SCIENTIFIC COMPUTING FINAL PROJECT REPORT

DAVID SUH
UNIVERSITY OF CHICAGO

Abstract. **Executive Summary:** SPH (Smoothed-particle Hydrodynamics) is a common method of simulating fluid movement, such as water. It is often more computationally efficient than Euclidean (grid) methods due to nearest-neighbor computations and simplification of the movement equations. The formulas that arise for SPH are ripe for paralleization and vectorization. Thus, it is of interest if the JAX and pytorch frameworks for python can be utilized to improve simulation performance. We implement a from-scratch version of SPH following the original paper by Mueller from 2003, which can take numpy, JAX, and pytorch as computational backends and perform benchmarking on a Apple Silicon M2, Nvidia V100 GPU, and Google TPUv4. Results show that contrary to projections, JAX was not very performant on the GPU (even worse than its CPU performance), while its TPU performance broken even with CPU. It was the best performer on the CPU, however. Pytorch on the GPU was extremely performant and showed advantages of framework maturity in the form of computational optimizations. Numpy as of course the slowest. These results point us to further research into what made JAX more performant on the CPU and perhaps implementing these optimizations piecewise into existing systems. It also further opens the gate for fluid simulations in e.g. game consoles and other edge devices that do not have access to strong GPU hardware.

1. Introduction

When it comes to simulating fluids, there are two main camps of methods to consider. There is the (more traditional) Euclidean method, in which the vector field for fluid movement is evaluated on a grid, with increasing accuracy by taking a denser lattice. The other, more recent method is Smoothed-Particle Hydrodynamics (SPH). Introduced in (Muller et al. 2003), in this method, we approximate the movement of particles via a system of n particles, with locations that do not have a particle gaining estimate scalar/vector values as a weighted sum of nearby particles. In particular, the scalar value $s_{\mathbf{r}}$ of a specific location $\mathbf{r} \in \mathbb{R}^n$ can be estimated as

$$s_{\mathbf{r}} = \sum_j (m_j/\rho_j) s_j W(\mathbf{r} - \mathbf{r}_j, h)$$

where W, the *smoothing kernel*, determines the weighting of the particle based on distance, and m_j, ρ_j being the mass and density of particle j. In the particular case of SPH, we use the above weighted sum method to calculate the density and various forces (pressure, viscosity, gravity) that are acting on particles to simulate the movement of fluids. It is immediately obvious that executing the above computations for a large number of particles is not only highly computationally complex, but also has a lot of potential for being parallelized/vectorized to improve performance and produce better results at a faster speed. Thus, there is strong motivation to experiment and explore with various computational options to see if we can achieve better speedups (especially in a modern context, where highly performant computational frameworks have become more common).

2. Background

There have been examples of using CUDA to implement SPH (Hérault, 2010)[1]. Indeed, one can expect any modern fluid simulator to be making good use of the GPU. In this experiment, I plan to not only attempt to recreate this results with a SPH implementation from scratch using Python, but also test the performance of SPH simulations using JAX (Google's XLA accelerated computing framework) as the backend. There haven't been (to my knowledge as of Decemeber, 2023), any papers that deal with specifically SPH simulations in JAX (though there have been some adjcacent problems), in particular leveraging TPUs (Tensor Processing Units), which is specific hardware developed by Google for Deep Learning and for which JAX is engineered to operate well with. Furthermore, a pytorch implementation of SPH should also be interesting, seeing as pytorch is a very recent framework. Thus, it is also an interesting question if computational programs coded in Python can be performant due to these new frameworks.

Date: December, 2023.

3. Methdology/Approach

The methodology is very simple. We implement a SPH simulation as per (Mueller, 2003)[2]. In this implementation, we do not include the surface tension forces that were included in the original paper due to time constraints. We have designed in the simulation such that we can use three different computational backends: `numpy, jax, torch`. Using this variation, we can benchmark the performance of the SPH simulation on different systems. It should be noted that unlike most SPH implementations, we do not use any QuadTree/KDTree or any other form of nearest-neighbor calculations in for calculating the values for each particle – this is principally because of JAX's requirement that matrix sizes be known at compile time – this makes it difficult to implement data dependent operations such as nearest-neighbor search (the array of neighbors can be varying sizes). The algorithm closely tracks that of (Mueller, 2003), thus is omitted in the report for the sake of brevity.
Since we are able to choose the computational backend, we can now benchmark the performance of each framework. We benchmark the program like so: we consider a number of particles to simulate and run the simulation. We use each framework and calculate the average time that it takes for the program to run 1000 steps using python's `time` module, and we consider 2 trials for each run. We chose n values of $100, 500, 1000$. We run these experiments on three different types of hardware: A Macbook Pro M2 (which uses Apple silicon M2/ARM), a V100 Nvidia GPU through Google Colab, and a TPUv4 through Kaggle Notebooks.

4. Results

We summarize the benchmarking results in Figure 1. We can observe that there is a clear trend of numpy being the

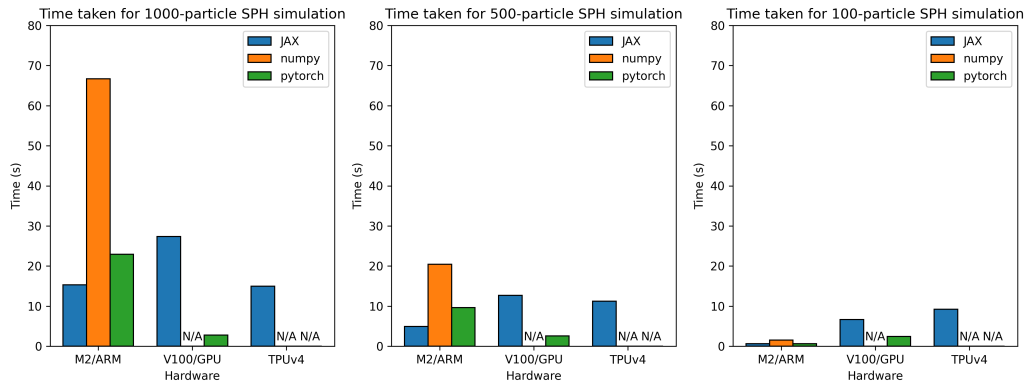

Figure 1. Benchmarking results for n-particle SPH simulations on various hardware, $n = 100, 500, 1000$. We can note that N/A is noted if there is no point benchmarking for that hardware (e.g. numpy does not use GPU and thus its benchmark has no meaning for the V100)

slowest computational framework in every benchmark. This was to be expected since we included numpy as a baseline comparison. We can note some interesting results, however.

4.1. **JAX performance is CPU $=$ TPU $>$ GPU.** This result is quite surprising and contrary to my initial estimate, as one would expect the GPU hardware to greatly accelerate matrix multiplication and other vector operations. However, we can consider a couple reasons behind this lag. One could be that the overhead of JAX's GPU implementation overshadows its gains – this is especially since JAX has compile time costs associatedf with it, which may further slow down on more complex hardware such as GPUs. Another reason could be that my programming of JAX was not perfect since I am also relatively new to the feature set. It is also the case that JAX may not be as efficient in utilizing the CUDA kernels as pytorch, which may be what results in its worsened performance. We can also note that JAX gains a lot of performance gains from operation fusing; this is difficult on a GPU, since its operational libraries is closed source. It is also interesting that the TPU did not accelerate the performance as well; it can be suggested that this perhaps for the same reasons as the GPU.

4.2. **JAX outperforms on CPU.** It can also be seen that JAX outperforms the other two frameworks when run on a CPU. In particular, it is able to beat out pytorch for better performance on the CPU.

4.3. **GPUs still reign supreme with pytorch.** It is clear that pytorch+GPU was the dominant performer in the benchmarks. In fact, the benchmark speed barely changed between $n = 100$ and $n = 1000$, suggesting that we haven't even begun to observe its true computational potential. It outperforms JAX on the GPU by a factor of about 10.

5. Discussion

These results are interesting and help the field because it is always important to survey new computational methods in case they yield a method for better performance. In our case, we were able to show that JAX might not be the best candidate for CUDA/GPU driven fluid simulation computation, perhaps due its youth and thus unoptimized GPU utilization. However, I think that JAX's CPU performance is still noteworthy, and is something that could be inspected further. It could be of particular interest for fluid simulations in edge/less performant devices in applications such as videogames. This also further exposes the importance of good coding practices when it comes to computation – simply having the hardware is not enough. It is clear that had I been a much more seasoned JAX programmer, the performance metrics would have been different. My program in general is not nearly as performant as existing solutions – these solutions can often render 50,000+ particles at 60fps; my program struggles even with just 2000 particles.
If I had more time, I would definitely reorganize the computational system that I had set up for my implementation. In particular, I have some ineffeciencies stemming from having to convert from torch Tensors/JAX arrays to numpy arrays for some of my computations, which introduced some overhead that could have affected the results. I would have also spent more time on performance profiling to relieve bottlenecks so that the performance of each framework could be assessed more fairly and undisturbed by coding issues. I would have also tried to control more constants in my simulation to achieve more realisitic fluid simulation movement. Finally, I would have tried to use a spatial hashmap or other sorting methods to create an efficient neighbor query method (like most SPH simulation) to see if I could get better performance.

References

[1] Giuseppe Bilotta Alexis Hérault and Robert A. Dalrymple. Sph on gpu with cuda. *Journal of Hydraulic Research*, 48(sup1):74–79, 2010.
[2] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, page 154–159, Goslar, DEU, 2003. Eurographics Association.