PyBONDEM-GPU: A discrete element bonded particle Python research framework – Development and examples

Sven Dressler[1]*, *Daniel N.* Wilke[1]

[1]Mechanical Engineering Department, University of Pretoria, 0083 Hatfield, South Africa

Abstract. Discrete element modelling (DEM) is widely used to simulate granular systems, nowadays routinely on graphical processing units. Graphics processing units (GPUs) are inherently designed for parallel computation, and recent advances in the architecture, compiler design and language development are allowing general-purpose computation to be computed on multiple GPUs. Application of DEM to bonded particle systems are much less common, with a number of open research questions remaining. This study outlines a Bonded-Particle Research DEM Framework, PyBONDEM-GPU, written in Python. This framework leverages the parallel nature of GPUs for computational speed-up and the rapid prototype flexibility of Python. Python is faster and easier to learn than classical compiled languages, making computational simulation development accessible to undergraduate and graduate engineers. PyBONDEM-GPU leverages the Numba-CUDA module to compile Python syntax for execution on GPUs. The framework enables research of fibre pull-out from fibre-matrix embeddings. Bonds are simulated between all interacting particles. The performance of PyBONDEM-GPU is compared against Python CPU implementations of PyBONDEM using the Numpy and Numba-CPU Python modules. PyBONDEM-GPU was found to be 1000 times faster than the Numpy implementation and 4 times faster than the Numba-CPU implementation to resolve forces and to integrate the equations of motion.

1 Introduction

The use of computational parallelization techniques for solving numerical problems has become widespread with the advent of improved hardware and systems architecture. Many problems in engineering and the sciences are well suited for parallelization. One such problem is the solution of the Discrete Element Method (DEM), a computationally demanding technique which resolves the interaction of a collection of interacting particles [1,2].

Interactions between interacting particles can be either cohesive or non-cohesive [1–3]. Cohesive materials can be simulated by allowing for tension transmitting bonds between interacting particles. These bonds can also transmit other forces such as torsion, bending moments and shear. Forces are calculated according to the inter-particles tangential and normal strains and rotations. A damage law can be applied to these bonds to simulate bond strengths.

The application of DEM to bonded particle systems have recently received more and more attention [4–6], as GPU compute extends DEM to improved particle shape representations and more advanced physics within particle systems. This paper outlines a research framework that was developed to research bonded DEM particle systems using the Python programming language and GPU acceleration through Numba-GPU.

The parallelization of DEM can be achieved using four primary methods, depending on hardware availability, as follows:

- Multi-CPU parallelization uses the Message Passing Interface (MPI) to facilitate communication between the processors [7,8].
- Multi-thread CPU parallelization takes advantage of the multicore and multi-thread nature of modern CPUs [9].
- GPU parallelization takes advantage of modern GPUs which offer a large number of simultaneously executable threads and large amounts of fast memory [10–12].
- CPU-GPU parallelization leverages heterogeneous architectures to combine the strengths of GPU and CPU parallelization techniques [13].

Initial GPU development required DirectX and OpenGL programming. As Nvidia GPUs were being used to solve more general computing problems, a programming language better suited for developers was developed. In 2007, Nvidia developed the CUDA programming framework which allows the rapid development of programs leveraging GPU hardware written in C++ or C.

NVIDIA introduced the compute unified device architecture programming paradigm known as CUDA to simplify the development of programs [14]. Python modules such as PyCUDA and NUMBA have since then

* Corresponding author: dressler.sven@gmail.com

A video is available at https://doi.org/10.48448/x5qe-nb17 been developed to allow for Python scripts to be executed using the parallel architecture of GPUs [15,16]. Although Python Numba-GPU is less efficient than optimized C-CUDA, it does realize around 50% - 85% performance of optimized C-CUDA [16]. In turn, Python enables rapid prototyping and development required for research, making GPU computing accessible to engineers.

1.1 Discrete element method

The Discrete Element Method (DEM) with soft contact was first proposed by Cundall [1,17] to investigate the behaviour of granular assemblies. DEM models discrete particles (such as discs, spheres or polyhedra) through interact through contact forces. Particle kinetics for soft contact are resolved through though constitutive relationship that defines the force-displacement interaction of contact forces at contact points. Deformation of the particles themselves is assumed to be sufficiently small (compared to the deformation of the overall assembly) that they can be assumed to be rigid. Newton’s second law is applied to the translational and rotational motion of individual particles [3] to define particle kinematics. Damping forces and moments are often applied to reduce spurious oscillations in the system. Bonds, with an associated damage law, can be simulated between particles and transmit a combination of shear, bending and normal forces.

1.2 General-purpose computing on graphics processing units

GPUs are special-purpose cards that are used in the computing environment to handle graphics operation such as image generation, resizing, recolouring and 3D rendering [19]. The GPU is an independent computing device that is controlled from the CPU.

The GPU has on-chip RAM and computational nodes designed primarily for the mostly parallel task of image generation. The host sends image data to the GPU, which processes the data and returns it to the host.

The GPU has multiple streaming multiprocessors that execute 32-thread warps within thread blocks. In turn, thread blocks are arranged in thread grids. Each thread-warp operates under the Single Instruction Multiple Thread (SIMT) parallel computing paradigm extending Flynn’s computing taxonomy. Although, all threads in a warp execute the same instruction in lock-step at all times, SIMT support conditional execution through predicated execution and instruction replay [14]. Each thread within a thread grid is assigned a unique global thread ID that can be used to access data in global memory. Thereby allowing distinct data to be accessed by each thread, such as different pixels in an image for image processing tasks [19].

1.3 Python modules

Python is an interpreted programming language that has increased in popularity in engineering, science and data science communities in recent years [19,20]. For various reasons, standard Python programs do not execute quickly when compared to compiled languages such as C, C++ and Fortran [15]. Several Python modules have been developed around compiled and optimized functions written primarily in C. These modules include SciPy (written for statistics, signal processing, optimization and image processing tasks), Numpy (handling of vast arrays of data) and Numba [15,16,21,22]. Numba is a high-performance Python compiler that provides tools to compile functions written in Python and Numpy using the industry-standard LLVM compiler [15]. Numba also supports execution on the parallel architecture of GPUs (13–15). In particular, Numba supports NVIDIA GPUs through Numba CUDA or AMD GPUs through Numba ROCm, making GPU compute readily available.

The functions to be compiled need to be written using a syntax that is allowed by the compilers. Numba includes a just-in-time (JIT) compiler that allows specially written Python functions to be executed on Nvidia GPUS using the CUDA programming framework. Numba has orders of magnitude performance gains over pure Python.

2 Methodology

PyBONDEM-GPU was written using the Python programming language to carry out a DEM simulations for bonded particles. Numba was used to optimize the program for execution on a GPU. The program and data structure are discussed in further detail below.

The program is comprised of five stages:

1. During the initial stage, the model comprising all the interacting particles is generated. The particle and bond property tables are generated and sent to the GPU.
2. The second stage generates bonds between adjacent particles. A collision detection algorithm is employed to populate the list of particle bonds with all possible interacting particles.
3. Collision detection for the third phase is only invoked when the maximum particle displacement in the system is large enough that new particle interactions could occur. When particle contacts are assumed not to change during a simulation, only a single collision detection step at the start of the simulation is required. A spatial hashing algorithm resolves this step.
4. The fourth phase calculates the contact and bond forces.
5. During the fifth phase, the equations of motion are integrated, and boundary conditions are enforced.

At each stage in the program, relevant parameters are updated and stored on the GPU so that they can be used for the next phase in the program. Data can be copied from the GPU device onto the host (CPU and RAM) for tracking system histories.

Checking each pair of particles for possible interaction would result in computational difficulties as the number of particles increases. Reducing the number of computationally intensive checks to identify possible collisions is the primary goal of broad phase collision detection. This first requires, the nearest neighbour search to identify particle neighbours.

3 Numerical results

In this study, an Nvidia K2200 graphics card is used to simulate fibre loading and fibre pull-out. The scaling for 1000 to 10000 particles is reported using Numpy, Numba-CPU and Numba-GPU code implementations. All three implementations used the same code structure. The execution speed for different numbers of particles was measured using the "time" Python library. The three code implementations were:

- Baseline: data is stored in Numpy arrays as custom Numpy data types. This code is executed sequentially on the CPU.
- Numba-CPU: data is stored in Numpy arrays as custom Numpy data types. Numba's "Autojit" function is used to compile the functions using the LLVM compiler. This code is executed sequentially on the CPU.
- Numba-GPU: data is stored in Numpy arrays as custom Numpy data types. Numba's "CUDA" compiler is used to compile specially written functions for execution on the GPU.

3.1 Isolated fibre loading example

The developed framework is used to simulate the axial loading of a cylinder of particles generated with a hexagonal close-packed structure. The execution time to resolve forces and to integrate the equation of motion was timed for 100 iterations. The average time is reported as the execution speed.

A cylinder of particles with hexagonal closest packing (HCP) is simulated. One end of the cylinder is kept fixed while the other end is displaced at a constant velocity. Different cylinder radii demonstrate the scaling of the number of particles in the simulation.

The average execution times for the force determination and integration are shown in Figure 1. The Numba optimized code executed significantly faster than the Numpy code (between 1000 and 1500 times faster for Numba-CPU optimized and between 1000 and 5500 times faster for Numba-GPU).

Notably, the CUDA code executes in a constant time that is independent of the number of particles, for systems where the number of particles is less than 6000. In contrast, CPU executed codes execute slower as the number of particles increase. This behaviour can be understood from the way instructions are executed on the GPU. Under the SIMT paradigm, all threads are executed on the GPU simultaneously with instruction broadcast across them. If more threads are required than are available on the GPU (or more resources such as memory are required) then the sets of threads are executed sequentially.

The result is that if for any number of launched threads less than the total possible number of threads on the GPU, the execution time is the same. In this study an Nvidia K2200 graphics card is used, which has a maximum of 10,240 threads available. The number of threads launched for each kernel is either equal to the number of particles for collision detection and equation of motion integration, or the number of bonds to resolve the forces. It can be seen from Figure 1 that once the number of particles exceeds the available threads, the execution time increases as the thread sets are executed sequentially.

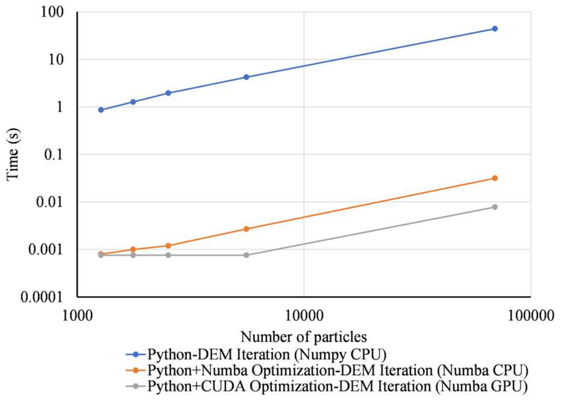

Figure 1. Execution speed for different optimization approaches.

3.2 Embedded fibre pull-out example

The example simulated in this paper is the extraction of a stiff fibre from a more pliable matrix. The matrix is fixed at one end, and the fibre is extracted away from the fixed end (Figure 2). The bonds at the interface between the fibre and matrix obey a bilinear softening rule which is a function of the shear displacement at the interface.

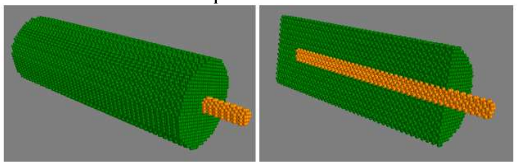

Figure 2. 3D models (cross section on right) of the simulated fibre (orange) embedded in a matrix (green), particle diameter = 0.5m.

The interfacial bond shear force magnitudes during the debonding of the fibre are plotted along the length of the fibre for different extents of debonding in Figure 3. The analytically predicted shear forces are also plotted [23]. There is good agreement between the analytical prediction and the numerical result from the DEM model.

It can be shown that the debonding force/displacement behaviour follows a compound path and requires an arc-length limiting algorithm to extract the full stress-strain path. Arc-length algorithms are often utilized in (semi-)implicit time integration techniques such as finite element models (FEM), but not explicit time integration approaches such as DEM.

An arc-length limiting algorithm can be mimicked in DEM using a loading/unloading strategy. The loading/unloading algorithm limits the damage that occurs along the interface. Once reached, the fibre and matrix are unloaded to their starting position without accruing additional damage before being loaded again. This approach proved useful in estimating the full force/displacement curve for this problem, as shown in Figure 4.

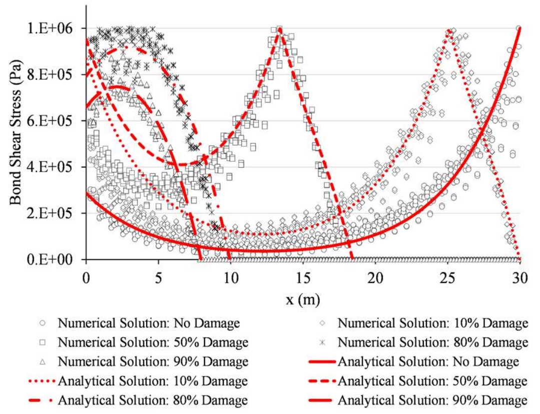

Figure 3. Bond shear stresses (in the z-direction) plotted against the bond centroid location along the interface length for a particle diameter of 0.5m. The bond shears are plotted for five points during the pull-out curve. The analytically predicted shear force is also plotted.

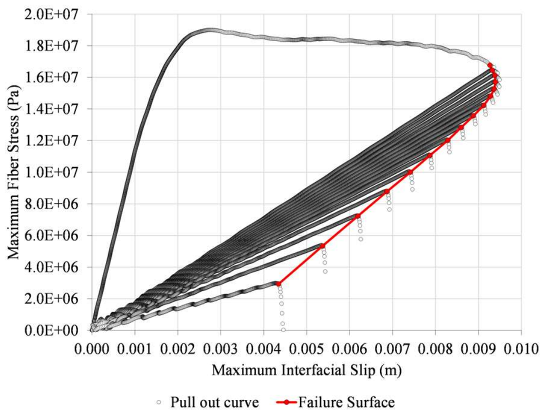

Figure 4. Full failure surface. The failure surface is defined as the minimum combination of interfacial stress slip and fibre stress which results in the remaining bonds accruing damage. It is apparent that the previous history of damage to the interface plays a role in the shape of the failure surface. Beyond a maximum interfacial slip, further damage to the interface occurs at reducing load and slip.

4 Conclusion

PyBONDEM-GPU, our newly proposed Discrete Element Bonded Particle Research Framework written in Python, demonstrated that research specific and flexible research frameworks can be established for DEM research, while still leveraging the computational benefits of GPUs. The research specific DEM code developed for this study was used to investigate the debonding behaviour of an embedded fibre.

The Python programming language together with the Numba module, was used to develop a GPU accelerated double-precision DEM code. The DEM was developed to investigate systems of bonded particles which undergo little relative motion. It could therefore be safely assumed that new collisions did not form, which allowed for additional speed-up of the simulation.

The developed framework was 1000 times faster than and equivalent Numpy implementation and 4 times faster than an equivalent Numba-CPU implementation for force determination and equation of motion integration.

The code presented in this paper is available at: https://github.com/SDressler-2020/BondDEM

References

[1] P.A. Cundall, O.D.L. Strack, Geotechnique **29**, 1 (1979)

[2] D.O. Potyondy, P.A. Cundall, Int. J. Rock Mech. Min. Sci. **41**, 8 (2004)

[3] J. Rojek, C. Labra, O. Su, E. Oñate, Int. J. Solids Struct. **49**, 13 (2012)

[4] A. Lisjak, G. Grasselli, J. Rock Mech. Geotech. Eng. **6**, 4 (2014)

[5] J. Shang, Y. Yokota, Z. Zhao, W. Dang, Constr. Build. Mater. **185**, 120-137 (2018)

[6] Y. Ouyang, Q. Yang, X. Chen, Appl. Sci. **7**, 686 (2017)

[7] O. Surakhi, M. Khanafseh, S. Sarhan, Adv. Sci. Technol. Eng. Syst. **3**, 152-160 (2018)

[8] R. Berger, C. Kloss, A. Kohlmeyer, S. Pirker, Powder Technol. **278,** 234-247 (2015)

[9] Y. Shigeto, M. Sakai, Particuology **9**, 4 (2011)

[10] J. Steuben, G. Mustoe, C. Turner, J. Comput. Inf. Sci. Eng. **16**, 3 (2016)

[11] Q. Liu, W. Wang, H. Ma, Int. J. Numer. Anal. Methods Geomech. **44**, 2 (2020)

[12] N. Govender, D. Wilke, P. Pizette, J. Khinast, EPJ Web Conf. **240**, 12-15 (2017)

[13] X. Yue, H. Zhang, C. Luo, S. Shu, C. Feng, *Parallel Computational Fluid Dynamics* (Springer Berlin Heidelberg, 2014)

[14] Nvidia. *OpenCL Programming Guide for the CUDA Architecture*, Version 3.2, (2010)

[15] S.K. Lam, A. Pitrou, S. Seibert, *Numba: A LLVM-based python JIT compiler*. Proc. Second Work. LLVM Compil. Infrastruc.t HPC, LLVM. (2015)

[16] L. Oden, *Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing*. Proc - 2020 28th Euromicro. Int. Conf. Parallel, Distrib. Network-Based Process PDP 2020, 216–223 (2020)

[17] P.A. Cundall, *A Computer Model for Simulating Progressive Large-Scale Movements in Blocky Rock Systems.* Proceedings Symp. Int. Soc. Rock Mech. (1971)

[18] J. Nickolls, D. Kirk, Appendix C of *Computer Organization and Design*, (2009)

[19] J.Y. Chen, *GPU technology trends and future requirements.* in Proceedings of Tech. Dig – Int. Electron. Devices Meet, IEDM. (2009)

[20] T.E. Oliphant, Comput. Sci. Eng. **9,** 10-20 (2007)

[21] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, et al. Nat. Methods. **17**, 3 (2020)

[22] S. van der Walt, M. Aivazis, Comput. Sci. Eng. **13**, 2 (2011)

[23] Z. Chen, W. Yan, Mech. Mater. **91**, 119-135 (2015)