Proceedings of the ASME 2021 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2021
August 17-20, 2021, Virtual, Online

DETC2021-XXXX

A SINGLE-CARD GPU IMPLEMENTATION OF PERIDYNAMICS

John D. Bartlett*
Department of Mechanical Engineering
University of Washington
Seattle, Washington, 98105
Email: jdbart@uw.edu

Duane Storti
Department of Mechanical Engineering
University of Washington
Seattle, Washington, 98105
Email: storti@uw.edu

ABSTRACT

The rapid development of parallelization technology over the recent decades has provided a promising avenue for the acceleration of meshfree simulation methods. One such method, peridynamics, is particularly well-suited for parallelization due to the simplicity of the operations which must occur at each material point. However, while MPI-based parallelization (Message-Passing Interface; a method for CPU-based parallelization) of peridynamic problems is commonplace, GPU parallelization of peridynamics has received far less attention. While GPU technology may have once been an inferior option to MPI parallelization for peridynamics, modern GPU cards are more than capable of handling substantially sized peridynamics problems. This paper presents the parallelization of the peridynamic method for single - card GPU computing, providing a schematic for a compact parallel approach. The resulting method is tested with CUDA on a NVIDIA Tesla P100 card with 16 GB of memory. The per - node memory requirements for each data structure used are evaluated, as well as the per - node execution times for each operation in a million - node benchmark test. This setup is shown to provide speedup factors over 200 for problems sized up to several million nodes, therefore indicating such a GPU is more than adequate for the single - card parallelization of the peridynamic method.

*Address all correspondence to this author.

1 INTRODUCTION

Solid mechanics modelling is a critical step in the analysis of a variety of mechanical systems, from automotive engines to tectonic plates. The most common method for such analysis is the Finite Element Method (FEM). Like the similar Finite Difference and Finite Volume methods, FEM relies on generating fixed meshes connecting points within the domain of interest [1]. These meshes are used for the numerical approximation of operators such as spatial derivatives within the domain, which in turn are used to solve the governing equations of the systems.

While such mesh-based methods are robust and extremely useful, there are a variety of problems for which such approaches are not appropriate. For example, meshed simulation methods tend to perform poorly for problems involving sharp discontinuities within the domain, where mesh-based derivative operators begin to fail, or in problems dealing with large deformations or time-dependent spatial relationships between points in the domain [2].

A variety of meshfree methods have been developed to better handle analysis of such problems. Numerous meshfree methods have been formulated, each designed for a specific class of analysis problems, but almost all of these methods share a common strategy of trading computational efficiency for flexibility in representation of the physical problem. The most common method of achieving this is by discretizing the domain of interest into material points, rather than meshed volume elements, applying some set of rules dictating the interactions between these points, then iteratively re-evaluating the states of the material

Copyright © 2021 by ASME

points based on those rules until some equilibrium is achieved. Meshfree methods such as the Finite Point Method [3], Material Point Method [4], and Peridynamics [2] all employ such a discretization technique. Such a discretization process allows for extremely flexible definitions of material point interactions and therefore the modeling of a broad set of physical problems, but generally with the cost of far more required computations than meshed methods [5].

However, with the rapid advancement of parallel computing technologies, many such meshfree methods have become far more computationally tractable. Most meshfree methods relying on pointwise discretization can be formulated such that the computations that must be performed at each material point can be performed in parallel, leaving only the iterative updating of material point states to be performed serially. Continued developments of parallel computing technologies drive down the computational cost that must be paid to achieve the flexibility of such methods, making them a more and more attractive alternative to traditional meshed methods.

One meshfree method that falls into this category of high parallelization potential is peridynamics. Developed in 2000 primarily for analysis of crack propagation and other damage in elastic bodies, peridynamics replaces local derivatives in the governing equations of continuum mechanics with integral equations evaluating the interactions between points in a material domain with their neighboring points [2]. While parallelized implementations of peridynamics are almost as old as the method itself, due to the high memory-per-material-point requirements of the method parallelization efforts have historically been focused around MPI parallelization on the CPU [6, 7]. GPU cards were typically held to have insufficient memory to model reasonably-sized problems without needing communication between GPU cards, which can undercut many of the computational improvements offered by the parallelization [8]. But, with the rapid improvements of GPU cards over the past two decades, the problem size a single GPU can handle is quickly increasing.

The aim of this paper is to present an effective approach to creating a single-card GPU implementation of peridynamics, and to demonstrate the usefulness of such an implementation. Peridynamics is particularly well-suited for GPU parallelization among other meshfree methods as many of its use cases do not require re-evaluation of nodal connections, a particularly expensive step that poses difficulty for parallelization. More importantly, unlike other meshfree methods, GPU parallelization of peridynamics is rather under-developed. Although publications occasionally reference GPU-parallelized implementations of the method [9], a detailed description of the parallelized computations used is generally lacking. While MPI-based parallelizations have received more detailed analysis [6], the methodology used in such implementation is aimed towards a small collection of processors working in parallel. Modern GPUs, on the other hand, have the capacity to run the computations required for each node in parallel, which leads to substantially different methodology than what is used in MPI parallelization. As such, an effective GPU implementation would be a significant advancement in the field of peridynamics.

2 Methodology

2.1 A Brief Overview of the Peridynamic Method

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 of x, $\mathscr{H}_x$. The governing peridynamic equation, describing the acceleration $\ddot{\mathbf{u}}(\mathbf{x},\mathbf{t})$ of each node $\mathbf{x}$, is given as

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

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 [2] and [10], respectively. State-based peridynamics is far more commonly used, as it resolves the inability of bond-based peridynamics to represent materials with arbitrary Poisson's ratio; as such the analysis here will be for the state-based formulation.

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 $\underline{e} = ||\xi + \eta||$, where ξ and η 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 $\underline{A} \bullet \underline{B} = \int_{\mathscr{H}_x} \underline{A} \cdot \underline{B} dV$ on states $\underline{A}$ and $\underline{B}$ of the bonds of node $\mathbf{x}$. The dilation θ of $\mathbf{x}$ is defined as

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

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

$$W = \frac{k'\theta^2}{2} + \frac{\alpha}{2}(\underline{\omega e^d}) \bullet \underline{e^d} \tag{3}$$

Copyright © 2021 by ASME

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

$$k' = \begin{cases} k & \text{(3D Elasticity)} \\ k + \frac{\mu}{9}\frac{(\nu+1)^2}{(2\nu-1)^2} & \text{(Plane Stress)} \\ k + \frac{\mu}{9} & \text{(Plane Strain)} \end{cases} \quad (4)$$

$$\alpha = \begin{cases} \frac{15\mu}{m} & \text{(3D Elasticity)} \\ \frac{8\mu}{m} & \text{(2D Elasticity)} \end{cases} \quad (5)$$

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 [11].

$$\underline{t} = \begin{cases} \frac{2k'\theta}{m}\underline{\omega x} + \alpha\underline{\omega e^d} & \text{(3D Elasticity)} \\ \frac{2(2\nu-1)}{\nu-1}\left(k'\theta - \frac{\alpha}{3}(\underline{\omega e^d} \bullet \underline{x})\right)\frac{\underline{\omega x}}{m} + \alpha\underline{\omega e^d} & \text{(Plane Stress)} \\ 2\left(k'\theta - \frac{\alpha}{3}(\underline{\omega e^d} \bullet \underline{x})\right)\frac{\underline{\omega x}}{m} + \alpha\underline{\omega e^d} & \text{(Plane Strain)} \end{cases} \quad (6)$$

where the force vector acting between two nodes $\mathbf{x}$ and $\hat{\mathbf{x}}$ is the 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 $\frac{\xi+\eta}{||\xi+\eta||}$; i.e.

$$\mathbf{f}(\mathbf{x}, \hat{\mathbf{x}}, t) = \left(\underline{t}[\mathbf{x}, \hat{\mathbf{x}}] + \underline{t}[\hat{\mathbf{x}}, \mathbf{x}]\right)\frac{\xi + \eta}{||\xi + \eta||} \quad (7)$$

Thus, Eq. 1 can be evaluated in terms of the displacements of each node to determine the accelerations of each node. Finally, some integration method is then required for evaluating the motion of the nodes. While the convergence rate of the model is strongly dependent on the method chosen for time integration, for the purposes of parallelization analysis a simple implementation of Verlet integration is employed at each node [12].

$$\dot{\mathbf{u}}(t + \frac{\Delta_t}{2}) = \dot{\mathbf{u}}(t) + \frac{\Delta_t}{2}(\ddot{\mathbf{u}} - \zeta\dot{\mathbf{u}}(t)) \quad (8a)$$

$$\mathbf{u}(t + \Delta_t) = \mathbf{u}(t) + \Delta_t\dot{\mathbf{u}}(t + \frac{\Delta_t}{2}) \quad (8b)$$

$$\dot{\mathbf{u}}(t + \Delta_t) = \dot{\mathbf{u}}(t + \frac{\Delta_t}{2}) + \frac{\Delta_t}{2}(\ddot{\mathbf{u}} - \zeta\dot{\mathbf{u}}(t + \frac{\Delta_t}{2})) \quad (8c)$$

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

2.2 Description of Computation Representation & Operations

The following data structures are used to represent the geometric domain of interest and efficiently perform the computations required for peridynamic analysis. These data structures are constant throughout the peridynamic simulation and can therefore be generated as a precomputation; as such, the generation method is not within the scope of the paper.

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

Note that while implementations for less common boundary condition types are not presented here, most can be represented in a similar paradigm with minimal modification.

Before the peridynamic operation sequence can be summarized, an influence function ω must be selected. A convenient choice is $\omega = \frac{1}{\underline{x}}$, which meets the requirement of being spherical while substantially simplifying the calculations which must be performed. Eq. 9 becomes

$$\theta = \begin{cases} \frac{3\bullet\underline{e}}{m} & \text{(3D Elasticity)} \\ \frac{2(2\nu-1)}{\nu-1}\frac{1\bullet\underline{e}}{m} & \text{(Plane Stress)} \\ \frac{2\bullet\underline{e}}{m} & \text{(Plane Strain)} \end{cases} \quad (9)$$

Copyright © 2021 by ASME

where m can now be defined as $1 \bullet \underline{x}$. Eq. 10 becomes

$$\underline{t} = \begin{cases} \frac{2k'\theta}{m} + \frac{\alpha \underline{e}^d}{\underline{x}} & \text{(3D Elasticity)} \\ \frac{2(2\nu-1)}{m(\nu-1)}\left(k'\theta - \frac{\alpha}{3}\left(\underline{e}^d \bullet 1\right)\right) + \frac{\alpha \underline{e}^d}{\underline{x}} & \text{(Plane Stress)} \\ \frac{2}{m}\left(k'\theta - \frac{\alpha}{3}\left(\underline{e}^d \bullet 1\right)\right) + \frac{\alpha \underline{e}^d}{\underline{x}} & \text{(Plane Strain)} \end{cases} \quad (10)$$

The computational peridynamic method for 2D Plane Stress is summarized in Algorithm 1, and can be readily extended to the Plane Strain or 3D Elasticity cases.

This algorithm serves as a reasonable computational baseline for both serial and parallel implementations. An efficient serial implementation can be obtained via the Numpy mathemetical package for Python with a nearly verbatim copy of this algorithm, inserting the appropriate Numpy array slicing syntax.

Algorithm 1 Peridynamic operation sequence

1: XI[i, j, k] = COORD[CONN[i,j], k] - COORD[i, k]
2: X[i, j] = sum(XI[i,j,k]^2, axis = k)^0.5
3: M[i] = sum(X[i, j], axis = j)
4:
5: U = zeros(N, D)
6: DUDT = zeros(N, D)
7:
8: **for** tt = 1:NT **do**
9: NU[i, j, k] = U[CONN[i,j], k] - U[i, k]
10: E[i, j] = sum((XI[i,j,k] + NU[i,j,k])^2, axis = k)^0.5 - X[i, j]
11: ESM[i] = sum(E[i, j], axis = j)
12: DIL[i] = $\frac{2(2\nu-1)}{\nu-1}$ESM[i]/M[i]
13: ED[i, j] = E[i, j] - DIL[i]X[i, j]/3
14: EDSM[i] = sum(ED[i, j], axis = j)
15: T[i, j] = $\frac{2(2\nu-1)}{\nu-1}$(k' DIL[i] - $\frac{\alpha}{3}$ EDSM[i])/M[i] + α ED[i, j]/X[i, j]
16: F[i, k] = sum((T[i, j] + T[CONN[i, j], ICONN[i, j]]) * (XI[i, j, k] + NU[i, j, k])/(E[i, j]+X[i, j]), axis = j)
17: F[NBCFILTER] = NBC[NBCFILTER]
18: DUDTH[i, k] = DUDT + $\frac{\Delta_t}{2}$(F[i, k]/ρ - ζDUDT[i, k])
19: U[i, k] += Δ_tDUDTH[i, k]
20: U[DBCFILTER] = DBC[DBCFILTER]
21: DUDT = DUDTH + + $\frac{\Delta_t}{2}$(F[i, k]/ρ - ζDUDTH[i, k])
22: **end for**

2.3 Notes on Parallel Implementation

Algorithm 1 can also be relatively directly translated to CUDA via the Numba package for Python. In this baseline implementation, there are 3 parallelized precomputation functions (lines 1-3) and 12 parallelized iteration functions (lines 9-20). For each of these functions the dimensionality of the CUDA grid used corresponds to that of the ouput array; each GPU thread used fills one index of the output array. For reference, the lengths of the i-, j-, and k-axes are N, M, and D respectively, where D is in the range 1-3 and M is typically < 10, < 50, or < 150 for 1D, 2D, and 3D problems respectively. N is necessarily much larger and therefore is the critical dimension defining the problem size.

In a single-GPU implementation of this algorithm, the problem size which can be represented is limited by the memory of the GPU; as such, analysis & management of memory consumption is crucial. The approximate per-node memory requirements of each necessary array and the total algorithm are summarized in Table 1.

Unfortunately, theoretically evaluating the time complexity of a GPU program is considerably more nuanced than doing so for its spatial complexity, as the processor workload is dependent on much more than the number of computations in each iteration as in a typical serial CPU program. However, for this minimally-optimized algorithm, a primary determining factor in an operation's speed which can be easily evaluated is the number of global memory accesses required by that function. In this baseline implementation, all operations are treated as discrete functions so therefore all arrays are stored in global memory, and any array access shown is a global memory access. Table 2 includes a per-thread total of array accesses for each function, which should roughly correspond to the speed of that operation.

3 Results

Timing evaluations were performed on serial and parallelized implementations of Algorithm 1. In peridynamics, the selected grid resolution can have a dramatic effect of the timesteps required for convergence to equilibrium; additionally, convergence criteria can vary widely across different use cases. Since rate of temporal convergence is not the focus of this paper, all timings are given in terms of the time required for a single iteration of the operation(s) in question. These values are obtained by taking the average speed of the operation(s) for the first 1000 iterations. In all cases a variance of less than 5% of the average speed was observed across the iterations, with no notable change in speed as the iterations progressed; as such, it is reasonable to conclude that the timings presented can be scaled to longer-duration simulations.

For the serial implementation, simulations were performed on a 2.4 GHz Intel Xeon Processor and timings were obtained using the Python time package. For the parallel implementation, simulations were performed on a 16 GB Nvidia Tesla P100 GPU.

Copyright © 2021 by ASME

Array	Data type	Memory per node (Bytes)		
		General	D = 2, M = 50	D = 3, M = 150
COORD	float64	8D	16	24
CONN	int32	4M	200	600
ICONN	int8	M	50	150
NBCFILTER	boolean	D	2	3
NBC	float64	8D	16	24
DBCFILTER	boolean	D	2	3
DBC	float64	8D	16	24
XI	float64	8MD	800	3600
X	float64	8M	400	1200
M	float64	8	8	8
U	float64	8D	16	24
DUDT	float64	8D	16	24
DUDTH	float64	8D	16	24
NU	float64	8MD	800	3600
E	float64	8M	400	1200
ESM	float64	8	8	8
DIL	float64	8	8	8
ED	float64	8M	400	1200
EDSM	float64	8	8	8
T	float64	8M	400	1200
F	float64	8D	16	24
Total	–	16MD + 37M + 50D + 32	3590	12916

TABLE 1. Approximate per-node memory requirements for typical 2- and 3-D implementations of Algorithm 1

Timings for individual functions were obtained using Nvidia's nvprof tool, while timings for overall iteration execution were again obtained with the time package. The test problem used is a simple elastic beam under tension, in 2D plane stress. Figure 1 shows the timing results for an entire iteration step as a function of the problem size (the number of nodes N) for both implementations, as well as the speedup factor lent by the parallelization.

Table 2 lists the per-operation breakdown of the overall runtime for a million-node benchmark. Every operation tested yields substantial acceleration, with all parallelized operations running at least 2 orders of magnitude quicker that their CPU counterparts. The most significant benefit was found in the most expensive computation, the calculation of **F**, which achieved a operation-wise speedup of over 400x.

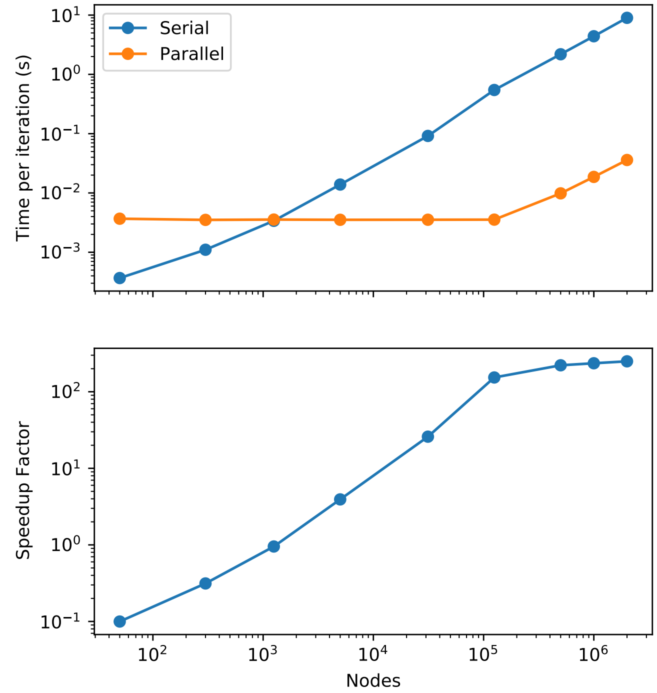

FIGURE 1. Top: Total execution time for a single iteration of Algorithm 1. The blue and orange lines depict the serial and parallel implementations respectively. Bottom: Ratio of parallel execution time to serial execution time.

4 Discussion

The suitability of the peridynamic method for GPU parallelization is immediately clear from the results summarized in Figure 1. As can be seen in the figure, the GPU used can simulate problem sizes up to $\sim 1 \times 10^5$ nodes in a relatively constant speed of $\sim 4 \times 10^{-3}$ second per timestep. Past this point, the GPU runs out of threads and must begin queuing nodes for evaluation, causing the computation time to begin to increase at a slightly lower rate than the CPU. This leads to a plateau in the speedup factor at $\sim 2 \times 10^2$. The single Nvidia Tesla P100 effectively handles the computational load resulting from a simulation with 2 million nodes. The limiting factor on the problem size manageable for the scope of this study was in fact not the peridynamic method evaluated here, but the method used to generate the geometry data for these models. This finding motivates ex-

Copyright © 2021 by ASME

Line Number	Description	Global Memory Accesses per Thread			Execution Time (s) N=1×10^6, D=2, M=50	
		General	D = 2, M = 50	D = 2, M = 150	Serial	Parallel
1	Calculate ξ	3	3	3	8.89×10^{-1}	3.85×10^{-3}
2	Calculate $\underline{x}$	D	2	3	3.06×10^{-1}	4.56×10^{-3}
3	Calculate m	M	50	150	9.41×10^{-2}	4.45×10^{-4}
9	Calculate η	3	3	3	8.89×10^{-1}	3.85×10^{-3}
10	Calculate $\underline{e}$	2D + 1	5	7	2.82×10^{-1}	6.37×10^{-3}
11	Calculate $\underline{e} \bullet 1$	M	50	150	9.41×10^{-2}	4.45×10^{-4}
12	Calculate θ	2	2	2	4.26×10^{-3}	5.67×10^{-5}
13	Calculate $\underline{e}^d$	3	3	3	1.08×10^{-1}	2.52×10^{-3}
14	Calculate $\underline{e}^d \bullet 1$	M	50	150	9.41×10^{-2}	4.45×10^{-4}
15	Calculate $\underline{t}$	5	5	5	2.11×10^{-1}	3.52×10^{-3}
16	Calculate $\mathbf{F}$	8M	400	1200	1.66	4.11×10^{-3}
17	Apply Neumann B.C.	1	1	1	1.97×10^{-3}	1.26×10^{-5}
18	Increment $\dot{\mathbf{u}}$	2	2	2	2.06×10^{-2}	1.08×10^{-4}
19	Increment $\mathbf{u}$	1	1	1	7.64×10^{-3}	4.63×10^{-5}
20	Apply Dirichlet B.C	1	1	1	1.97×10^{-3}	1.26×10^{-5}
Total	–	10M + 2D + 19	523	1525	3.37	2.15×10^{-2}

TABLE 2. Per-operation breakdown of timing theory & results. Note that functions on lines 1-3 are precomputations only; as such their values are not included in the total at the bottom, which aims to quantify the workload on the GPU for each iteration of the simulation. The execution times shown are for a million-node benchmark problem.

ploration of parallelized geometry generation methods in future work.

However, it is important to interpret these results in the appropriate context. The algorithm presented is a minimal version of peridynamics, not including functionality like evaluation of plasticity and damage, or handling issues like the peridynamic surface effect, which would need to be corrected for a more exact simulation [13, 14]. While the operations necessary for such functionality can certainly be organized into a similarly compact & parallelizable form as the peridynamic operations presented here, this would of course increase the per-node spatial requirements and thus reduce the problem size representable on a single GPU.

A study of Algorithm 1 and Tables 1 & 2 yields a number of questions meriting further investigation. For example, there are a number of points where a tradeoff was made between time & space resources, such as the choice to store XI instead of calculating it on the fly. A detailed exploration of these tradeoffs could yield substantial improvement. Additionally, while there is some correspondence between memory accesses and GPU execution time for the individual operations, there are several notable outliers. In particular, the elogation calculation, which was the most expensive computation, had relatively few memory accesses, which could indicate a suboptimal discretization of the GPU grid used. Finally, it is generally worth noting that a number of other optimizations could be employed, such as shared memory usage, condensing functions to limit memory accesses, and so on.

5 Conclusion

We have presented a novel GPU parallelization of the essential elements of a peridynamic simulation method. For problems of representative size (with node count on the order of 10^7), our GPU-parallel implementation reduces runtimes by a factor of greater than 200X compared to a CPU implementation. While many further embellishments of peridynamics simulation are possible, the results presented for the example treated in this paper provide evidence of the promise of GPU-based parallelism

Copyright © 2021 by ASME

to provide significant advancements in the effectiveness and efficiency of peridynamics simulations. Moreover, as access to high-end GPUs is expanding through publicly available cloud-based resources, GPU-based parallelism holds the promise to play a significant role in the "democratization" of peridynamics by providing a broader spectrum of researchers access to the combination of software and hardware needed to participate in cutting edge peridynamics research.

REFERENCES

[1] Groetsch, C., 2003. "Functional analysis". In *Encyclopedia of Physical Science and Technology*, R. A. Meyers, ed., 3rd ed. Academic Press, New York, pp. 337 – 353.

[2] Silling, S., 2000. "Reformulation of elasticity theory for discontinuities and long-range forces". *Journal of the Mechanics and Physics of Solids,* ***48***, pp. 175–209.

[3] Oñate, E., Perazzo, F., and Miquel, J., 2001. "A finite point method for elasticity problems". *Computers & Structures,* ***79***(22), pp. 2151 – 2163.

[4] Steffen, M., Wallstedt, P., Guilkey, J., Kirby, R., and Berzins, M., 2008. "Examination and analysis of implementation choices within the material point method (mpm)". *Computer Modeling in Engineering and Sciences,* ***31***(2), pp. 107–127.

[5] Liu, G., 2016. "An overview on meshfree methods: for computational solid mechanics". *International Journal of Computational Methods,* ***13***(05), p. 1630001.

[6] Gerstle, W., 2015. *Introduction To Practical Peridynamics: Computational Solid Mechanics Without Stress And Strain*. Frontier Research In Computation And Mechanics Of Materials And Biology. World Scientific Publishing Company.

[7] Lee, J., Oh, S. E., and Hong, J.-W., 2017. "Parallel programming of a peridynamics code coupled with finite element method". *International Journal of Fracture,* ***203***(1-2), pp. 99–114.

[8] Kaiser, H., Diehl, P., Lemoine, A. S., Lelbach, B. A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S. R., Gupta, N., Heller, T., et al., 2020. "Hpx-the c++ standard library for parallelism and concurrency". *Journal of Open Source Software,* ***5***(53), p. 2352.

[9] Diehl, P., and Schweitzer, M. A., 2014. "Efficient particle-based simulation of dynamic cracks and fractures in ceramic material".

[10] Silling, S., Epton, M., Weckner, O., Xu, J., and Askari, E., 2007. "Peridynamic states and constitutive modeling". *Journal of Elasticity,* ***88***(2), pp. 151–184.

[11] Silling, S. A., 2010. "Linearized theory of peridynamic states". *Journal of Elasticity,* ***99***(1), pp. 85–111.

[12] Parks, M. L., Lehoucq, R. B., Plimpton, S. J., and Silling, S. A., 2008. "Implementing peridynamics within a molecular dynamics code". *Computer Physics Communications,* ***179***(11), pp. 777–783.

[13] Le, Q. V., and Bobaru, F., 2018. "Surface corrections for peridynamic models in elasticity and fracture". *Computational Mechanics,* ***61***, pp. 499–518.

[14] Bartlett, J., and Storti, D., 2021. "A generalized fictitious node approach for surface effect correction in peridynamic simulation". *Journal of Peridynamics and Nonlocal Modeling*.

Copyright © 2021 by ASME