorithm. *Computers & Structures* **40.** Special Issue: Computational Structures Technology, 1321–1327. ISSN: 0045-7949. https://www.sciencedirect.com/science/article/pii/0045794991904028 (1991).

68. Rozvany, G. I., Zhou, M. & Birker, T. Generalized shape optimization without homogenization. *Structural Optimization* **4,** 250–252 (1992).

69. Sethian, J. & Wiegmann, A. Structural Boundary Design via Level Set and Immersed Interface Methods. *Journal of Computational Physics* **163,** 489–528. ISSN: 0021-9991.

https://www.sciencedirect.com/science/article/pii/S0021999100965811 (2000).

70. Chu, D., Xie, Y., Hira, A. & Steven, G. Evolutionary structural optimization for problems with stiffness constraints. *Finite Elements in Analysis and Design* **21,** 239–251. ISSN: 0168-874X. https://www.sciencedirect.com/science/article/pii/0168874X9500043S (1996).

71. Sigmund, O. A 99 line topology optimization code written in MATLAB. *Structural and Multidisciplinary Optimization* **21,** 120–127 (2001).

72. Biyikli, E. & To, A. C. Proportional Topology Optimization: A New Non-Sensitivity Method for Solving Stress Constrained and Minimum Compliance Problems and Its Implementation in MATLAB. *Plos One* **10** (2015).

73. Ratnakar, S. K., Sanfui, S. & Sharma, D. *SIMP-Based Structural Topology Optimization Using Unstructured Mesh on GPU* in *Advances in Interdisciplinary Engineering* (eds Kumar, N., Tibor, S., Sindhwani, R., Lee, J. & Srivastava, P.) (Springer Singapore, Singapore, 2021), 1–10.

74. Martínez-Frutos, J., Martínez-Castejón, P. J. & Herrero-Pérez, D. Efficient topology optimization using GPU computing with multilevel granularity. *Advances in Engineering Software* **106,** 47–62. ISSN: 0965-9978. https://www.sciencedirect.com/science/article/pii/S0965997816302332 (2017).

75. Zhu, B. *et al.* An 89-line code for geometrically nonlinear topology optimization written in freefem. *Structural and Multidisciplinary Optimization* **63,** 1015–1027 (2020).

Appendix A
CRACK GROWTH VALIDATION

In addition to the novel applications of peridynamics explored in Chapter 4, it is important to validate the GPU software developed in this work on the other problem classes for which peridynamics is commonly used. In their textbook *Peridynamic Theory and its Applications*, Madenci & Oterkus provide a number of example problems demonstrating peridynamics in a broad set of use cases [10]. Of these test problems, a quintessential crack growth study is selected for comparison to demonstrate that the GPU peridynamics software is capable not only of recreating FEM solutions, but also of reproducing the results of existing peridynamic studies for cases in which the FEM cannot be readily applied. In this selected problem, a thin square plate is initialized with a horizontal crack in its midplane, and the horizontal edges of the plate are subjected to rapid velocity boundary conditions, stretching the plate vertically.

Specifically, the 50 x 50 mm plate contains a 10 mm central crack. The material is modelled using 2-D bond-based peridynamics with a Young's modulus of 192 GPa, Poisson's ratio of 1/3, and density of 8000 kg/m^3. A micro-brittle damage model is used, with a critical bond stretch of 0.04472. The domain is discretized into 500x500 node, and a horizon of 3x the grid spacing is used. A velocity-Verlet integrator, as described in Section 1.2.3, is employed to capture the dynamic crack growth, using a timestep of 1.3367e-8 seconds. Figure A.1 depicts the results for both a 20 m/s and 50 m/s applied velocity. The GPU software shows excellent agreement with the results reproduced from Madenci & Oterkus for both loading conditions. Note that the bifurcation between self-similar and branching fracture patterns in the two loading conditions is a result unique to peridynamics; alternative crack propagation modelling methods like XFEM produce self-similar crack growth as is seen

with the 20 m/s loading, but fail to naturally produce the complex fracture patterns shown in the 50 m/s loading without applying specially tailored criteria to trigger branching [4]. Running the compiled Fortran program provided by Madenci & Oterkus on an Intel i7 CPU takes 7,388 seconds. The GPU software runs the same problem on a Nvidia RTX 3090 in 0.3 seconds, yielding a remarkable speedup of 24,000x.

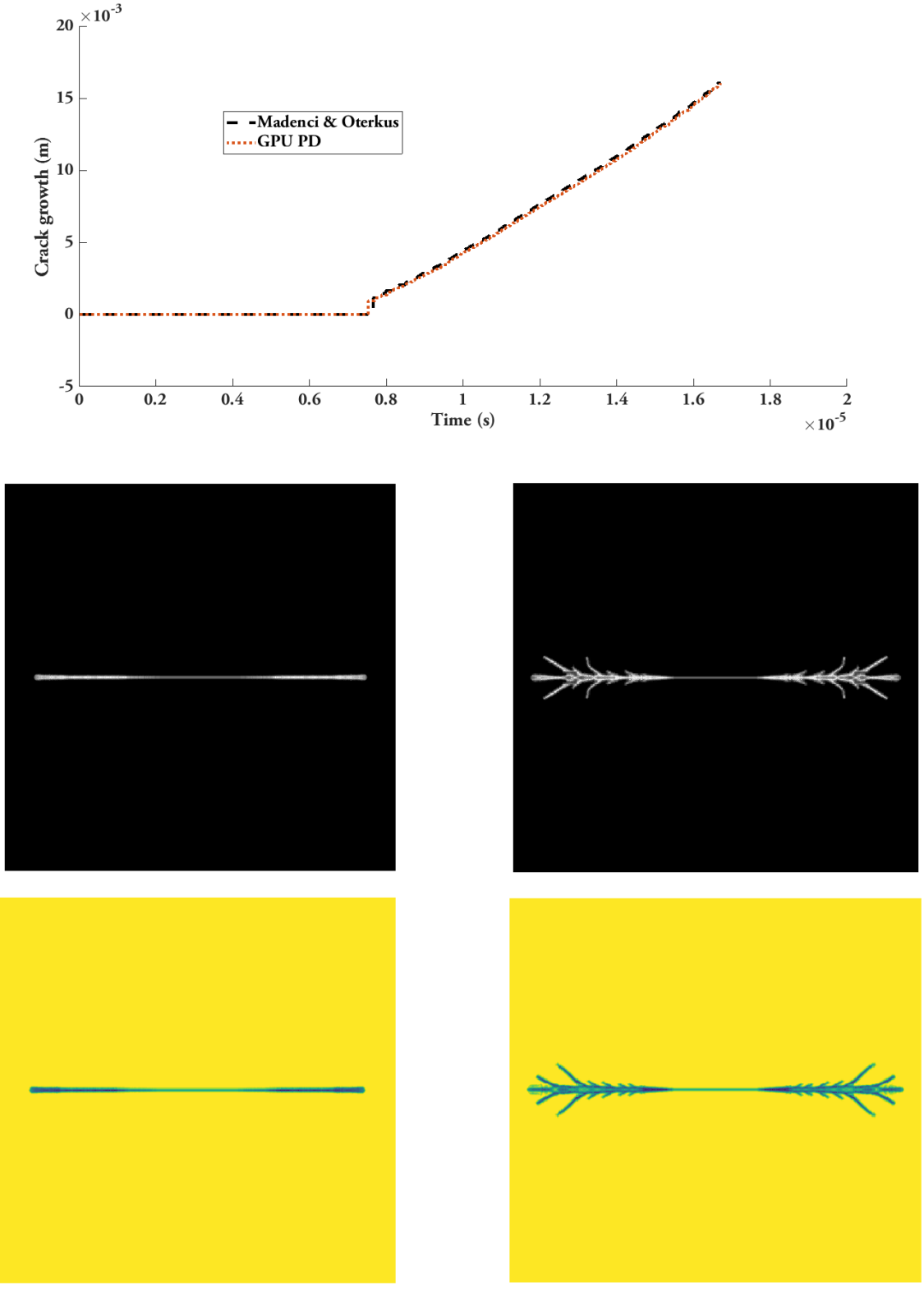

Figure A.1: Results for a thin plate with a pre-existing crack under velocity boundary conditions, compared with results reproduced from Madenci & Oterkus [10]. Panel (a) shows crack growth as a function of time for a 20 m/s applied velocity. Panels (b) and (c) show the fracture pattern produced by Madenci & Oterkus for 20 m/s and 50 m/s respectively. Pandels (d) and (e) show the corresponding results produced by the GPU software developed in this work.

Appendix B

CODE

B.1 Repository Links

A complete code repository for all the original software developed in this work can be found at GitHub:jd-bartlett96/peridynamics. The PeriPy software used as a reference can be found at GitHub:alan-turing-institute/PeriPy. The extension of PeriPy to state-based peridynamics developed in this work can be found at GitHub:jd-bartlett96/PeriPy. The following sections of this appendix include a collection of the final optimized software versions from the complete repository required to run a minimal example of state-based peridynamics simulation with topology optimization.

This code requires the following Python packages:

- `numpy`
- `pycuda`
- `time`
- `matplotlib`
- `mayavi`

B.2 GPU Kernels (CUDA)

```
#include <math.h>
#include <inttypes.h>

__device__ __constant__ float L = Lr;
__device__ __constant__ int NX = NXr;
__device__ __constant__ int NY = NYr;
__device__ __constant__ int NZ = NZr;
__device__ __constant__ int NB = NBr;
__device__ __constant__ int NN = NNr;
__device__ __constant__ double LOs[];
__device__ __constant__ int jadd[];
__device__ __constant__ double dt = dtr;
__device__ __constant__ double ntau = ntaur;
__device__ __constant__ double rho = rhor;
__device__ __constant__ double ecrit = ecritr;
__device__ __constant__ double bc = bcr;
__device__ __constant__ double kappa = kappar;
__device__ __constant__ double am = amr;
__device__ __constant__ double dV = dVr;
__device__ __constant__ double mass = massr;
__device__ __constant__ double MLT = MLTr;
__device__ __constant__ float xh = xhr;
__device__ __constant__ float xl = xlr;
__device__ __constant__ float yh = yhr;
__device__ __constant__ float yl = ylr;
__device__ __constant__ float zh = zhr;
__device__ __constant__ float zl = zlr;
__device__ __constant__ int penal = penalr;
__device__ __constant__ double alpha = alphar;
__device__ __constant__ double hrad = hradr;
__device__ __constant__ float HLF = 0.5;
__device__ __constant__ float TRE = 3;
__device__ __constant__ double ZER = 0;
__device__ __constant__ double ONE = 1;
__device__ __constant__ double TWO = 2;
__device__ int tt = 0;
__device__ __shared__ float sh_Sf[SHr]; // 4*NB + 1
```

```
__device__ bool TestBbox(float x, float y, float z){
    return (x>=xl) && (x<=xh) && (y>=yl) && (y<=yh) && (z>=zl) && (z<=zh);
}

__device__ bool TestBit(bool A[],  int64_t k ){
    return ( (A[k/8] & (1 << (k%8) )) != 0 ) ;
}

__device__ void  SetBit(bool A[],  int64_t k ){
    A[k/8] |= 1 << (k%8);
}

__global__ void calcVolume(float *d_Sf, double *d_m, bool *d_dmg, bool *d_chi){
    int tix = threadIdx.x;
    int iind = blockIdx.x * blockDim.x + tix;

    int k = iind/(NX*NY);
    int j = iind%(NX*NY)/NX;
    int i = iind%NX;

    if(tix<NB){
        sh_Sf[tix] = d_Sf[tix];
        sh_Sf[NB+tix] = d_Sf[NB+tix];
        sh_Sf[2*NB+tix] = d_Sf[2*NB+tix];
    }
    __syncthreads();

    float xi = L*(i+HLF) + xl;
    float yi = L*(j+HLF) + yl;
    float zi = L*(k+HLF) + zl;

    if (TestBbox(xi,yi,zi) && d_chi[iind]) {
        double mi = ZER;
        for (int64_t b = 0;b<NB;b++){
            float dx2 = sh_Sf[b];
            float dy2 = sh_Sf[NB+b];
            float dz2 = sh_Sf[2*NB+b];
            if (TestBbox(xi+dx2, yi+dy2, zi+dz2) && !TestBit(d_dmg, b*NN + iind)) {
                int jind = iind + jadd[b];
                if(d_chi[jind]){
                    double L0 = sqrt(dx2*dx2 + dy2*dy2 + dz2*dz2);
```

```
                    mi += L0*L*L*L;
                }
            }
        }
        d_m[iind] = mi;
    }
}

__global__ void calcDilation(float *d_Sf, double *d_u, double *d_dil,
                             bool *d_dmg, double *d_m, bool *d_chi){
    int tix = threadIdx.x;
    int iind = blockIdx.x * blockDim.x + tix;

    int k = iind/(NX*NY);
    int j = iind%(NX*NY)/NX;
    int i = iind%NX;

    if(iind==0){
        tt += 1;
    }

    if(tix<NB){
        sh_Sf[tix] = d_Sf[tix];
        sh_Sf[NB+tix] = d_Sf[NB+tix];
        sh_Sf[2*NB+tix] = d_Sf[2*NB+tix];
    }
    __syncthreads();

    float xi = L*(i+HLF) + xl;
    float yi = L*(j+HLF) + yl;
    float zi = L*(k+HLF) + zl;

    if (TestBbox(xi,yi,zi) && d_chi[iind]) {
        double ui = d_u[iind];
        double vi = d_u[NN+iind];
        double wi = d_u[2*NN+iind];

        double mi = d_m[iind];

        double dil = ZER;
        for (int64_t b = 0;b<NB;b++){
```

```
            float dx2 = sh_Sf[b];
            float dy2 = sh_Sf[NB+b];
            float dz2 = sh_Sf[2*NB+b];
            if (TestBbox(xi+dx2, yi+dy2, zi+dz2) && !TestBit(d_dmg, b*NN + iind)) {
                int jind = iind + jadd[b];
                if(d_chi[jind]){
                    double uj = d_u[jind];
                    double vj = d_u[NN+jind];
                    double wj = d_u[2*NN+jind];
                    double L0 = sqrt(dx2*dx2 + dy2*dy2 + dz2*dz2);

                    double A = dx2+uj-ui;
                    double B = dy2+vj-vi;
                    double C = dz2+wj-wi;
                    double LN = sqrt(A*A + B*B + C*C);
                    double eij = LN - L0;
                    if (eij/L0 > ecrit){
                        SetBit(d_dmg, b*NN + iind);
                        printf("Bond broken");
                    }else{
                        dil += eij*L*L*L;
                    }
                }
            }
        }
        d_dil[iind] = dil*3/mi;
    }
}

__global__ void calcForce(float *d_Sf, double *d_u, double *d_dil, double *d_m,
                          bool *d_dmg, double *d_F, double *d_vh, double *d_cd,
                          double *d_cn, int *d_EBCi, double *d_k, double *d_W,
                          double *d_Ft, bool *d_chi) {

    int tix = threadIdx.x;
    int iind = blockIdx.x * blockDim.x + tix;
    int k = iind/(NX*NY);
    int j = iind%(NX*NY)/NX;
    int i = iind%NX;
    if(tix<NB){
        sh_Sf[tix] = d_Sf[tix];
```

```
    sh_Sf[NB+tix] = d_Sf[NB+tix];
    sh_Sf[2*NB+tix] = d_Sf[2*NB+tix];
}
__syncthreads();
float xi = L*(i+HLF) + xl;
float yi = L*(j+HLF) + yl;
float zi = L*(k+HLF) + zl;
if (TestBbox(xi,yi,zi) && d_chi[iind]) {
    double ui = d_u[iind];
    double vi = d_u[NN+iind];
    double wi = d_u[2*NN+iind];
    double ki = pow(d_k[iind],penal);
    double mi = d_m[iind];
    double dili = d_dil[iind];
    double fx = ZER;
    double fy = ZER;
    double fz = ZER;
    double Wsm = 0;
    for (int64_t b = 0;b<NB;b++){
        float dx2 = sh_Sf[b];
        float dy2 = sh_Sf[NB+b];
        float dz2 = sh_Sf[2*NB+b];
        if (TestBbox(xi+dx2, yi+dy2, zi+dz2) && !TestBit(d_dmg, b*NN + iind)) {
            int jind = iind + jadd[b];
            if(d_chi[jind]){
                double uj = d_u[jind];
                double vj = d_u[NN+jind];
                double wj = d_u[2*NN+jind];
                double L0 = sqrt(dx2*dx2 + dy2*dy2 + dz2*dz2);//L0s[b];
                double A = dx2+uj-ui;
                double B = dy2+vj-vi;
                double C = dz2+wj-wi;
                double LN = sqrt(A*A + B*B + C*C);
                double eij = LN - L0;
                double kj = pow(d_k[jind],penal);
                double mj = d_m[jind];
                double dilj = d_dil[jind];
                double tij = ki*(3*kappa*dili + am*(eij/L0 - dili/3))/mi;
                double tji = kj*(3*kappa*dilj + am*(eij/L0 - dilj/3))/mj;
                double fsm = (tij + tji)*L*L*L;
                fx += fsm*A/LN;
```

```
                fy += fsm*B/LN;
                fz += fsm*C/LN;
                Wsm += 2*(ki*kj)/(ki + kj)*0.5*0.5*bc*eij*eij/L0*dV;
            }
        }
    }
    int ebcx = d_EBCi[iind];
    int ebcy = d_EBCi[NN + iind];
    int ebcz = d_EBCi[2*NN + iind];
    double vhx = d_vh[iind];
    double vhy = d_vh[NN + iind];
    double vhz = d_vh[2*NN + iind];
    double pfx = d_F[iind];
    double pfy = d_F[NN + iind];
    double pfz = d_F[2*NN + iind];
    double cn = ZER;
    double cd = ZER;
    if(ebcx<0 && vhx != ZER){
        cn -= MLT*ui*ui*(fx - pfx)/(ki*mass*dt*vhx);
    }
    cd += ui*ui;
    if(ebcy<0 && vhy != ZER){
        cn -= MLT*vi*vi*(fy - pfy)/(ki*mass*dt*vhy);
    }
    cd += vi*vi;
    if(ebcz<0 && vhz != ZER){
        cn -= MLT*wi*wi*(fz - pfz)/(ki*mass*dt*vhz);
    }
    cd += wi*wi;
    d_cn[iind] = cn;
    d_cd[iind] = cd;
    d_F[iind] = fx;
    d_F[NN + iind] = fy;
    d_F[2*NN + iind] = fz;
    if(ebcx<0 && ebcy<0 && ebcz<0){
        d_Ft[iind] = sqrt(fx*fx + fy*fy + fz*fz);
        d_W[iind] = Wsm;
    }
  }
}
```

```
__global__ void calcDisplacement(double *d_c, double *d_u, double *d_vh,
                                 double *d_F, int *d_NBCi, float *d_NBC,
                                 int *d_EBCi, float *d_EBC, double *d_k,
                                 bool *d_chi){
    int tix = threadIdx.x;
    int iind = blockIdx.x * blockDim.x + tix;
    int k = iind/(NX*NY);
    int j = iind%(NX*NY)/NX;
    int i = iind%NX;
    float xi = L*(i+HLF) + xl;
    float yi = L*(j+HLF) + yl;
    float zi = L*(k+HLF) + zl;
    double c = d_c[0];
    if (TestBbox(xi,yi,zi) && d_chi[iind]) {
        double pfx = d_F[iind];
        double pfy = d_F[NN + iind];
        double pfz = d_F[2*NN + iind];
        int nbci = d_NBCi[iind];
        if(nbci>=0){
            pfx += d_NBC[3*nbci];
            pfy += d_NBC[3*nbci + 1];
            pfz += d_NBC[3*nbci + 2];
        }
        double vhox = d_vh[iind];
        double vhoy = d_vh[NN + iind];
        double vhoz = d_vh[2*NN + iind];
        double ui = d_u[iind];
        double vi = d_u[NN + iind];
        double wi = d_u[2*NN + iind];
        double ki = d_k[iind];
        double vhx; double vhy; double vhz;
        if (tt==0){
            vhx = dt/max(0.05,ki)/mass * pfx / TWO;
            vhy = dt/max(0.05,ki)/mass * pfy / TWO;
            vhz = dt/max(0.05,ki)/mass * pfz / TWO;
        } else {
            vhx = ((TWO - c*dt)*vhox + TWO*dt/max(0.05,ki)/mass*pfx)/(TWO + c*dt);
            vhy = ((TWO - c*dt)*vhoy + TWO*dt/max(0.05,ki)/mass*pfy)/(TWO + c*dt);
            vhz = ((TWO - c*dt)*vhoz + TWO*dt/max(0.05,ki)/mass*pfz)/(TWO + c*dt);
        }
        int ebcx = d_EBCi[iind];
```

```
        int ebcy = d_EBCi[NN + iind];
        int ebcz = d_EBCi[2*NN + iind];
        if(ebcx<0){
            d_u[iind] = ui + dt*vhx;
        }else{
            d_u[iind] = d_EBC[ebcx]*min(ONE, tt/ntau);
        }
        if(ebcy<0){
            d_u[NN+iind] = vi + dt*vhy;
        }else{
            d_u[NN+iind] = d_EBC[ebcy]*min(ONE, tt/ntau);
        }
        if(ebcz<0){
            d_u[2*NN + iind] = wi + dt*vhz;
        }else{
            d_u[2*NN + iind] = d_EBC[ebcz]*min(ONE, tt/ntau);
        }
        d_vh[iind] = vhx;
        d_vh[NN + iind] = vhy;
        d_vh[2*NN + iind] = vhz;
    }
}

__global__ void calcKbar(float *d_Sf, double *d_Wt, double *d_RM, double *d_W,
                         bool *d_dmg, double *d_kbar, bool *d_chi){
    int tix = threadIdx.x;
    int iind = blockIdx.x * blockDim.x + tix;
    int k = iind/(NX*NY);
    int j = iind%(NX*NY)/NX;
    int i = iind%NX;
    if(tix<NB){
        sh_Sf[tix] = d_Sf[tix];
        sh_Sf[NB+tix] = d_Sf[NB+tix];
        sh_Sf[2*NB+tix] = d_Sf[2*NB+tix];
    }
    __syncthreads();
    float xi = L*(i+HLF) + xl;
    float yi = L*(j+HLF) + yl;
    float zi = L*(k+HLF) + zl;
    double Wt = d_Wt[0];
    double RM = d_RM[0];
```

```
    if (TestBbox(xi,yi,zi) && d_chi[iind]) {
        double kopti = d_W[iind] *NN * RM / Wt;
        double nsm = kopti*hrad;
        double dsm = hrad;
        for (int64_t b = 0; b<NB; b++){
            float dx2 = sh_Sf[b];
            float dy2 = sh_Sf[NB+b];
            float dz2 = sh_Sf[2*NB+b];
            if (TestBbox(xi+dx2, yi+dy2, zi+dz2) && !TestBit(d_dmg, b*NN + iind)) {
                int jind = iind + jadd[b];
                if(d_chi[jind]){
                    double psi = max(ZER, hrad - L0s[b]);
                    nsm += psi * d_W[jind] * NN * RM / Wt;
                    dsm += psi;
                }
            }
        }
        d_kbar[iind] = max(0.0001, min(ONE, d_kbar[iind] + nsm / dsm));
    }
}

__global__ void updateK(double *d_k, double *d_kbar, int *d_NBCi, int *d_EBCi){
    int iind = blockIdx.x * blockDim.x + threadIdx.x;
    if(iind<NN){
        if(d_NBCi[iind]<0 && d_EBCi[iind]<0 &&
           d_EBCi[NN + iind]<0 && d_EBCi[2*NN + iind]<0){
            d_k[iind] = alpha*d_k[iind] + (1 - alpha) * d_kbar[iind];
            d_kbar[iind] = ZER;
        }
    }
}
```

B.3 Peridynamics and Optimization Wrapper Classes (Python)

```
import numpy as np
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import time
import pycuda.reduction as reduction
import pycuda.gpuarray as gpuarray

def makeStencil(hrad = 3.01):
    grad = int(hrad+1)
    xs = np.arange(-grad, grad+1, dtype=np.int32)
    XS,YS,ZS = np.meshgrid(xs,xs,xs)
    xs = np.reshape(XS, [-1])
    ys = np.reshape(YS, [-1])
    zs = np.reshape(ZS, [-1])
    rad = (xs**2 + ys**2 + zs**2)**0.5
    filt = (rad < hrad) & (rad>0.001)
    S = np.vstack([xs[filt], ys[filt], zs[filt]]).T
    return S

class PDMaterial:
    def __init__(self, E, nu, rho, tsi = 1, ecrit = 100000):
        self.E = E
        self.nu = nu
        self.rho = rho
        self.tsi = tsi
        self.ecrit = ecrit

class PDGeometry:
    def __init__(self, bbox, NX, chi, hrad=3.01):
        L = np.float64((bbox[0][1] - bbox[0][0])/NX)
        NY = int(np.round((bbox[1][1] - bbox[1][0])/L))
        NZ = int(np.round((bbox[2][1] - bbox[2][0])/L))
        NN = NX*NY*NZ
        if chi is None:
            chi = np.ones(NN)
        S = makeStencil(hrad)
        NB = S.shape[0]
```

```
        Sf = (L*S).astype(np.float32)
        L0s = (np.sum((L*S)**2, axis=1)**0.5).astype(np.float32)
        jadd = (S[:,0] + NX*S[:,1] + NX*NY*S[:,2]).astype(np.int32)
        self.L = L
        self.NX = NX
        self.NY = NY
        self.NZ = NZ
        self.NN = NN
        self.NB = NB
        self.Sf = Sf
        self.L0s = L0s
        self.jadd = jadd
        self.hrad = hrad
        self.bbox = bbox
        self.chi = chi

class PDBoundaryConditions:
    def __init__(self, geom, ntau = 500):
        self.ntau = ntau
        NN = geom.NN
        NX = geom.NX
        NY = geom.NY
        L = geom.L
        inds = np.arange(NN)
        i = (inds%NX)
        j = (inds%(NX*NY)//NX)
        k = (inds//(NX*NY))
        self.x = L*i + L/2 + geom.bbox[0][0]
        self.y = L*j + L/2 + geom.bbox[1][0]
        self.z = L*k + L/2 + geom.bbox[2][0]
        self.NBCi = np.zeros(NN, dtype = np.int32) - 1
        self.NBC = []
        self.EBCi = np.zeros((3, NN), dtype = np.int32) - 1
        self.EBC = []
        self.geom = geom

    def makeFilt(self, bbox):
        return (self.x>bbox[0][0]) & (self.x<bbox[0][1]) & \
               (self.y>bbox[1][0]) & (self.y<bbox[1][1]) & \
               (self.z>bbox[2][0]) & (self.z<bbox[2][1])
```

```
    def addForce(self, bbox, vec):
        if len(vec)==2:
            vec = np.hstack([vec,0])
        filt = self.makeFilt(bbox)
        self.NBCi[filt] = len(self.NBC)
        self.NBC.append(vec)

    def addDistributedForce(self, bbox, vec):
        if len(vec)==2:
            vec = np.hstack([vec,0])
        filt = self.makeFilt(bbox)
        self.NBCi[filt] = len(self.NBC)
        self.NBC.append(vec/np.sum(filt)/self.geom.L**3)

    def addFixed(self, bbox, val, dims):
        filt = self.makeFilt(bbox)
        for d in dims:
            self.EBCi[d, filt] = len(self.EBC)
        self.EBC.append(val)

    def addFixedFunctional(self, bbox, func):
        filt = self.makeFilt(bbox)
        vals = func(self.x[filt], self.y[filt], self.z[filt])
        for d in range(len(vals)):
            if vals[d] is not None:
                self.EBCi[d, filt] = len(self.EBC) + np.arange(len(vals[d]))
                [self.EBC.append(val) for val in vals[d]]

    def getRotFunc(self, tht):
        def rot(x,y,z):
            rmat = np.array([[np.cos(tht), np.sin(-tht), 0],
                             [np.sin(tht), np.cos(tht), 0],
                             [0, 0, 1]])
            xnew = np.dot(rmat,np.vstack([x,y,z]))
            return [xnew[0]-x, xnew[1]-y, xnew[2]-z]
        return rot

class PDOptimizer:
    def __init__(self, alpha, volfrac, penal, tol=0.00001):
        self.alpha = alpha
        self.volfrac = volfrac
```

```
        self.tol = tol
        self.penal = penal

    def step(self, pd):
        RM = self.volfrac
        d_Wt = pd.sum_reduce(pd.d_W)
        while RM > self.tol:
            cuda.memcpy_htod(self.d_RM, np.array([RM], dtype = pd.dtype))
            self.d_calcKbar(pd.d_Sf, d_Wt, self.d_RM, pd.d_W, pd.d_dmg,
                self.d_kbar, pd.d_chi, block = (pd.TPB, 1, 1),
                grid = ((pd.geom.NN+pd.TPB-1)//pd.TPB, 1 , 1))
            AM = pd.sum_reduce(self.d_kbar).get()/pd.geom.NN
            RM = self.volfrac - AM
        self.d_updateK(pd.d_k, self.d_kbar, pd.d_NBCi, pd.d_EBCi,
            block = (pd.TPB, 1, 1), grid = ((pd.geom.NN+pd.TPB-1)//pd.TPB, 1 , 1))


class PDModel:
    def __init__(self, mat, geom, bcs, opt=None, dtype=np.float64, TPB = 128):
        dt = 1
        k = mat.E/(1-2*mat.nu)/3
        mu = mat.E/(1+mat.nu)/2
        am = 15*mu
        bc = 18*k/np.pi/(geom.hrad*geom.L)**4
        dV = geom.L**3
        mass = .5*dt**2*4./3.*np.pi*(geom.hrad*geom.L)**3*bc/geom.L
        if opt is None:
            alpha = 0
        else:
            alpha = opt.alpha
        source = open("modules\\kernels.cu", "r")
        src = source.read()
        src = src.replace("NBr",str(geom.NB))
        src = src.replace("NNr",str(geom.NN))
        src = src.replace("NXr",str(geom.NX))
        src = src.replace("NYr",str(geom.NY))
        src = src.replace("NZr",str(geom.NZ))
        src = src.replace("Lr",str(geom.L))
        src = src.replace("dtr",str(dt))
        src = src.replace("bcr",str(bc))
        src = src.replace("amr",str(am))
```

```
src = src.replace("kappar",str(k))
src = src.replace("dVr",str(dV))
src = src.replace("massr",str(mass))
if dtype == np.float32:
    src = src.replace("MLTr",str(0.002))
else:
    src = src.replace("MLTr",str(1))
src = src.replace("ntaur",str(bcs.ntau))
src = src.replace("rhor",str(mat.rho))
src = src.replace("cnr",str(0))
src = src.replace("ecritr",str(mat.ecrit))
src = src.replace("SHr",str(4*geom.NB + 1))
src = src.replace("TPB",str(TPB))
src = src.replace("L0s[]","L0s["+str(geom.NB)+"] = "+
    np.array2string(geom.L0s,separator = ',',
    max_line_width=np.nan).replace('[','{').replace(']','}'))
src = src.replace("jadd[]","jadd["+str(geom.NB)+"] = "+
    np.array2string(geom.jadd,separator = ',',
    max_line_width=np.nan).replace('[','{').replace(']','}'))
src = src.replace("alphar",str(alpha))
src = src.replace("hradr",str(geom.L*geom.hrad))
src = src.replace("xlr",str(geom.bbox[0][0]))
src = src.replace("xhr",str(geom.bbox[0][1]))
src = src.replace("ylr",str(geom.bbox[1][0]))
src = src.replace("yhr",str(geom.bbox[1][1]))
src = src.replace("zlr",str(geom.bbox[2][0]))
src = src.replace("zhr",str(geom.bbox[2][1]))
if opt is None:
    src = src.replace("penalr",str(0))
else:
    src = src.replace("penalr",str(opt.penal))
dsize = 8
if dtype == np.float32:
    dsize = 4
    src = src.replace("double","float")
    src = src.replace("sqrt","sqrtf")
self.dtype = dtype
mod = SourceModule(src, options=["--use_fast_math"])
self.d_m = gpuarray.GPUArray([geom.NN], dtype)
self.d_dil = gpuarray.GPUArray([geom.NN], dtype)
self.d_u = gpuarray.GPUArray([3*geom.NN], dtype)
```

```
        self.d_vh = gpuarray.GPUArray([3*geom.NN], dtype)
        self.d_F = gpuarray.GPUArray([3*geom.NN], dtype)
        self.d_cd = gpuarray.GPUArray([geom.NN], dtype)
        self.d_cn = gpuarray.GPUArray([geom.NN], dtype)
        self.d_Sf = gpuarray.to_gpu(geom.Sf)
        self.d_dmg = gpuarray.GPUArray([((geom.NB)*geom.NN + 7)//8], np.bool_)
        self.d_NBCi = gpuarray.to_gpu(bcs.NBCi)
        self.d_EBCi = gpuarray.to_gpu(bcs.EBCi.flatten())
        self.d_NBC = gpuarray.to_gpu(np.array(bcs.NBC).astype(np.float32).flatten())
        self.d_EBC = gpuarray.to_gpu(np.array(bcs.EBC).astype(np.float32))
        self.d_chi = gpuarray.to_gpu(geom.chi.astype(np.bool_))
        self.d_c = cuda.mem_alloc(dsize)
        self.d_k = gpuarray.to_gpu(np.ones(geom.NN, dtype=dtype))
        self.d_W = gpuarray.GPUArray([geom.NN], dtype)
        self.d_Ft = gpuarray.GPUArray([geom.NN], dtype)
        d_calcVolume = mod.get_function("calcVolume")
        self.d_calcDilation = mod.get_function("calcDilation")
        self.d_calcForce = mod.get_function("calcForce")
        self.d_calcDisplacement = mod.get_function("calcDisplacement")
        self.sum_reduce = reduction.get_sum_kernel(dtype, dtype)
        if opt is not None:
            opt.d_kbar = gpuarray.GPUArray([geom.NN], dtype)
            opt.d_RM = cuda.mem_alloc(dsize)
            opt.d_calcKbar = mod.get_function("calcKbar")
            opt.d_updateK = mod.get_function("updateK")
        self.TPB = TPB
        self.geom = geom
        self.opt = opt
        self.bcs = bcs
        self.ft = 0
        d_calcVolume(self.d_Sf, self.d_m, self.d_dmg, self.d_chi,
            block = (TPB, 1, 1), grid = ((geom.NN+TPB-1)//TPB, 1 , 1),
            shared = (4*geom.NB + 1)*4)

    def evalC(self):
        cn1 = self.sum_reduce(self.d_cn).get()
        cn2 = self.sum_reduce(self.d_cd).get()
        if (cn2 != 0.0):
            if ((cn1 / cn2) > 0.0):
                cn = 2.0 * (cn1 / cn2)**0.5
            else:
```

```
            cn = 0.0
        else:
            cn = 0.0
        if (cn > 2.0):
            cn = 1.9
        return np.array([cn], dtype = self.dtype)

    def solve(self, NT, tol = None):
        NN = self.geom.NN
        NB = self.geom.NB
        TPB = self.TPB
        d_Sf = self.d_Sf
        d_u = self.d_u
        d_vh = self.d_vh
        d_F = self.d_F
        d_cn = self.d_cn
        d_cd = self.d_cd
        d_c = self.d_c
        d_dmg = self.d_dmg
        d_NBC = self.d_NBC
        d_EBC = self.d_EBC
        d_NBCi = self.d_NBCi
        d_EBCi = self.d_EBCi
        d_W = self.d_W
        d_k = self.d_k
        d_Ft = self.d_Ft
        d_chi = self.d_chi
        d_dil = self.d_dil
        d_m = self.d_m
        BPG = (NN+TPB-1)//TPB
        t0 = time.time()
        for tt in range(NT):
            self.d_calcDilation(d_Sf,d_u, d_dil, d_dmg, d_m, d_chi,
                block = (TPB, 1, 1), grid = (BPG, 1 , 1), shared = (4*NB + 1)*4)
            self.d_calcForce(d_Sf,d_u, d_dil, d_m, d_dmg, d_F, d_vh,
                d_cd, d_cn, d_EBCi, d_k, d_W, d_Ft, d_chi, block = (TPB, 1, 1),
                grid = (BPG, 1 , 1), shared = (4*NB + 1)*4)
            cuda.memcpy_htod(d_c, self.evalC())
            self.d_calcDisplacement(d_c, d_u, d_vh, d_F, d_NBCi, d_NBC, d_EBCi,
                d_EBC, d_k, d_chi, block = (TPB, 1, 1), grid = (BPG, 1 , 1))
            if tt%50==0 and tol != None:
```

```
            ft = self.sum_reduce(d_Ft).get()
            if ft>self.ft:
                self.ft = ft
            if ft<tol*self.ft:
                break

def get_displacement(self):
    NN = self.geom.NN
    u = self.d_u.get()
    return u[:NN], u[NN:2*NN], u[2*NN:]

def get_fill(self):
    return self.d_k.get()

def get_W(self):
    return self.d_W.get()

def get_coords(self):
    inds = np.arange(self.geom.NN)
    z = self.geom.L*(inds//(self.geom.NX*self.geom.NY) + .5)
    y = self.geom.L*(inds%(self.geom.NX*self.geom.NY)//self.geom.NX + .5)
    x = self.geom.L*(inds%self.geom.NX + .5)
    return x, y, z
```

B.4 Example Execution Script (Python)

```
import numpy as np
import modules.pd as pd

bbox = [[0, 160], [0, 20], [0, 3]]
E = 30e6
nu = 0.25
volfrac = 0.2
penal = 2
NX = 2*int(bbox[0][1])

hrad = 3.01
rho = 1250.
ntau = 1000
alpha = 0.8

mat = pd.PDMaterial(E, nu, rho)
geom = pd.PDGeometry(bbox, NX, None, hrad)
bcs = pd.PDBoundaryConditions(geom, ntau)

bcs.addFixed([[0, 1], [0, 20], [0, 10]], 0, [0,1,2])
bcs.addFixed([[159, 160], [0, 20], [0, 10]], 0, [0,1,2])
bcs.addFixed([[79,81], [0, 1], [0, 10]], -.4, [1])

opt = pd.PDOptimizer(alpha, volfrac, penal)
model = pd.PDModel(mat, geom, bcs, opt, dtype=np.float32)

for _ in range(10):
    model.solve(2000, tol = 0.01)
    opt.step(model)
```