Murcury: A Parallel On-Chip Simulator of Soft-Body Robots

Dawson Cohen , Amr El-Azizi , Hod Lipson
Department Of Mechanical Engineering, Columbia University New York, USA

***Abstract*—We present Mercury, a CUDA-based C++ soft-body simulator that provides a 6x speedup over similar simulation software by limiting robot body size to the size of the GPU's on-chip cache. Existing simulators support robots of arbitrary size, but this limits their ability to utilize cached memory without mesh partitioning. By limiting robot size, Mercury is able to fit all mass information into the on-chip GPU cache and dramatically improve the speed of uncoalesced memory access and atomic operations. Mercury provides a significant speedup for a size of robot that exhibit complex dynamics is are suitable for research in the areas of evolutionary optimization and reinforcement learning.**

I. Goals and Motivation

Soft robotics offers incredible potential for safe human-robot interaction and for adaptability to complex environments. Automated design of soft robots may offer rapid prototyping, customization, and exploration of a high-dimensional design space. The goals of this research is to lay the groundwork for future exploration of soft robot design by evolution.

The primary computational task in automated soft robot design lies in soft body simulation. Existing GPU-accelerated soft body simulators are built for arbitrarily large elements. As a result, they are unable to fully utilize the GPU's functionality such as on-chip data caches. Thus, the first goal of this paper is to introduce a custom GPU-accelerated simulator that surpasses comparable simulation software in performance, leveraging the full potential of GPU architecture.

The second goal is to establish frameworks in the simulation that will allow customization to a variety of soft-robotics tasks, thus allowing potential users to customize it to their needs. The optimal model for a simulator varies based on the parameters being optimized, and having the flexibility to vary them creates a more robust and useful simulator that can be made specific to a variety of uses.

II. Method/Technical Approach

A. Robot Encoding

Simulated robots in Mercury are represented as soft bodies defined by three primary units:

1) Point masses that hold mass and location information
2) Springs that connect two masses and hold material information
3) Triangular Faces that define the surface of the body

Note, to minimize the cached memory requirement, all point masses are assumed to have unit mass.

Robot movement is made possible by expansion and contraction of active "muscle" springs, supplemented by the passive actuation of "bone" and "tissue" springs. Each material is defined by a spring constant k, the maximum relative change in length ΔL_0, an activation frequency ω, and an activation phase ϕ. Spring activation is defined as follows:

$$L = L_0(1 + \Delta L_0 \sin(\omega t + \phi))$$

Currently, the simulator supports 254 materials, each of which are constructed from a combination of 2 the 22 base materials, with an additional option for air. The current defaults for base materials follow Table I, where Muscles vary in phase and period.

	k	ΔL_0	ω	ϕ
Muscle (20)	50	0.14	f(x)	f(y)
Tissue	40	0	0	0
Bone	100	0	0	0
Air	0	0	0	0

TABLE I
Base Material, where f(x) and f(y) for Muscles vary for each of the 20 base muscles. f(x) takes the form of xω. f(y) takes the form yπ.

B. Kinematic Solver

The Mercury physics simulator uses a fixed time-step kinematic solver to update mass locations based on external environmental forces and internal spring forces with specified actuation schemes.

Mercury supports two types of kinematic solvers to evolve system dynamics: a force-based solver that has a simplified computation model but can be unstable at large time steps, and an extended position-based dynamics (XPBD) solver that runs more compute-intensive operations but is more stable at higher time steps [1].

At each time step, each kinematic solver applies environmental forces such as gravity, collisions, and hydraulic forces. Hydraulic forces are split into a lift and drag term and calculated along triangular face elements of the mesh using equation 1.

$$F_d = \frac{1}{2}\rho A((C_D - C_L)(\vec{v}\cdot\vec{n})\vec{v} + C_L\|\vec{v}\|^2\vec{n}) \qquad (1)$$

Where $\vec{\rho}$ is the fluid density, A is the area of the triangular face, $\vec{v}$ is the velocity of the face centroid as calculated from the velocities of the connected masses, $\vec{n}$ is the face normal, C_D is the drag coefficient, and C_L is the lift coefficient.

The force solver then applies body forces from each spring to its connected masses using Hooke's law:

$$F = k\frac{(x_2 - x_1)}{\|x_2 - x_1\|}(\|x_2 - x_1\| - L) \tag{2}$$

where k is the spring constant. From this point, each solver diverges:

1) Force-Based: The force based solver follows the process in Algorithm 1, wherein the classical Newtonian physics relationships for force (F = ma), acceleration, velocity ($v = \frac{da}{dt} + v_0$), and position ($x = \frac{dv}{dt} + x_0$) are used in order to calculate the position at the next time-step.

Algorithm 1 Force-Based Simulation Loop

$F \leftarrow$ environmental forces
for all springs **do**
  compute F_s using equation 2
  $F \leftarrow F + F_s$
end for
Update positions $x^{n+1} \leftarrow x^n + \Delta t v^n + F\Delta t^2$
Update velocities $v^{n+1} \leftarrow \zeta\frac{1}{\Delta t}(x^{n+1} - x^n)$

2) XPBD: The XPBD solver projects mass locations to satisfy a distance constraint for each spring using Jacobi iteration as described in Algorithm 2. A Lagrange multiplier is calculated for each mass using the distance constraint gradient C_l and inverse spring stiffness α.

Algorithm 2 XPBD Simulation Loop [1]

$F \leftarrow$ environmental forces
predict position $\tilde{x} \leftarrow x^n + \Delta t^2 F$
for all springs **do**
  compute $\Delta\lambda$ using equation 3
  compute Δx using equation 4
  $\tilde{x} = \tilde{x} + \Delta x$
end for
Update positions $x^{n+1} \leftarrow \tilde{x}$
Update velocities $v^{n+1} \leftarrow \zeta\frac{1}{\Delta t}(x^{n+1} - x^n)$

$$\nabla C = \frac{x_2 - x_1}{\|x_2 - x_1\|} \quad \Delta\lambda_i = \frac{\|x_2 - x_1\|}{\nabla C^2 + \alpha/\Delta t^2} \tag{3}$$

$$\Delta x = \nabla C \Delta\lambda \tag{4}$$

C. Design Overview

Efficient GPU algorithms share the following characteristics:

1) Computations can be run simultaneously
2) Minimal data transfer to and from the CPU
3) A contiguous and predictable memory access pattern

While simulation methods on the GPU easily satisfy parallel computation and minimum data transfer from host to device, they struggle to satisfy coalesced reads and writes to memory.

Environmental forces and integration updates are coalesced as each thread can be assigned a mass by the thread ID. However, spring forces require near-random access to the mass buffer as there is no guarantee of correlation between the memory location of a spring and the memory location of the connected masses. This is a significant performance bottleneck that exists in soft-body simulators such as Titan [2].

To address this slowdown, mass information is brought from global memory to shared memory (i.e. L1 cache) with coalesced reads. Forces are calculated and added in this L1 cache and the integrated mass position and velocity are written back to global memory in a coalesced manner. This shared memory scheme drastically reduces the performance slowdown with much faster read and write speeds. As a consequence of this strategy, a robot's masses must fit within the 48 kB or the maximum amount of shared memory in a thread block. This also requires that a thread block only operates on robots that fit within this shared memory limit. Concretely, 36 bytes of information must be stored for each mass as described in Table II.

Variable	Type	Bytes
Position	float3	12
Velocity	float3	12
Force/Lagrangian	float3	12

TABLE II
CACHED MASS INFORMATION

This results in a maximum robot size of 49152/36 = 1365 masses. Mercury robots have no restriction on the number of springs and faces.

D. Data Architecture

The simulator combines all robots' information into a mass buffer, spring buffer, and face buffer. Each soft body is assigned a chunk of each buffer of size maximum element count, where the maximum element is determined by the largest simulated robot. This allows the simulator to load a specific robot's mass information into shared memory for a thread block.

E. Alternate Environment Models

Mercury also provides support for two alternative environment models to evaluate robots within. The first is a simplified variation on the ocean environment, where drag is applied proportionate to velocity at each point-mass. This removes the slowdown of calculating surface area by creating a simplified model of drag, which is less representative of real physics. The second is a land-based environment, with gravity, friction, a floor, and collision between point-masses. Results will focus on the primary environment, water with surface drag, but each option has been fully tested and implemented for any who seek to use it.

F. Visualizer

The simulator comes equipped with a visualizer that can run live along with the simulator. It accepts newly created robots or can load in pre-saved ones for visualization after optimization. The visualizer has useful tools, such as speeding up and slowing down the simulation speed, generating new

random robots, and visualizing drag forces. The visualizer can export a video of a group of robots and can run in headless mode for rendering on remote terminals.

III. Experiments

Here we discuss two robot morphologies used to evaluate and test the simulator. A voxel-based robot used as a standard robot type to compare with other simulation software such as Titan, and a robot type constructed from a neural network.

A. Voxel Robot

A standard robot morphology where each robot consists of N voxels and each voxel consists of 8 masses and a material with X springs connecting them. The materials of the springs are generated by combining the materials of each intersecting voxel for a given mass. Any voxel assigned as air deletes all connected springs.

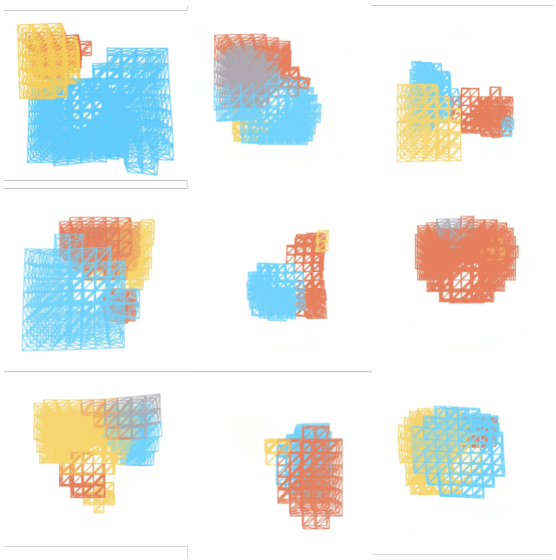

Fig. 1. Random voxel robots

B. NN Robot

We present a new robot morphology. Mass locations are determined by passing random points sampled from a Guassian distribution through a neural network, similar to a diffusion model. The network maps these random locations to both a new location 3D space and a material. The new locations are normalized to fit within a unit sphere. The network architecture currently utilizes a series of MLP layers and a RELU activation function. For the final layer, softmax is used just on the materials to select the one with highest probability. See the footnote [1] for sample NNRobot solutions.

A surface is constructed from the resulting point cloud using AlphaShape [3] and a tetrahedral mesh is fit to the interior points. Importantly, this construction places no limit on the distance between mapped mass locations. If springs are allocated to masses too close together, simulation of this body runs the risk of losing precision resulting in severe instability. To remedy this, springs of distance below $\epsilon = 0.001$ are considered invalid.

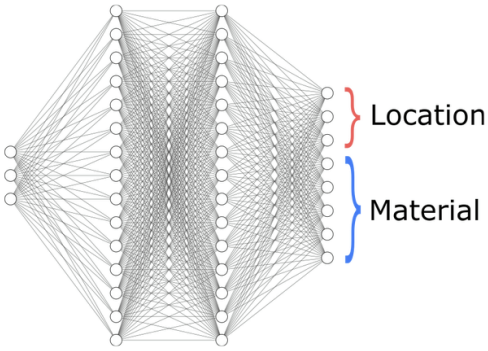

Fig. 2. Demonstration of NN Morphology

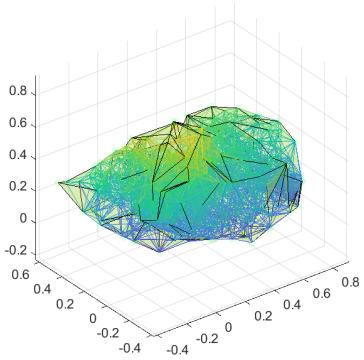

Fig. 3. Random NN robot with alpha-shape surface

IV. Results

We benchmark the simulator under various conditions using the Voxel Robots to provide a standardized metric that is comparable with other simulators. For each benchmark, we create more voxel robots at maximum size to scale up the number of springs being evaluated in parallel. We evaluate our benchmark on a 2080ti. We find that we achieve a performance of approximately 6e9 springs per second starting at around 5e8 springs, handling up to 1e11 total springs. This provides an approximately 6x speedup over Titan, which performs at 1e9 on a 2080ti GPU [4].

V. Conclusions

Mercury is a soft-body simulator optimized for fast simulation of multiple parallel soft-body robots within a small to medium sized frame. It achieves performance far exceeding alternatives at the size range while being compatible with various environment and robot architectures. While developed primarily for simulation, evolution, and development of soft-body robots, the framework is flexible enough to be applied to other soft body simulations and tasks.

[1] Sample NNRobot solutions

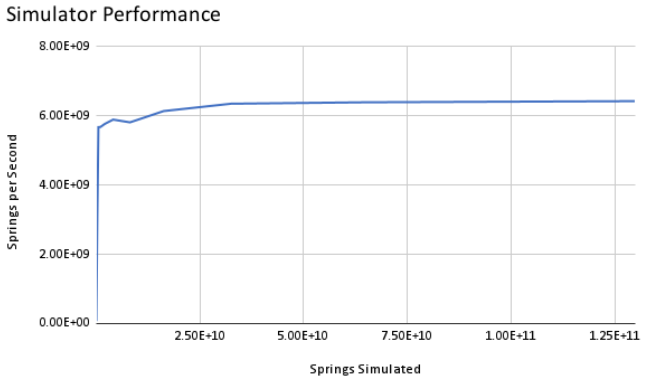

Fig. 4. A graph contrasting the total springs simulated every second vs the total amount of springs evaluated. As we add more springs to the parallelism simulator, we can see that performance rises as parallelism is fully utilized, then plateaus at the maximum.

VI. Future Work

With the simulation framework complete, Mercury is ready for exploring more complex tasks. One of the core motivations was the implementation of evolutionary development for soft robots, and exploring the various parameters that would lead to the best results utilizing the combination of Mercury and the NN Robot Architecture.

References

[1] M. Macklin, M. Müller, and N. Chentanez, "Xpbd: Position-based simulation of compliant constrained dynamics," in *Proceedings of the 9th International Conference on Motion in Games*, ser. MIG '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 49–54. [Online]. Available: https://doi.org/10.1145/2994258.2994272

[2] J. Austin, R. Corrales-Fatou, S. Wyetzner, and H. Lipson, "Titan: A Parallel Asynchronous Library for Multi-Agent and Soft-Body Robotics using NVIDIA CUDA," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 7754–7760.

[3] T. K. F. Da, S. Loriot, and M. Yvinec, "3D alpha shapes," in *CGAL User and Reference Manual*, 5.6 ed. CGAL Editorial Board, 2023. [Online]. Available: https://doc.cgal.org/5.6/Manual/packages.html#PkgAlphaShapes3

[4] C. M. Labs, "Massively-parallel simulation of soft and hybrid soft-rigid robots," Nov. 2019. [Online]. Available: https://www.youtube.com/watch?v=76jSeaZEbJ0