COLORADO SCHOOL OF MINES

Project Presentation

CSCI-582 —— Dr. Mehmet Belviranli

Group 1 —— Shaun Kannady
TQ Bill Huynh

Paper Overview

Paper Title

Titan: A Parallel Asynchronous Library for Multi-Agent and Soft-Body Robotics using NVIDIA CUDA

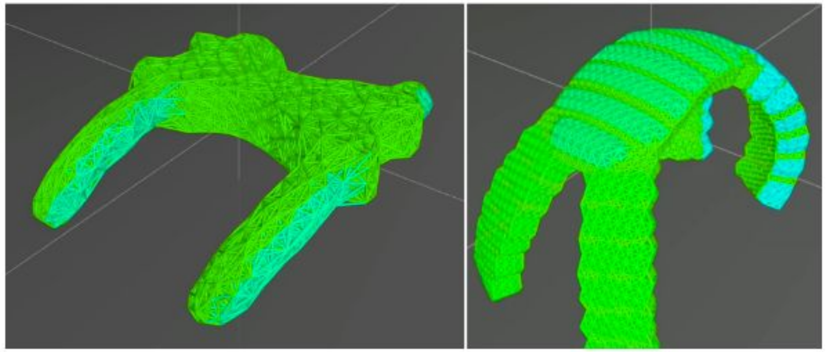


Fig. 1. Two four-legged soft robot simulated in Titan with hundreds of thousands of independent internal joints.

Venue

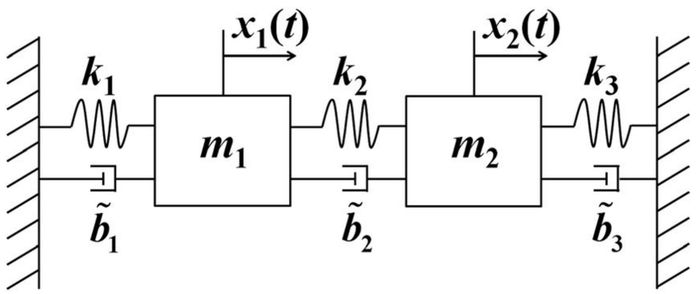
2020 ICRA (IEEE International Conference on Robotics and Automation)

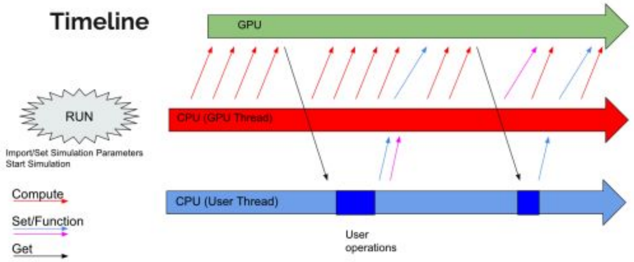
One-Sentence Summary

A GPU-accelerated simulation library for multiple interacting robot bodies (e.g. multi-agent robotics)

Titan Summary

Spring Kinematics



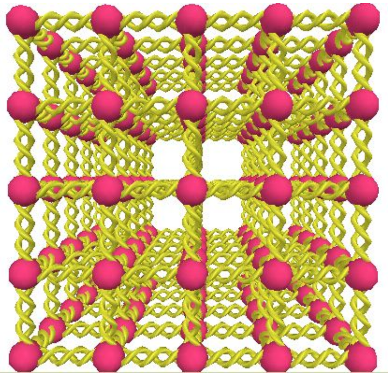


Breakpoints, Joins, and Events

Step-Based Integration

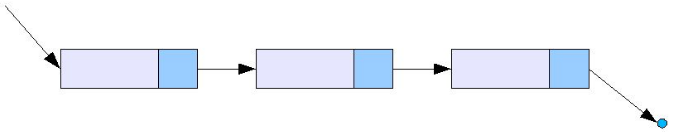
```
for i = 2:length(t)
    k1 = h * f(t(i - 1), w(i - 1, :));
    k2 = h * f(t(i - 1) + (h / 2), w(i - 1, :) + (k1 / 2));
    k3 = h * f(t(i - 1) + (h / 2), w(i - 1, :) + (k2 / 2));
    k4 = h * f(t(i - 1) + h, w(i - 1, :) + k3);
    w(i, :) = w(i - 1, :) + ((k1 + (2 * k2) + (2 * k3) + k4) /
6);
end

for i = 2:length(t)
    w(i, :) = w(i - 1, :) + h * f(t(i - 1), w(i - 1, :));
end
```



Lattice Structures

Parallelized Computation



O(1) Insertion and Deletion

Project Overview

Project Summary

Architecture - PikesPeak

1 x Intel 11900K, 1 x NVIDIA RTX 3080 Ti

Application - CUDA programming in Robotics simulation

Domain

1. **Soft Body Robotics**
2. **Multi Agent Robotics**
3. **Simulation**

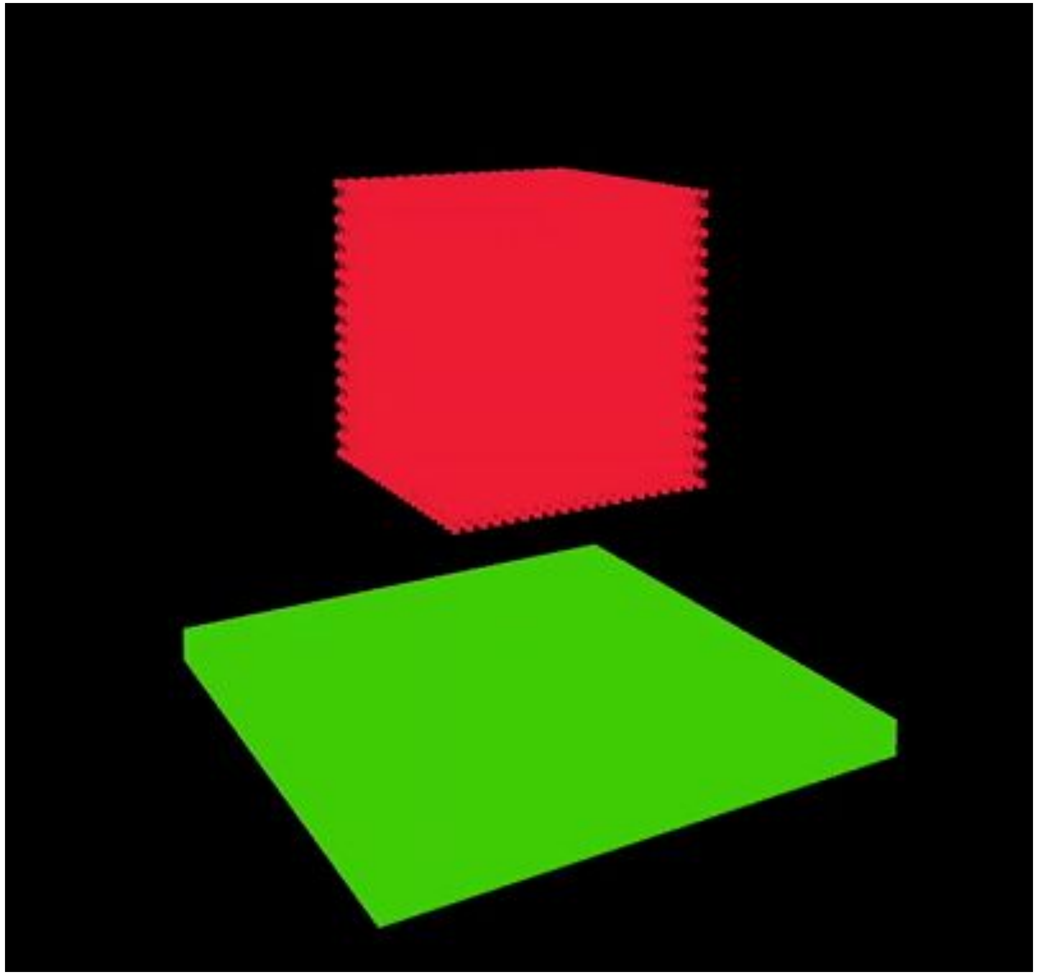
Key Point

Apply GPU acceleration to robotics simulation, where parallelization brings benefits.

GOAL

Accelerate the following kernels:
- massForcesAndUpdate
- computeSpringForces

What is being parallelized?
- alternate between spring and mass updates
- Spring
  - compute elastic force applied to masses (Hooke's Law)
  - atomically add the force(s) to the system
- Mass
  - apply all constraints/forces that are acting on the mass
  - update the kinematics of the mass using desired integration technique



computeSpringForces Kernel

```
__global__ void computeSpringForces(CUDA_SPRING ** d_spring, int num_springs, double t) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if ( i < num_springs ) {
        CUDA_SPRING & spring = *d_spring[i];

        if (spring._left == nullptr || spring._right == nullptr || ! spring._left -> valid || ! spring._right -> valid) // TODO might be expensive w
ith CUDA instruction set
            return;

        Vec temp = (spring._right -> pos) - (spring._left -> pos);

        double scale = 1.0;
        if (spring._type == ACTIVE_CONTRACT_THEN_EXPAND){
            scale = (1 - 0.2 * sin(spring._omega * t));
        } else if (spring._type == ACTIVE_EXPAND_THEN_CONTRACT){
            scale = (1 + 0.2 * sin(spring._omega * t));
            }

        Vec force = spring._k * (spring._rest * scale - temp.norm()) * (temp / temp.norm()); // normal spring force
        force += dot(spring._left -> vel - spring._right -> vel, temp / temp.norm()) * spring._damping * (temp / temp.norm()); // damping

#ifdef CONSTRAINTS
        if (spring._right -> constraints.fixed == false) {
            spring._right->force.atomicVecAdd(force); // need atomics here
        }
        if (spring._left -> constraints.fixed == false) {
            spring._left->force.atomicVecAdd(-force);
        }
#else
        spring._right -> force.atomicVecAdd(force);
        spring._left -> force.atomicVecAdd(-force);
#endif

    }
```

massForcesAndUpdate Kernel

```
__global__ void massForcesAndUpdate(CUDA_MASS ** d_mass, int num_masses, double dt, double T, Vec global_acc, CUDA_GLOBAL_CONSTRAINTS c) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < num_masses) {
        CUDA_MASS &mass = *d_mass[i];

#ifdef CONSTRAINTS
        if (mass.constraints.fixed == 1)
            return;
#endif

        mass.force += mass.m * global_acc;
        mass.force += mass.extern_force;

        // mass.force += mass.external;

        for (int j = 0; j < c.num_planes; j++) { // global constraints
            c.d_planes[j].applyForce(&mass);
        }

        for (int j = 0; j < c.num_balls; j++) {
            c.d_balls[j].applyForce(&mass);
        }

#ifdef CONSTRAINTS
        for (int j = 0; j < mass.constraints.num_contact_planes; j++) { // local constraints
            mass.constraints.contact_plane[j].applyForce(&mass);
        }

        for (int j = 0; j < mass.constraints.num_balls; j++) {
            mass.constraints.ball[j].applyForce(&mass);
        }

        for (int j = 0; j < mass.constraints.num_constraint_planes; j++) {
            mass.constraints.constraint_plane[j].applyForce(&mass);
        }

        for (int j = 0; j < mass.constraints.num_directions; j++) {
            mass.constraints.direction[j].applyForce(&mass);
        }

        // NOTE TODO this is really janky. On certain platforms, the following code causes excessive memory usage on the GPU.
        if (mass.vel.norm() != 0.0) {
            double norm = mass.vel.norm();
            mass.force += - mass.constraints.drag_coefficient * pow(norm, 2) * mass.vel / norm; // drag
        }
#endif
```

```
#ifdef RK2
        if constexpr(step) {
            mass.acc = mass.force / mass.m;
            mass.__rk2_backup_vel = mass.vel;
            mass.__rk2_backup_pos = mass.pos;

            mass.pos = mass.pos + 0.5 * mass.vel * dt;
            mass.vel = mass.vel + 0.5 * mass.acc * dt;
            mass.T += 0.5 * dt;
        } else {
            mass.acc = mass.force / mass.m;
            mass.pos = mass.__rk2_backup_pos + mass.vel * dt;
            mass.vel = mass.__rk2_backup_vel + mass.acc * dt;
            mass.T += 0.5 * dt;
        }
#elif VERLET
        mass.vel += 0.5 * (mass.acc + mass.force / mass.m) * dt;
        mass.acc = mass.force / mass.m;
        mass.pos += mass.vel * dt + 0.5 * mass.acc * pow(dt, 2);
        mass.T += dt;
#else // simple leapfrog Euler integration
        mass.acc = mass.force / mass.m;
        mass.vel = mass.vel + mass.acc * dt;
        mass.pos = mass.pos + mass.vel * dt;
        mass.T += dt;
#endif
        mass.force = Vec(0, 0, 0);
    }
}
```

Acceleration Details via CUDA

How does the Titan library work?

- **CUDA!**

Memory

ie: cudaMalloc, cudaMemcpy, cudaFree

- Done mostly with OOP
- Getters/setters
  - Turn springs/masses into usable data
- Constructors/Destructors
  - Getting simulation environment ready/removed

Kernels

ie:
```
computeSpringForces
<<<springBlocksPerGrid,
THREADS_PER_BLOCK>>>
massForcesAndUpdate
<<<massBlocksPerGrid,
THREADS_PER_BLOCK>>>
```

- Done in simulation class execute method
  - While true {alternate between spring and mass operations}

Sync

ie: cudaDeviceSynchronize

- Mostly in Simulation::execute while true loop
  - Before doing anything
    - Make sure that last time step values are done
  - Right before kernel calls
    - Have all other stuff (ie: graphics)taken care of before kernel calls

Technical Issues

- Istanbul Setup Issues
  - One core buggy/broken
  - nvcc unavailable
  - Legacy errors when starting Docker container w/ GPUs
- Lack of Docker Experience
  - Needed refresher on what Docker is
  - How do we create a Dockerfile?
  - Docker commands and their options
- Using maximum Threads-per-block count in kernel calls caused GPU error
- Solutions
  - Assistance from Prof. Mehmet and Justin
  - Moved to Pikespeak
  - Cursor/GPT :)
  - Group Documentation
  - vim ./include/Titan/sim.h
    - * #define THREADS_PER_BLOCK

```
#include <cuda_runtime.h>

int main() {
    int device_count;
    cudaGetDeviceCount(&device_count);

    if (device_count == 0) {
        std::cout << "No CUDA devices found." << std::endl;
        return 1;
    }

    for (int i = 0; i < device_count; i++) {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        std::cout << "GPU " << i << ": " << prop.name << std::endl;
    }
    return 0;
}
skannady@istanbul:~$ nvcc gpu_info.cu -o gpu_info

Command 'nvcc' not found, but can be installed with:

apt install nvidia-cuda-toolkit
Please ask your administrator.

skannady@istanbul:~$ docker run --gpus '"device=0,1,2"' -it titan-container
docker: Error response from daemon: failed to create task for container: failed to create shim task: OCI runtime create failed: runc create failed:
unable to start container process: error during container init: error running hook #0: error running hook: exit status 1, stdout: , stderr: Auto-det
ected mode as 'legacy'
nvidia-container-cli: detection error: nvml error: unknown error: unknown.
skannady@istanbul:~$ docker run --gpus device=GPU-eceee751-57a0-47cc-0b7e-7adc963a9bf5 -it titan-container
docker: Error response from daemon: failed to create task for container: failed to create shim task: OCI runtime create failed: runc create failed:
unable to start container process: error during container init: error running hook #0: error running hook: exit status 1, stdout: , stderr: Auto-det
ected mode as 'legacy'
nvidia-container-cli: detection error: nvml error: unknown error: unknown.
```

```
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1586
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1589
GPUassert: 701 too many resources requested for launch /Titan/src/sim.cu 1569
GPUassert: 701 too many resources reques
```

Experiment

Platform

- Simulations on the Titan library
- Accelerator: Pikespeak
- GPU: 1x NVIDIA RTX 3080 Ti

Goal: Measure the timing differences between different TPB counts while running the default unit tests that Titan provides. Understand how the THREADS_PER_BLOCK macro impacts performance for the unit test suite. We will run the application with THREADS_PER_BLOCK values in the set {32, 64, 128, 192, 256, 512}.

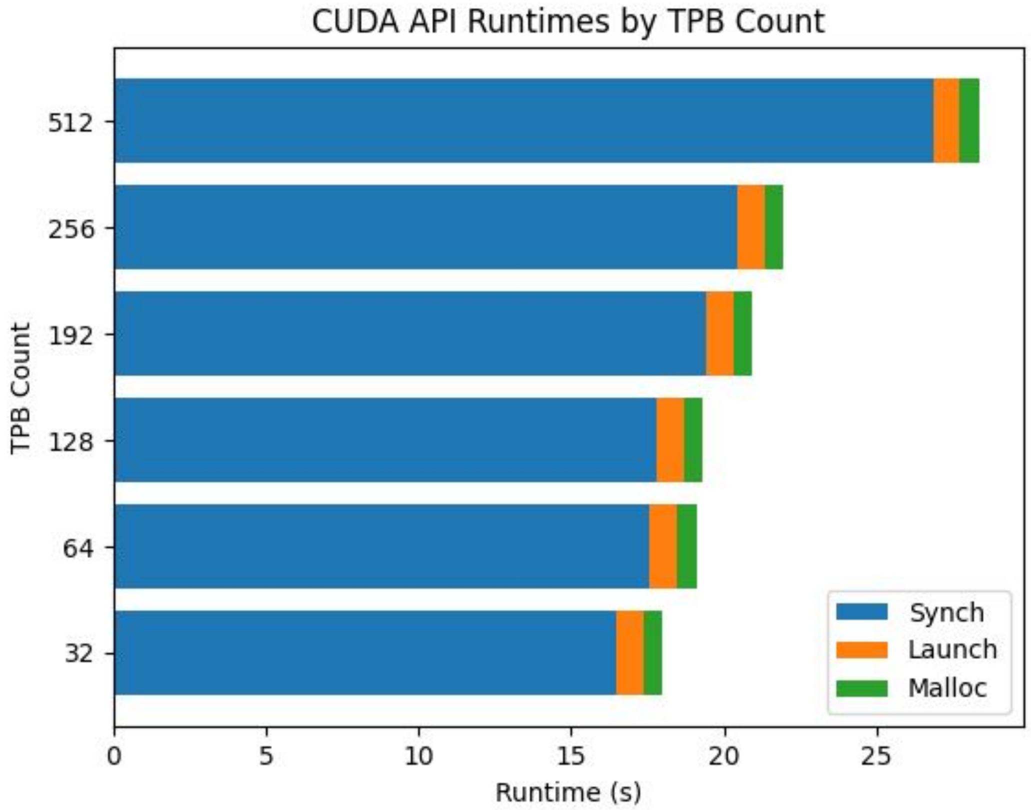
Procedure:

1. vim ./include/Titan/sim.h, change #define THREADS_PER_BLOCK ??? line to a different TPB count
2. Recompile, ./clean-build.sh debug -DTITAN_BUILD_TESTS=ON
3. Run nsys profile ./build/debug/test/physics_unittest -i 10 to profile things like kernel activity, memory transfers, and API/OS call runtimes
4. Run nsys stats {report_name} to view how the different THREADS_PER_BLOCK value impacts performance across the board.
5. Repeat for all TPB values

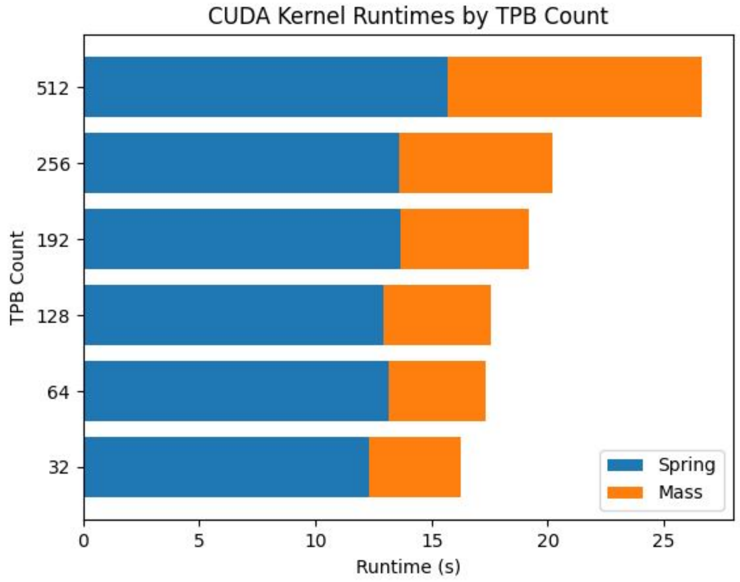
Results

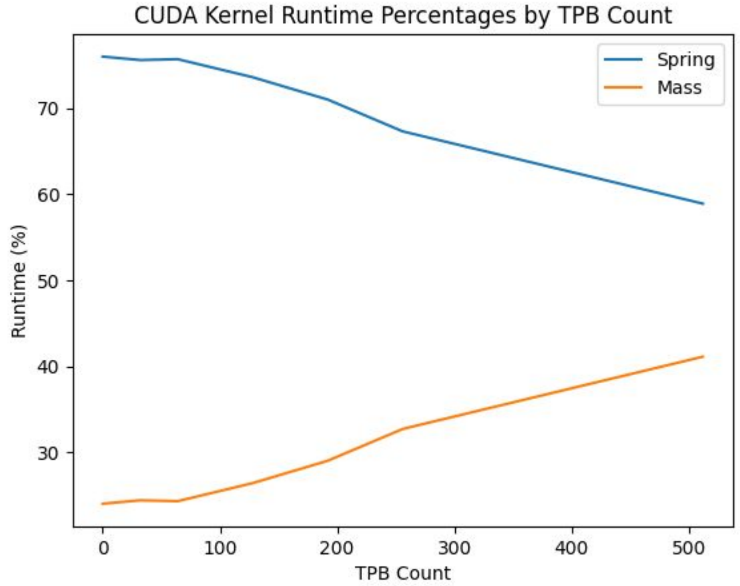
Insights

- OS Runtime, MemCpy results remain consistent across TPB counts
  - Not surprising, functionality from the host's POV shouldn't significantly change
  - Similar Info sent to GPU, CPU regardless of TPB count
- CUDA API Runtimes vary significantly more
  - Makes sense, waiting for more threads to synchronize will naturally take longer
  - Synchronization is necessary so that we don't move to the next time step before the current one is complete
  - The pattern of having springs apply forces, join, forcing masses to update once the forces are calculated, join, repeat, explains a lot



Results (cont)





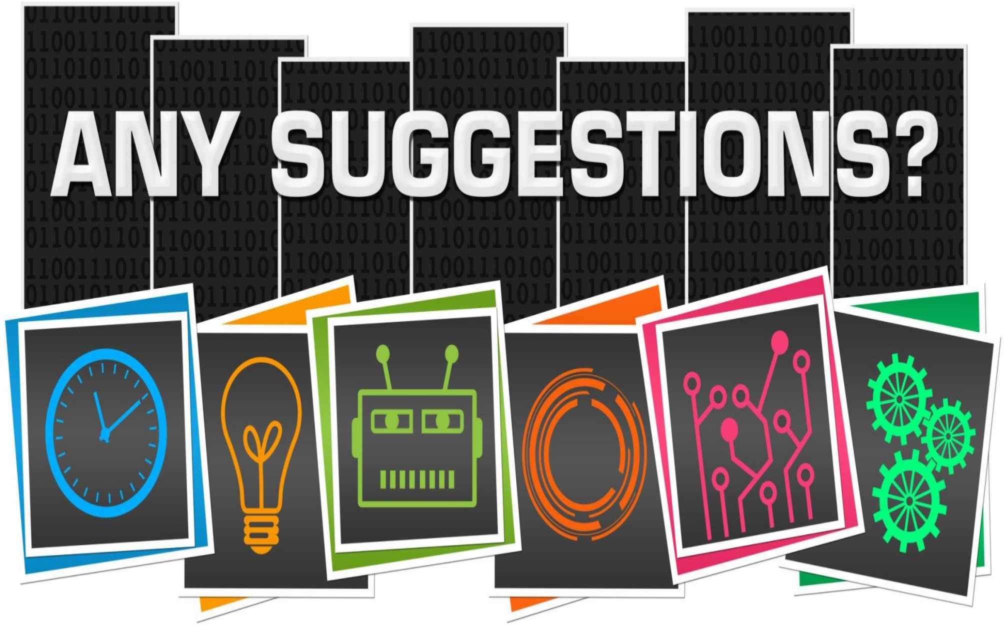
Insights

- Kernel Runtime Summary shows significant differences between TPB counts
  - Higher TPB -> Higher Runtime
    - Synchronization increases runtime
    - Memory = bottleneck (need to update all springs and all masses for all time steps)
  - High TPB -> High Mass:Spring Runtime Ratio
    - Mass kernel benefits less than Spring kernel from having more computational power
      - All of those FOR-loops running serially in mass kernel = more runtime
      - Spring kernel is just a $F = k\Delta x$
    - Spring count >> mass count in test suite, so % higher for low TPB
- Our speculation: mass converges to 100% as TPB approaches ∞, but can't prove it experimentally w/ our setup

What next?

Potential Ideas

1. Compare execution of unit test suite between accelerators
   a. Would need to find another usable accelerator
2. Vary spring or mass counts on a chosen test case, observe changes in execution speed
   a. Some intuition gained on spring vs mass kernel behavior from existing experiment, but more could be learned if we make them independent variables in an experiment
3. Run the exact same experiment that we ran, but make power/energy the dependent variable
   a. Efficiency is crucial for robotic applications, so this information would be valuable
4. Suggestions are appreciated!



Reflections/Takeaways

Things We Learned/Liked

- Higher TPB, more synchronization needed among threads (time spent waiting)
- Higher computational space does not always equal better performance

Disliked

- Mass calculations use default Euler integration - already claimed to have better performance than RK4 integration, yet still very serial
- Authors claim that the framework favors parallelism, sacrifices accuracy. But, mass calculations still cannot take full advantage of increasing parallelism
- SETUP!!!

Room for Improvement

- To have at least one other experiment with a different independent variable
- To compare performance across different GPUs

COLORADO SCHOOL OF MINES

Questions?

Sources

Austin, Jacob, et al. "Titan: A parallel asynchronous library for multi-agent and soft-body robotics using nvidia cuda." *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020.