Virtual Research Presentation Conference

GPU Computing for Fast, Massively Parallel Simulations of Manufacturing Processes

Principal Investigator: Richard Otis (357)
Co-Is: Hamsa Shwetha Venkataram (1740)
Program: Spontaneous Concept

RPC-127

Jet Propulsion Laboratory
California Institute of Technology

Tutorial Introduction

Abstract

JPL is increasingly relying on manufacturing simulation software, including for additive manufacturing process design. As our manufacturing simulation capability scales up to whole parts, the demand for parallel processing will increase faster than what our present algorithms can support. The objective of this work was to explore moving certain types of engineering computations to the GPU, for the purpose of performing billions of computations in parallel with the potential upside being multiple orders-of-magnitude improvement in calculation times.

Empirical evidence was found during this study that the orders-of-magnitude (**100x+**) speedups enabled by GPUs, reported in the literature, are possible to achieve on problems relevant to JPL and NASA. The speedups enabled by GPU-aware simulation codes may constitute a game-changing technology opportunity for certain classes of physical simulation problems.

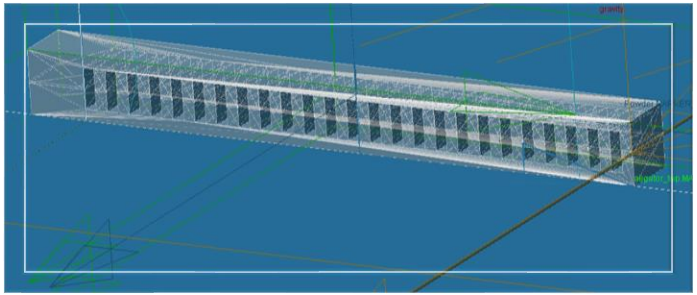
Problem Description

JPL is experiencing computational “growing pains” relating to our increasing infusion of advanced manufacturing technology, particularly 3D printing. Internal customers are demanding results from more advanced manufacturing processes with less iteration time.

SOA process modeling codes do not provide the just-in-time accuracy needed to deliver quick-turn solutions, resulting in time-consuming experimental trials.

GPU-based algorithms have been reported as an approach for achieving orders-of-magnitude speedups in performance for certain classes of problems. Within the last 12 months the software tools for programming GPUs have matured significantly in terms of accessibility to non-experts.

If the reported speedups can be replicated in modeling codes used by NASA and JPL, this would mean running high-fidelity process models, which currently take *weeks* or *months*, in a matter of hours.



Dr. Cornelia Altenbuchner/JPL/CIF

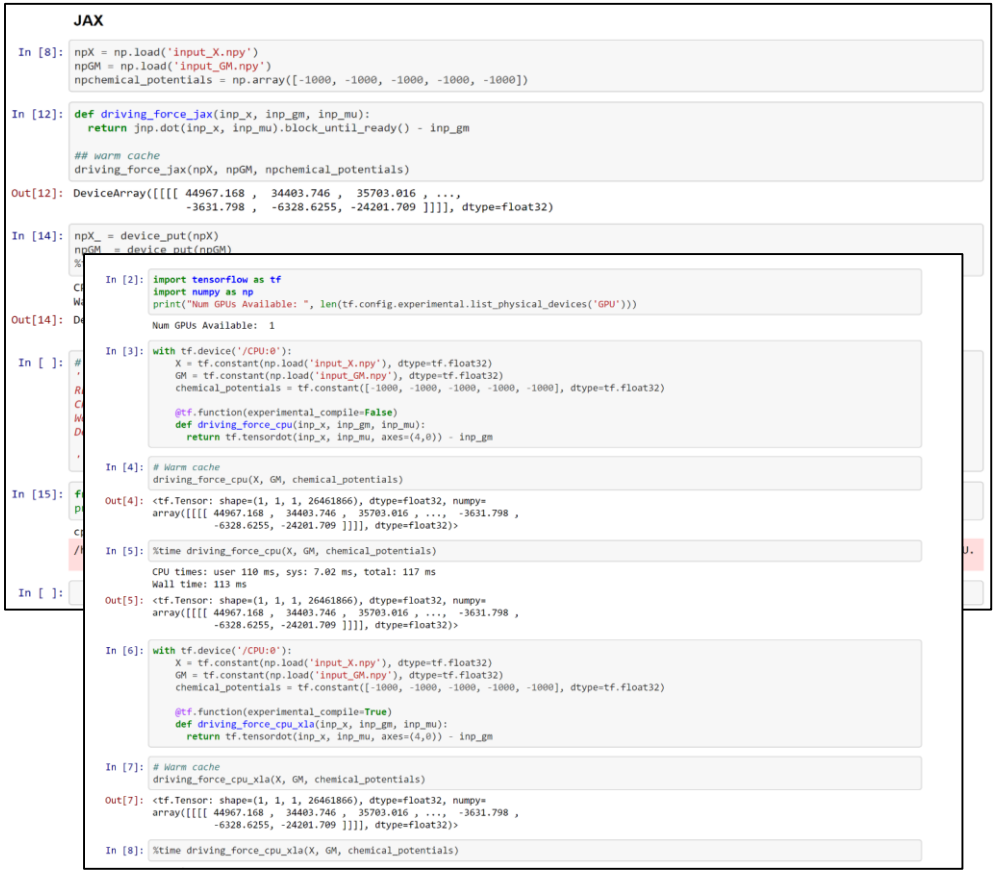
```
# Each thread computes one element in the result matrix.
# The dot product is chunked into dot products of TPB-long vectors.
tmp = 0.
for i in range(bpg):
    # Preload data into shared memory
    sA[tx, ty] = A[x, ty + i * TPB]
    sB[tx, ty] = B[tx + i * TPB, y]

    # Wait until all threads finish preloading
    cuda.syncthreads()

    # Computes partial product on the shared memory
    for j in range(TPB):
        tmp += sA[tx, j] * sB[j, ty]

    # Wait until all threads finish computing
    cuda.syncthreads()
```

Methodology

Several GPU-enabled array computing packages were considered during this study, including:

- Tensorflow: https://tensorflow.org/
- JAX: https://jax.readthedocs.io/en/latest/
- Numba: http://numba.pydata.org/
- CuPy: https://cupy.dev/
- Scikit-cuda: https://scikit-cuda.readthedocs.io/en/latest/
- CUDA.jl: https://juliagpu.org/

Array arithmetic (addition, multiplication) and linear algebra capabilities such as matrix inversion were also investigated for each of the packages.



Results

Goals: Demonstrate 100x+ speedup from GPU on a JPL-relevant simulation problem

Accomplishments

- Demonstrated **3-4x** speedups from GPU at JPL
- Observed **500x** speedup due to GPU by a NASA-funded University team for a relevant simulation

Next Steps: Pursue the solution of a more focused problem using the Numba toolkit to achieve the observed speedups in a “real world” scenario.

The Julia programming language is also an interesting option for new simulation software packages.

```
function hyperplane(compositions::MatrixType,
    energies::FloatVector, target_composition::FloatVector,
    chemical_potentials::FloatVector, total_moles::FloatType,
    fixed_chempot_indices::IndexVector, fixed_comp_indices::IndexVector,
    result_fractions::FloatVector, result_simplex::IndexVector)
    num_components = size(compositions)[2]
    num_points = length(energies)
    num_fixed_chempots = size(fixed_chempot_indices)[1]
    simplex_size = num_components - num_fixed_chempots
    # composition index of -1 indicates total number of moles, i.e., N=1 condition
    included_composition_indices = fixed_comp_indices
    best_guess_simplex = sort(setdiff!(collect(1:num_components), fixed_chempot_indices))
    free_chempot_indices = best_guess_simplex[:]
    candidate_simplex = best_guess_simplex[:]
    trial_simplices = Array{IntType, 2}(undef, simplex_size, simplex_size)
    fractions = Array{FloatType, 2}(undef, simplex_size, simplex_size)
    driving_forces = Array{FloatType, 1}(undef, num_points)
    for i in 1:simplex_size
        trial_simplices[i, :] = best_guess_simplex
    end
    trial_matrix = Array{FloatType, 3}(undef, simplex_size, simplex_size, simplex_size)
    candidate_tieline = Array{FloatType, 2}(undef, simplex_size, simplex_size)
    candidate_energies = Array{FloatType, 1}(undef, simplex_size)
    candidate_potentials = Array{FloatType, 1}(undef, simplex_size)
    smallest_fractions = Array{FloatType, 1}(undef, simplex_size)
    saved_trial = 0

    max_iterations = 1000
    for _ in 1:max_iterations
        for trial_idx in 1:simplex_size
            for comp_idx in 1:simplex_size
                ici = included_composition_indices[comp_idx]
                for simplex_idx in 1:simplex_size
                    if ici > 0
                        trial_matrix[comp_idx, simplex_idx, trial_idx] = compositions[trial_simplices[trial_idx, simplex_idx], ici]
                    else
                        # ici = -1, refers to N=1 condition
                        trial_matrix[comp_idx, simplex_idx, trial_idx] = 1 # 1 mole-formula per formula unit of a phase
                    end # if
                end # for
            end # for
        end # for
```

Publications and References

https://rapids.ai/

https://cloud.google.com/tpu/docs/tpus

https://hackage.haskell.org/package/accelerate

http://numba.pydata.org/

https://scikit-cuda.readthedocs.io/en/latest/

https://github.com/AdditiveModeling/pyphasefield/blob/a089672a9659b36a888f2be22c913d2235bd3a30/pyphasefield/pyphasefield/Engines/NCGPU.py