Department: Head
Editor: Name, xxxx@email

# High-Productivity Parallelism with Python Plus Packages (but without a Cluster)

**John Bartlett**
University of Washington-Seattle

**Chris Uchytil**
University of Washington-Seattle

**Duane Storti**
University of Washington-Seattle

***Abstract*—We present two computing projects, peridynamics simulation and numerical integration on implicit domains, for which we realized high performance implementations using Python with appropriate packages. The problems are sufficiently compute-intensive that a straightforward serial implementation is prohibitively slow. While conventional wisdom suggests moving such problems onto a computing cluster, we very directly produced high-performance parallel implementations that effectively perform the computing tasks on a single GPU. For the peridynamics application, the only package needed in addition to Numpy is Numba whose just-in-time compiler allows us to write kernel functions in Python and compile them to run in parallel on a CUDA-enabled GPU. Our approach to numerical integration on implicit domains invokes two additional packages to support interval arithmetic and dynamic parallelism to enable tree-structured recursive refinement. Use of Python (with only kernels requiring dynamic parallelism written in C) enabled rapid development of concise code that successfully achieves significant performance enhancement.**

**WITH THE RIGHT PACKAGES,** Python serves as a very effective tool for productive implementation of scientific codes that achieve high-performance by running in parallel on a modern graphics processing unit [GPU]. Here we present two examples from our research where Python, combined with appropriate Python packages, enabled rapid development of high-performance GPU-parallel applications.

The first example involves peridynamics [7], a mesh-free alternative to finite element analysis [FEA] that offers advantages for analyzing systems with composite materials, damage, or large deformation for which FEA would be pro-

 Published by the IEEE Computer Society  

hibitively expensive due to the need for very fine meshes or repeated re-meshing. The nodewise operations comprising peridynamics are both simple and highly parallelizable, making peridynamics an ideal candidate for GPU parallelization using Python and Numba.

The second example involves a recently developed grid-based approach to numerical integration on implicitly defined domains [11], [10]. The novel aspect we describe here involves incorporating a combination of interval arithmetic methods along with dynamic parallelism to more efficiently eliminate regions that cannot provide a non-trivial contribution to the numerical value of the integral. Here we use two packages that we developed as part of related projects: a package to perform interval arithmetic computations on the GPU, and PyCu, a package that enables direct compilation of C kernels that we can then execute from within Python.

The remainder of the paper presents the peridynamics example followed by the integration example and concludes with a discussion of some key takeaways from our experiences.

## GPU-parallel Peridynamics

Simulation of mechanical behavior has become a standard aspect of the design and optimization of engineered systems, and the most well-known method for performing such simulations is Finite Element Analysis [FEA] [2]. FEA seeks to produce an approximate solution of the governing partial differential equations [PDEs] of continuum mechanics by solving a discretized version obtained using a mesh that approximates the continuous domain. Peridynamics takes an alternative approach. Instead of going to the continuum limit to derive a PDE and then discretizing to obtain a tractable system, peridynamics employs an inherently discrete model based on interacting particles with non-local interaction forces that can reproduce continuum behavior [7]. A concise introduction to the basic formulation of peridynamics is given by Bartlett & Storti [1], so here we provide only the very essentials of the state-based formulation for analysis of 2D elastic deformation under mechanical loading. (Bond-based peridynamics is a bit simpler but too restrictive to model materials with generic values of properties, namely Poisson's ratio $\nu$. More detailed derivations & descriptions of bond- and state-based formulations are given by Silling & collaborators [7], Le & Bobaru [4].)

A peridynamic model discretizes a domain into a collection of material points or nodes. Each node $\mathbf{x}$ interacts with other nodes that lie within a local neighborhood $\mathcal{H}_{\mathbf{x}}$ (according to Newton's law for dynamical behavior) to produce the peridynamic equation

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},\mathbf{t}) = \int_{\mathcal{H}_x} \mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t)dV_{\hat{\mathbf{x}}} + \mathbf{b}(\mathbf{x},t) \quad (1)$$

describing the acceleration $\ddot{\mathbf{u}}(\mathbf{x},\mathbf{t})$ of each node $\mathbf{x}$, 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 $\mathbf{x}$.

State-based peridynamics describes the deformation of a body in terms of the elongation state $\underline{e} = \|\xi + \eta\|$ where $\xi$ is the vector between two nodal (undeformed) coordinates $\mathbf{x}$ and $\hat{\mathbf{x}}$, and $\eta$ is the difference between the displacement vectors of $\mathbf{x}$ and $\hat{\mathbf{x}}$. The force $\mathbf{f}$ in the peridynamic equation is defined in terms of the peridynamic force state $\underline{t}$

$$\underline{t} = 2\Big(k'\theta - \frac{\alpha}{3}(\underline{\omega e^d} \bullet \underline{x})\Big)\frac{\underline{\omega x}}{m} + \alpha\underline{\omega e^d} \quad (2)$$

$$\mathbf{f}(\mathbf{x},\hat{\mathbf{x}},t) = \big(\underline{t}[\mathbf{x},\hat{\mathbf{x}}] + \underline{t}[\hat{\mathbf{x}},\mathbf{x}]\big)\frac{\boldsymbol{\xi}+\boldsymbol{\eta}}{\|\boldsymbol{\xi}+\boldsymbol{\eta}\|} \quad (3)$$

where dilation $\theta$ is defined as

$$\theta = \frac{2(2\nu-1)}{\nu-1}\frac{(\underline{\omega x}) \bullet \underline{e}}{m}. \quad (4)$$

Here $\underline{\omega}$ 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 $\bullet$ operator represents the peridynamic dot product, defined as $\underline{A} \bullet \underline{B} = \int_{\mathcal{H}_x} \underline{A} \cdot \underline{B} dV$ on states $\underline{A}$ and $\underline{B}$ of the bonds of node $\mathbf{x}$. $k'$ and $\alpha$, defined by Le & Bobaru [4], are peridynamic analogies of the material's bulk and shear moduli $k$ and $\mu$.

Note that the formulation we present here is for 2D plane stress; see Silling et al. [7] for the minor modifications necessary for plane strain and 3D elasticity formulations. Together, these equations can be used to determine the acceleration of each node at a given time. From there, a numerical integration technique is needed

**Algorithm 1** Peridynamic operation sequence

```
XI[i, j, k] = COORD[CONN[i,j], k] - COORD[i, k]                          ▷ Initialize ξ
X[i, j] = sum(XI[i,j,k]^2, axis = k)^0.5                                 ▷ Initialize x
M[i] = sum(X[i, j], axis = j)                                            ▷ Initialize m

U = zeros(N, D)                                                          ▷ Initialize displacements
DUDT = zeros(N, D)                                                       ▷ Initialize velocities

for tt = 1:NT do                                                         ▷ For each timestep tt
    NU[i, j, k] = U[CONN[i,j], k] - U[i, k]                              ▷ Calculate η
    E[i, j] = sum((XI[i,j,k] + NU[i,j,k])^2, axis = k)^0.5 - X[i, j]     ▷ Calculate e
    ESM[i] = sum(E[i, j], axis = j)                                      ▷ Calculate (ωx) • e
    DIL[i] = 2(2ν−1)/(ν−1) ESM[i]/M[i]                                   ▷ Calculate θ
    ED[i, j] = E[i, j] - DIL[i]X[i, j]/3                                 ▷ Calculate e^d
    EDSM[i] = sum(ED[i, j], axis = j)                                    ▷ Calculate (ωe^d) • x
    T[i, j] = 2(2ν−1)/(ν−1)(k′ DIL[i] - α/3 * EDSM[i])/M[i] + α ED[i, j] / X[i, j]   ▷ Calculate t
    F[i, k] = sum( (T[i, j] + T[CONN[i, j], ICONN[i, j]])                ▷ Calculate nodal forces
                * (XI[i, j, k] + NU[i, j, k]) / (E[i,j] + X[i, j]), axis = j)
    F[ NBCFILTER ] = NBC[ NBCFILTER ]                                    ▷ Apply Neumann B.C.s
    DUDTH[i, k] = DUDT + Δ_t/2 * (F[i, k]/ρ - ζDUDT[i, k])               ▷ Calculate velocity half-step
    U[i, k] += Δ_tDUDTH[i, k]                                            ▷ Update displacements
    U[ DBCFILTER ] = DBC[ DBCFILTER ]                                    ▷ Apply Dirichlet B.C.s
    DUDT = DUDTH + + Δ_t/2 * (F[i, k]/ρ - ζDUDTH[i, k])                  ▷ Update velocities
end for
```

to evaluate the motion of the nodes which can model either actual dynamic behavior or, by including an artificial damping term $\zeta$, approach to an equilibrium configuration. A number of integration methods may be used and, as is typical in the peridynamics literature, we choose Verlet integration to provide a reasonable balance of stability and computational complexity.

When numerically evaluating these equations in a peridynamic simulation, it is convenient to structure each of the peridynamic states as a tensor, and each equation as a tensor operation. This makes parallelization with the Numba package for Python relatively straightforward: for a serial implementation, we loop over each index in an operation to fill that location of the output tensor; to convert to a parallel implementation, we simply replace these loops with a parallelized grid evaluating over multiple indices concurrently.

**Algorithm 1** details this series of tensor operations. Besides the tensors representing peridynamic states, the following data structures are required for the representation of the domain of interest:

- COORD: N x D coordinate array, where N is the number of peridynamic nodes and D is the dimension of the geometry, containing the undeformed coordinates of each node.
- CONN: N x M connectivity array, where M is the maximum number of bonds a node can have, containing M indices for each node denoting the row of COORD for each neighbor bonded to that node. For a regular grid of points and a given peridynamic horizon size, M can be explicitly calculated.
- ICONN: N x M inverse connectivity array, filled with indices such that CONN[CONN[i,j], ICONN[i, j]] = i.
- NBCFILTER: N x D boolean array, evaluating to true at indices corresponding to points with applied Neumann boundary conditions.
- NBC: N x D Neumann boundary condition array, containing applied forces at indices corresponding to points where those forces are applied.
- DBCFILTER: N x D boolean array, evaluating to true at indices corresponding to points with

applied Dirichlet boundary conditions.
- DBC: N x D Dirichlet boundary condition array, containing applied displacements at indices corresponding to points where those displacements are applied.

To demonstrate the power of this fairly straightforward approach, we evaluate a canonical test problem in peridynamics, the cracked plate, the results of which are shown in **Figure 1**. A vertical displacement is applied to the top and bottom boundaries of a (non-dimensional) square plate with a crack spanning half of its width. The simulation we present here consists of 1 million nodes (a rather typical size for a peridynamics simulation) updated for 10,000 timesteps. To get an idea of the density of such a simulation, the images shown in **Figure 1** only show 2% of the million nodes used. The serial version of the same method, written as vectorized, optimized Numpy array operations took over **9 hours** to run on a 2.4 GHz Intel Xeon Processor. The parallelized simulation ran in **2 minutes and 33 seconds** on a 16 GB Nvidia Tesla P100 GPU. While the example we present is a somewhat pared-down version of a true peridynamic simulation, leaving out functionality such as plasticity and damage evaluation, the cracked plate simulation clearly demonstrates the power of this basic GPU parallelization.

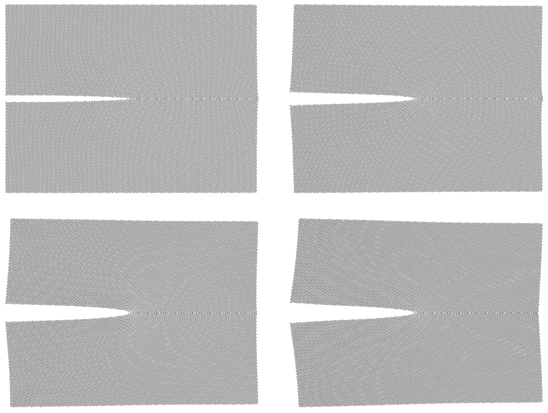

**Figure 1.** A 1-million-node peridynamic simulation of a cracked plate. From left to right, top to bottom: tt = 2500, tt = 5000, tt = 7500, tt = 10000. Displacements shown are scaled by a factor of 10 to enhance visibility,

## Interval enhancement of grid-based integration

Traditionally numerical evaluation of integrals or *quadrature* is presented as a straightforward matter of finding appropriate coefficients to approximate the value of the integral as a linear combination of function values sampled over the domain of integration. This sort of simple approach can produce a reliable estimate of the value of the integral and is indeed appropriate in the case of domains in a 1D space. However, when computing numerical values of integrals on domains in a multi-dimensional space (e.g. computing the area of a surface in 3D ambient space), complications arise associated with parametrizing the domain and specifying the limits of integration (or, equivalently, specifying the geometry of the integration domain). One classical approach is to discretize the integration into simplices with a known parametrization; e.g. compute the area of a surface by approximating the surface by a collection of triangles (i.e. a mesh) and summing the areas of the triangles. Examples of how triangulation can fail (e.g. the Schwarz lantern) have been known since circa 1890 [3] and led us to investigate an alternate approach that arises from the wavelet literature [6] for computing integrals on implicitly defined domains based on function evaluations on a regular grid of the ambient space. The full details of the formulation of the grid-based integration method are given in Yurtoglu et al. [11] and Uchytil & Storti [10] but, for current purposes, the important point is that the scheme involves a modified stencil computation. To evaluate the integral

$$I = \int\int_{\partial\Omega} g(\mathbf{r})\mathbf{dv} \tag{5}$$

where $\mathbf{r}$ is the Cartesian displacement, $g$ is the integrand, $\mathbf{dv}$ is the volume element $dx\,dy\,dz$, and $\partial\Omega = \{\mathbf{r} \mid \mathbf{f}(\mathbf{r}) = \mathbf{0}\}$ is the implicitly defined domain of integration bounding the region $\Omega = \{\mathbf{r} \mid \mathbf{f}(\mathbf{r}) < \mathbf{0}\}$, the grid-based integration method involves three steps:

1) Create a regular grid of points to discretize a box containing the integration domain.
2) Evaluate the integrand $g$ and the implicit defining function $f$ on the gridpoints.

3) Compute the contribution from each gridpoint and sum.

The contribution $q(\mathbf{r})$ from each gridpoint is

$$q(\mathbf{r}) = -\mathbf{g}(\mathbf{r})\frac{\nabla\mathbf{f}(\mathbf{r})\cdot\nabla\chi(\mathbf{r})}{\|\nabla\mathbf{f}(\mathbf{r})\|}\mathbf{\Delta^3} \qquad (6)$$

where $\Delta$ is the spacing between gridpoints, $\chi(\mathbf{r}) = \frac{\mathbf{1}}{\mathbf{2}}(\mathbf{1} - \mathbf{sign}(\mathbf{f}(\mathbf{r})))$ is the occupancy or indicator function that has value 1 in $\Omega$ and 0 outside $\Omega$. The gradients are evaluated in Cartesian coordinates and, to exploit the wavelet-based convergence proof, the derivatives are evaluated using wavelet connection coefficients. However, that detail requires no special attention because connection coefficients for low genus Daubechies wavelets coincide with standard central difference coefficients, so the derivative estimate becomes a standard stencil computation (e.g., with coefficients $\frac{1}{2\Delta}\{-1, 0, 1\}$). Equation 6 defines a modified stencil computation where the derivative estimate for $f(\mathbf{r})$ is not just evaluated, but also multiplied by the derivative estimate for $\chi(\mathbf{r})$ which is likewise computed using the central difference formula. The quadrature estimate $Q$ for the integral $I$ is

$$Q = \sum_{i,j,k} q(\mathbf{r_{i,j,k}}), \qquad (7)$$

where $i, j, k$ are the indices for grid coordinates along the Cartesian axes. A gridpoint is *irregular* and can make a non-trivial contribution only if $\nabla\chi \neq 0$. Non-trivial contributions arise when $\partial\Omega$ is crossed so that $f(\mathbf{r})$ changes sign under the stencil. Such computations parallelize nicely under the single instruction multiple thread (SIMT) model of parallelism implemented in CUDA [9]. The most straightforward CUDA implementation, and the implementation presented in Yurtoglu et al. [11], involves a kernel function that iterates over all points within the discretized domain, evaluates Eq. 6 for some user specified implicit defining function $f$ and integrand $g$, and sums all quadrature contributions.

For purposes of efficiency, if the location of the irregular grid points (the points located near $\partial\Omega$) could be identified before evaluation of the kernel function it would be possible to reduce the number of times Equation 6 is computed. Regions of space where $\mathbf{f}(\mathbf{r})$ does not change sign can be excluded as they do not contribute to the numerical quadrature result. One approach to achieving this goal is to start by launching a kernel on a relatively coarse grid where each grid point represents not just a position but the surrounding 3D interval or voxel, $\hat{\mathbf{r}} = (\hat{x},\hat{y},\hat{z})$, with width along each coordinate direction equal to the grid spacing of the coarse grid. Evaluating an interval extension $F$ of the defining function $f$ over the 3D interval representation of a grid point produces a real output interval $\hat{f} = F(\hat{\mathbf{r}})$ that is guaranteed to contain all possible values of $f(x, y, z)$ for $(x, y, z) \in \hat{\mathbf{r}}$ [5]. If $0 \notin \hat{f}$, then the coarse interval including the grid point cannot contribute to the numerical result and the computation required in that region is finished. For points on the coarse grid where $0 \in \hat{f}$, a new kernel is launched on a more refined grid that samples the enclosing coarse interval. This exclusion/refinement process is repeated until the termination criterion, a maximum refinement depth, is satisfied. At the maximum depth, one final kernel is launched that computes quadrature contributions of the maximally refined local grid. This interval subdivision approach aims to achieve the accuracy associated with a highly refined grid at a significantly reduced computational cost.

In our implementation of the proposed method, when launching a kernel to evaluate a coarse grid we take the approach of computing the output interval, $\hat{f}$, of each grid point on a parallel computational thread. A thread that produces an output interval containing 0 then needs to launch its own computational kernel to evaluate output intervals on a further refined grid. The ability for a thread within a kernel function to launch a new kernel, a process called *dynamic parallelism*, is one of the few CUDA features not supported in Numba, so recursive subdivision via dynamic parallelism requires an alternative library.

In particular, we employed Pycu, a custom package developed by one of the authors (CU) that provides a lightweight Python binding to the complete CUDA Driver API as well as the NVIDIA Runtime Compiler [NVRTC]. Pycu invokes NVRTC to create a fully compiled version of the C kernel that is callable from within Python. With Pycu, we write just the CUDA kernels in C and immediately gain access to

the sought-after dynamic parallelism while maintaining the rest of Python's developer-friendly properties like access to Numpy arrays for data storage and manipulation. Note that PyCUDA also provides much of this functionality and can be used as an alternative. The main advantage of Pycu in this case is its ease of installation and use. Pycu is written completely in Python and does not require external libraries or packages other than the CUDA driver and toolkit.

In addition to writing C versions of the CUDA kernels for recursive refinement and evaluation of quadrature contributions, we also need an interval arithmetic library which supports basic mathematical operations used in the interval extension $F$ of the defining function $f$. In this section, while we focus primarily on implementation details and results of the proposed integration kernel, we first present some relevant information about the custom interval arithmetic library.

**Algorithm 2**

```
1:  function ROOT(quad, domain, h)
2:      size = get_size(domain, h)
3:      depth = get_depth(size)
4:      span = get_span(depth - 1)
5:      children = get_child_count(size, span)
6:      for i = 0 : children.x do
7:          for j = 0 : children.y do
8:              for k = 0 : children.z do
9:                  origin = {i,j,k}*span
10:                 SUBDIVIDE(quad, origin, span,
    h, depth)
11:             end for
12:         end for
13:     end for
```

**Algorithm 3**

```
1:  function INTERNAL(quad, origin, h, depth)
2:      i = threadIdx.x + blockIdx.x*blockDim.x
3:      j = threadIdx.y + blockIdx.y*blockDim.y
4:      k = threadIdx.z + blockIdx.z*blockDim.z
5:      span = get_span(depth - 1)
6:      origin = {i,j,k}*span + origin
7:      SUBDIVIDE(quad, origin, span, h, depth)
```

We think of an interval as a contiguous segment of the real number line and, for each

**Algorithm 4**

```
1:  function LEAF(quad, origin, h)
2:      i = threadIdx.x + blockIdx.x*blockDim.x
3:      j = threadIdx.y + blockIdx.y*blockDim.y
4:      k = threadIdx.z + blockIdx.z*blockDim.z
5:      origin = {i,j,k} + origin
6:      r = origin*h
7:      quad += −g(r) (∇f(r) · ∇χ(r)) / ‖∇f(r)‖ h^3
```

elementary mathematical operation, we employ the corresponding interval extension following Stolfi & De Figueiredo [8] to define a real output interval that contains all possible real outcomes. For purposes of computation, we represent an interval by a pair of numbers corresponding to a lower and an upper bound of the real numbers in the segment. In our implementation, we employ a fixed-length approach and store the bounds as floating point numbers. In the interval arithmetic library, we perform two important tasks: (1) we implement the interval extension of each elementary mathematical function, and (2) we invoke CUDA variants of mathematical operations that control the rounding direction to ensure the containment property of the interval; namely the lower and upper bounds are rounded towards $-\infty$ and $+\infty$ respectively.

The interval subdivision integration method aims to accumulate values only from grid points that can provide a non-trivial quadrature contribution. This is achieved through the use of interval arithmetic in combination with recursive coarse grid subdivision, allowing for the exclusion of regions within the domain that cannot contribute to the quadrature. The recursive process of exclusion or subdivision mirrors the space partitioning effects that arise within a tree structure. There are three kernels involved in the integral computation and, to reflect this close association, the name of each kernel corresponds to the tree-equivalent node type.

- ROOT (**Algorithm** 2): To initiate the integration process, the user calls ROOT with input parameters to specify an axis-aligned bounding box and desired effective sample spacing. Based on those inputs, we determine an appropriate initial coarse grid, and execute a loop

**Algorithm 5**

```
1: function SUBDIVIDE(quad, origin, span, h, depth)
2:     lo = ↓ ((origin - 1)*h) ↓          ▷ ↓: floating point multiplication is rounded down
3:     hi = ↑ ((origin + span + 1)*h) ↑    ▷ ↑: floating point multiplication is rounded up
4:     x̂ = {lo.x, hi.x}
5:     ŷ = {lo.y, hi.y}
6:     ẑ = {lo.z, hi.z}
7:     f̂ = F(x̂, ŷ, ẑ)
8:     if 0 ∈ f̂ then
9:         if (depth - 1) > 0 then
10:            INTERNAL<<<blocks, threads>>>(quad, origin, h, depth - 1)
11:        else
12:            LEAF<<<blocks, threads>>>(quad, origin, h)
13:        end if
14:    end if
```

over the points in the coarse grid with each iteration evaluates the interval extension of $f$. If $0 \in \hat{f}$, the coarse voxel may contribute to the quadrature value and the thread calls INTERNAL to launch a child kernel that refines the sampling of the corresponding voxel. ROOT handles the coarse grid points sequentially (with a grid and block dimension of 1) to avoid performance degradation associated with an excessive number of co-existent kernels.

- INTERNAL (**Algorithm** 3): INTERNAL carries out the concurrent refinement process. Whether called by ROOT or recursively by an earlier instance of INTERNAL, INTERNAL launches a computational grid that refines the sampling of a coarser voxel. We associate each thread with a node on the locally refined grid to enable concurrent evaluation of the output intervals for the refined voxels. If the recursion limit has not been reached, each thread computing an output interval including 0 calls INTERNAL to continue refinement of the grid. When the recursion limit is reached, INTERNAL calls LEAF to complete the computation of the quadrature contributions.
- LEAF (**Algorithm** 4): The LEAF kernel computes quadrature contributions from non-excluded grid points and increments the total quadrature value. The contents of the LEAF kernel match closely with the kernel presented in Yurtoglu et al. [11] that implements the simpler, non-recursive approach based on a uniform grid; the significant distinction being the range over which the kernel computes quadrature contributions within the domain. The uniform sampling kernel needs to densely sample the entire Cartesian box containing the domain of integration while the Leaf kernel only samples over a localized portion within the box.

When an INTERNAL kernel is launched by the thread associated with a coarse grid point, the factor by which the coarse grid point is subdivided corresponds to the number of threads specified in the kernel's launch configuration. We chose to support any refinement along each axis corresponding to a power of 2 so that the subdivision process can be thought of as a $(2^p)^3$–tree. Choosing $p = 1$ produces the equivalent of a traditional octree. However, efficient evaluation requires more than $2^3$ threads per block (a sensible lower bound on the block size is some integer multiple of the number of the 32 computational cores in a typical CUDA streaming multiprocessor) so we refine by a factor larger than 2. Through experimentation we found the optimal value to be $p = 5$ producing the equivalent of a $32^3$–tree.

To obtain some data on relative performance, we ran both the previous algorithm with uniform sampling and our implementation with adaptive refinement on several basic test shapes including a sphere, a superellipse, and a cube defined by

the following implicit functions:

$$f_{\text{sphere}} = \sqrt{x^2 + y^2 + z^2} - r$$
$$f_{\text{superellipse}} = x^6 + y^6 + z^6 - r^6 \quad (8)$$
$$f_{\text{cube}} = \max(|x|, |y|, |z|) - l$$

We chose the sphere to serve as a common shape with a well-known area and uniformly smooth surface. We chose the cube, another common shape with well-known area, but having sharp edges, to test whether discontinuity of the gradient causes any significant problems. We include the superellipse because the higher degree defining function leads to a looser Lipschitz bound on the magnitude of the gradient. As a result, we expect the superellipse to produce somewhat broader output intervals and provide a more stringent test for the interval refinement scheme.

We note first that both the uniform sampling and the recursive refinement algorithms produce quadrature values that agree at single precision, so the focus of the comparison is on computational cost measured in terms of compute time. **Table** 1 summarizes the data obtained by computing the area of each of the three shapes by the two different methods at five different grid spacings. In particular, the columns specify the equivalent number of grid points at maximum refinement depth, the compute time with uniform sampling, the compute time with recursive refinement, the speedup factor (i.e. ratio of compute times), and the fraction of points on the uniform grid evaluated at the leaf nodes of the recursive refinement algorithm. The salient features of the performance comparison include:

- Even at lower resolutions where the overhead of computing interval extensions is relatively large (compared to the computational cost of evaluating the leaf node contributions), the recursive refinement implementation achieves a significant reduction of computation time with speedup factors starting in the range of 5X to 11X.
- As the refinement level increases (corresponding to larger grid point counts), the speedup factors increase steadily (improving by at least 2X with each doubling of the grid refinement) and eventually exceed 100X for each of the example computations.
- While the uniform sampling implementation does offer very significant performance improvement compared to a serial implementation, at the highest level of refinement tested (i.e. the $16384^3$ grid) the computation times for high levels of refinement became intolerable (>10 min.) while the timing for the recursive implementation remained less than 3s.
- For the examples tested, the recursive refinement implementation evaluate the integral based on a $1024^3$ grid at interactive speed (on the order of 10ms or 100 evaluations per second).

## Conclusion

We have presented two examples of computational problems arising from our research whose solutions seemed particularly well suited for GPU parallelization but did not seem particularly easy to implement. In each case, we turned to Python, with appropriate add-on packages, to simplify and speed the realization of parallel implementations that achieve high performance.

The first example centered on peridynamic simulation, and we translated a basic peridynamics formulation into a series of tensor operations. Writing the peridynamics simulation as a sequence of tensor operations enabled a serial implementation involving a sequence of Numpy array operations that was straightforward but did not provide the desired level of computational efficiency. We converted each Numpy operation to a Numba device function that can compute the output tensor in parallel. Leveraging the capabilities of Numba, this intuitive approach enabled us to produce a parallel implementation without substantially more effort than it took to produce the serial implementation. Thus the developer overhead of parallelization was relatively small, while the performance benefit of parallelization was significant corresponding to a speedup factor of over 200X for the example presented. With peridynamics simulations that previously took several hours to run completing in a matter of minutes, the argument for running such peridynamics simulations on the GPU is compelling.

We also presented a recursive interval refinement update of a grid-based approach to numerical evaluation of integrals on implicitly defined

**Table 1. Performance results of both the integration method presented in Yurtoglu et al. [11] as well as our new recursive implementation. Both algorithms were evaluated and timed on a computer equipped with a GTX 1080 and Intel i7-6700k.**

| | Grid Size | Dense (s) | Recursive (s) | Speedup factor | % of dense grid evaluated |
|---|---|---|---|---|---|
| $f_{\text{sphere}}$ | $1024^3$ | 0.07 | 0.008 | 11.4 | 49.80% |
| | $2048^3$ | 0.89 | 0.04 | 22.3 | 6.28% |
| | $4096^3$ | 10 | 0.14 | 71.4 | 3.08% |
| | $8192^3$ | 82 | 0.57 | 143.9 | 1.55% |
| | $16384^3$ | - | 2.2 | - | 0.77% |
| $f_{\text{superellipse}}$ | $1024^3$ | 0.05 | 0.01 | 5 | 59.42% |
| | $2048^3$ | 0.77 | 0.04 | 19.3 | 7.38% |
| | $4096^3$ | 9.1 | 0.17 | 53.5 | 3.61% |
| | $8192^3$ | 76 | 0.63 | 120.6 | 1.81% |
| | $16384^3$ | - | 2.6 | - | 0.91% |
| $f_{\text{cube}}$ | $1024^3$ | 0.04 | 0.01 | 4 | 61.62% |
| | $2048^3$ | 0.75 | 0.04 | 18.8 | 7.44% |
| | $4096^3$ | 9 | 0.17 | 52.9 | 3.65% |
| | $8192^3$ | 75 | 0.68 | 110.3 | 1.84% |
| | $16384^3$ | - | 2.8 | - | 0.93% |

domains that requires a broader range of software tools. Again, Python augmented with appropriate packages (a custom package for executing interval arithmetic on the GPU and Pycu, a package that provides full access to CUDA APIs including dynamic parallelism at the cost of writing just the kernel functions in C) enabled very rapid development - a single developer working over a weekend - of code that provides orders of magnitude improvement even compared to a previous parallel implementation.

The case studies presented here serve as illustrations of the ease and efficacy of using Python to access the computing power of modern GPU resources. In both examples, the overhead of developing the parallelizations was dwarfed by the resulting accelerations achieved. In the case of the peridynamic example, the time necessary to parallelize the code was comparable to the time previously required to run a single serial simulation. Our experiences with these examples support the conclusion that the use of Python offers a low barrier of entry to GPU computing, allowing for both rapid development and rapid execution of scientific computations.

## Source Code

The software developed by the authors discussed in this work can be found in the following repositories:

- https://github.com/jd-bartlett96/peridynamics
- https://github.com/uchytilc/Recursive-Grid-Based-Integration
- https://github.com/uchytilc/PyCu

## Acknowledgment

The authors gratefully acknowledge the support provided by U.S. Army Research Office grant W911NF-17-1-0595.

## REFERENCES

1. Bartlett, J., Storti, D.: A generalized fictitious node approach for surface effect correction in peridynamic simulation. Journal of Peridynamics and Nonlocal Modeling pp. 1–11 (2021). DOI 10.1007/s42102-020-00045-8
2. Hughes, T.J.: The finite element method: linear static and dynamic finite element analysis. Courier Corporation (2012)
3. Koenderink, J.J.: Solid shape. MIT press (1990)
4. Le, Q.V., Bobaru, F.: Surface corrections for peridynamic models in elasticity and fracture. Computational Mechanics **61**, 499–518 (2018). DOI 10.1007/s00466-017-1469-1
5. Moore, R.E.: Interval analysis, vol. 4. Prentice-Hall Englewood Cliffs (1966)
6. Resnikoff, H.L., Raymond Jr, O., et al.: Wavelet analysis: the scalable structure of information. Springer Science & Business Media (2012)
7. Silling, S.A.: Reformulation of elasticity theory for discontinuities and long-range forces. Journal of the Mechanics and Physics of Solids **48**(1), 175–209 (2000). DOI 10.2172/1895

8. Stol, J., De Figueiredo, L.H.: Self-validated numerical methods and applications. In: Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer, vol. 5. Citeseer (1997)
9. Storti, D., Yurtoglu, M.: CUDA for engineers: an introduction to high-performance parallel computing. Addison-Wesley Professional (2015)
10. Uchytil, C., Storti, D.: A coarea formulation for grid-based evaluation of volume integrals. Journal of Computing and Information Science in Engineering **20**(6) (2020). DOI 10.1115/1.4047355
11. Yurtoglu, M., Carton, M., Storti, D.: Treat all integrals as volume integrals: a unified, parallel, grid-based method for evaluation of volume, surface, and path integrals on implicitly defined domains. Journal of computing and information science in engineering **18**(2) (2018). DOI 10.1115/1.4039639

**John Bartlett** is a PhD Candidate in Mechanical Engineering at the University of Washington-Seattle. Mr. Bartlett received a B.S. (2017) and M.Eng. (2018) in Mechanical Aerospace Engineering from Cornell University. Mr. Bartlett's work has focused on numerical simulations and computational elasticity, along with high-accuracy and high-performance computing methods. Contact him at jdbart@uw.edu.

**Christopher Uchytil** is a PhD Candidate in Mechanical Engineering at the University of Washington-Seattle. Mr. Uchytil recieved a B.S. (2016) in Astronomy and Physics and a M.S. (2018) in Engineering from University of Washington. His research is focused primarily on applications of high-performance computing for geometric modeling and additive manufacturing. Mr. Uchytil can be contacted at uchytilc@uw.edu.

**Duane Storti** is Professor of Mechanical Engineering at the University of Washington-Seattle. Dr. Storti received a B.S. in Applied Engineering Physics in 1979 and M.S (1981) and Ph.D. (1984) in Theoretical Applied Mechanics from Cornell University. He is a member of ASME, ACM, and AAUP and is the author (with M. Yurtoglu) of "CUDA for Engineers: An Introduction to High-Performance Parallel Computing". Prof. Storti has published works in a variety of fields including dynamics of coupled nonlinear oscillators, geometric design, 3D printing, and applications of symbolic and parallel computation. His ongoing research interests focus on high-performance computing to support advances in additive manufacturing. Contact him at storti@uw.edu.