**Noname manuscript No.**
(will be inserted by the editor)

# A Case Study on Pytorch based CFD simulation

**Yunyan Ding · Aiichiro Nakano**

Received: date / Accepted: date

**Abstract** Graphics Processing Units (GPUs), which were originally developed for games and graphics, are widely used in high-performance computing (HPC) and scientific applications because of the strong floating-point calculation capability and the advantages in parallelism. Compute Unified Device Architecture (CUDA) is utilized as a parallel computing platform and programming model for GPUs to reduce the complexity of programming. Python is a popular programming language which is widely used in Machine Learning with many powerful libraries. However, Python is not a common choice for Computational Fluid Mechanics (CFD) simulation and visualization. To combine the merits of Python and CUDA, we inspired the idea to apply machine learning on CFD simulation and visualization. Here, we introduce Lettuce, which is a new lattice Boltzmann method (LBM) framework that enables both CPUs and CUDA. It is based on the machine learning framework PyTorch. In this paper, we will compare CPUs and GPUs efficiency on Lettuce based CFD simulation in 2D and 3D, and analyze how each factor influence the efficiency.

**Keywords** CUDA · Lettuce · PyTorch

**Mathematics Subject Classification (2020)** MSC code1 · MSC code2 · more

F. Author
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

## 1 Introduction

PyTorch is a Python library that performs immediate execution of dynamic tensor computations with GPUs acceleration, which can also maintain performance comparable to the fastest current libraries for deep learningPaszke et al. (2019). To implement CFD by PyTorch, we introduced Lettuce, a PyTorch based and LBM based framework. Because of its characteristics, Lettuce can accelerate calculations with minimal source code, with a neural collision model and combining machine learning. It also combines the benefits of PyTorch's automatic differentiation framework in flow control and optimizationBedrunka et al. (2021). An example of flow visualization based on Lettuce is presented in **Figure 1** Due to the lack of the research on Python-based CFD, we are curious is Python able to do the simulation and visualization effectively and how is the time efficiency of running CFD with Python. To solve these questions comprehensively, we concluded the following research questions (RQ): **RQ1: What is the most time consuming composition of CPUs and GPUs on the simulation? RQ2: How is the efficiency of CPUs and GPUs on the simulation, which one is better? RQ3: Which factors can largely affect the efficiency?**
To answer these questions and simplify our tasks, the flow module Obstacle2D was chosen. According to Lettuce documentation, Obstacle2D is a flow class to simulate the flow around an object (mask) in 2D, which consists of one inflow (equilibrium boundary) and one outflow (anti-bounce-back-boundary), leading to a flow in positive x direction Bedrunka et al. (2021).

## 2 Research Preparation

### 2.1 Device

The device that we used is: GTX1070 (GPUs) and Intel(R) i7 (CPU).

### 2.2 Data Preparation

In order to have a straightforward view to our research, for RQ1 and RQ2, we have three input data Obstacles_1, Obstacles_2 and Obstacles_3. All of them are in the same dimension, which is $500\times300$.
In Obstacles_1, it contains a circle centered at (200,200) with radius 10, and a polygon with the following position (70,70),(70,90),(90,90),(90,70),(80,60). For Obstacles_2, including all the obstacles in Obstacles_1, it contains one more square, with its center located at (105,105) and width 10. For Obstacles_3, it contains all the obstacles in Obstacles_2 and one new rectangle located at (400,100), in a $20\times5$ dimension.
All of these three data have the same shape, but with different obstacles area in total, with following relation: Obscales_area(Obstacles_1) < Obscales_area(Obstacles_2)

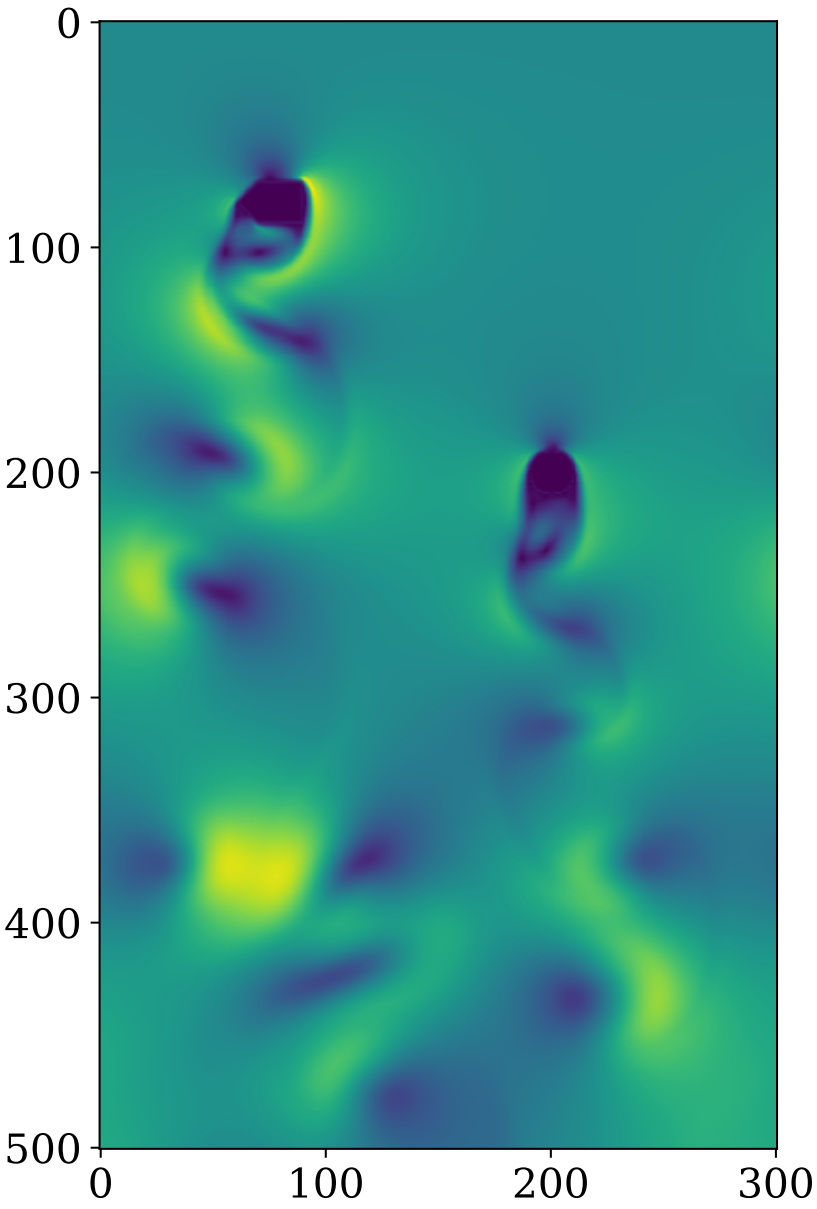


**Fig. 1** An example of flow visualization by Lettuce

< Obscales_area(Obstacles_3).
In order to answer RQ3, 4 more input data will be introduced. Obstacles_4, Obstacles_5, Obstacles_6, Obstacles_7 have a dimension 750×300, 1000×300, 1250×300 and 1500×300, respectively. All of them have the same obstacles information as Obstacles_1.

## 3 Simulation and Visualization Result

### 3.1 RQ1: What is the most time consuming composition of CPUs and GPUs in the simulation?

**Approach:** To answer this question, we generated the profiling file by using cProfile, which is a Python module that provides statistics for accumulated duration and number of invocations for all parts of the programWagner et al. (2017). In order to reduce the noisy factors, such as the running environment, we run each input data three times, where **Table 1** presents the results from CPUs and **Table 2** presents the results from GPUs. To have a more straightforward view, we also identified the time and percentage for each function calls.

**Result:** According to the profiling files from cProfile, we plotted the pie chart to have a better view of the composition. Because there are more than eight millions of function calls each time, we only show the top 10 function calls with longest total time of Obstacle_1, which are presented in **Table 3** (CPUs) and **Table 4** (GPUs). The pie chart with percentage data are presented in **Figure 2** for CPUs and **Figure 3** for GPUs. CPUs spent 34.22% of total time on lettuce/equilibrium, which is the highest one. The second highest time consuming process is lettuce collision, following by method where, einsum, Lettuce/simulation, method sum, Lettuce/boundary, method pow, tensordot and Leetuce/lattices in an decreasing order. For GPUs, the top 3 time consuming processes are Lettuce/boundary, method einsum and Lettuce/equilibrium, which is 64.47%, 6.85% and 5.40%, respectively. By comparing these 20 function calls, we can see that both CPUs and GPUs spent many times on "method tensordot", "lettuce/boundary", "method pow", "lettuce/equilibrium", "method einsum" and "method where". For the CPUs, the highest time spent is the equilibrium, which is the moment equilibrium calculation process in Lettuce and it took 126 seconds. Within all the top 10 function calls, half of them are Lettuce functionalies and half of them are build-in methods from Numpy and PyTorch. However, for the GPUs, 3 of the 10 function calls are Lettuce functionalities, 1 of them is the Visualization Toolkit (VTK) for image processing and the rest of them are build-in methods. Lettuce equilibrium, the most time consuming invocations for CPUs, took only 2 seconds for GPUs, which is only about 1.66% of that of CPUs. By checking out the source code, equilibrium mainly contains calculations. According to the documentation of Lettuce, it does the same as the normal equilibrium. It uses around 20% less RAM, but runs about 2% slower on GPU and 11% on CPU. Moreover, the GPUs running time on Numpy method einsum, where, tensordot and pow are 7.34%, 3.21%, 8.11% and 8.79% of CPUs, respectively. Only the GPUs running time on the boundary from Lettuce is higher than that of CPUs, which is 1.68 times that of CPUs. By looking at the boundary part, we conjecture that this might be because of the architecture of CPUs and GPUs. Boundary function doesn't contain many high-dimensional computations. Instead, it is used to define the boundary setting and boundary conditions for

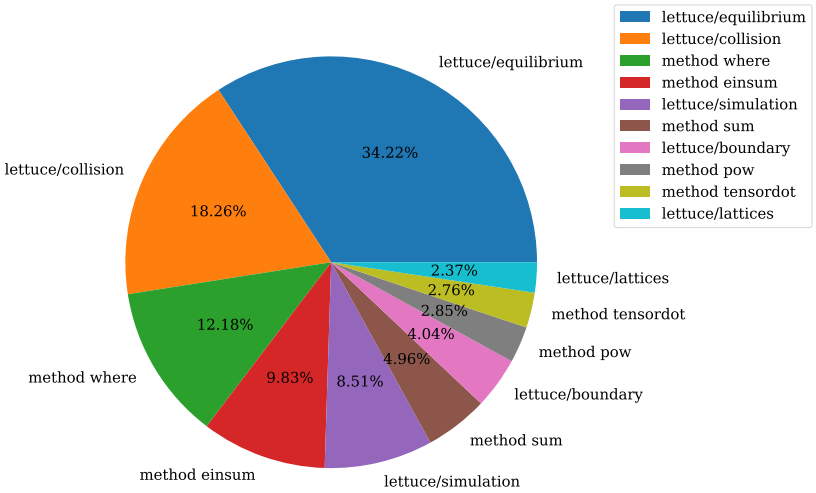


**Fig. 2** Percentage of top 10 components of total running time on CPUs

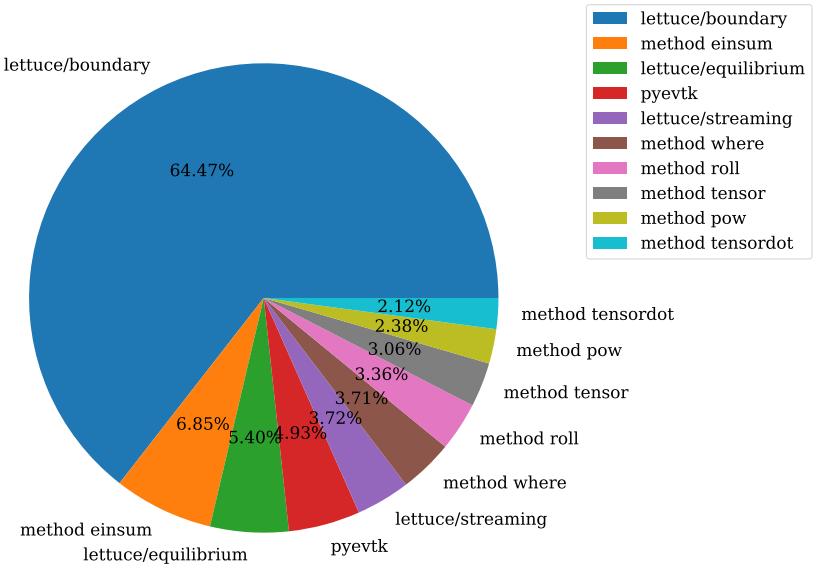


**Fig. 3** Percentage of top 10 components of total running time on GPUs

the simulation, which is more suitable for CPUs.

**Table 1** CPUs Running time Information in 2D

| | Obstacles_1 | Obstacles_2 | Obstacles_3 |
|---|---|---|---|
| Total Time 1(s) | 410 | 403 | 393 |
| Total Time 2(s) | 406 | 395 | 394 |
| Total Time 3(s) | 408 | 394 | 396 |
| MLUPS 1 | 3.7 | 3.8 | 3.9 |
| MLUPS 2 | 3.8 | 3.9 | 3.9 |
| MLUPS 3 | 3.7 | 3.9 | 3.8 |
| Function Calls 1 | 8,644,484 | 8,644,369 | 8,644,376 |
| Function Calls 2 | 8,644,341 | 8,644,512 | 8,644,376 |
| Function Calls 3 | 8,644,341 | 8,644,369 | 8,644,376 |
| Obstacles Area | 814 | 914 | 1,014 |

**Table 2** GPUs Running time Information in 2D

| | Obstacles_1 | Obstacles_2 | Obstacles_3 |
|---|---|---|---|
| Total Time 1(s) | 49 | 49 | 50 |
| Total Time 2(s) | 49 | 50 | 50 |
| Total Time 3(s) | 49 | 50 | 50 |
| Avg Total Time(s) | 49 | 50 | 50 |
| MLUPS 1 | 34.0 | 34.1 | 33.6 |
| MLUPS 2 | 34.1 | 33.7 | 33.6 |
| MLUPS 3 | 33.8 | 33.6 | 33.6 |
| Avg Total Time(s) | 49 | 50 | 50 |
| Function Calls 1 | 8,590,547 | 8,590,575 | 8,590,582 |
| Function Calls 2 | 8,590,547 | 8,590,575 | 8,590,582 |
| Function Calls 3 | 8,590,547 | 9,090,997 | 9,090,997 |
| Obstacles Area | 814 | 914 | 1,014 |

**Table 3** A composition of running profile on Obstacle1 by CPUs

| ncalls | tottime | percall | cumtime | percall | origin |
|---|---|---|---|---|---|
| 20002 | 125.595 | 0.006 | 162.459 | 0.008 | lettuce/equilibrium |
| 10000 | 67.037 | 0.007 | 246.456 | 0.025 | lettuce/collision |
| 110000 | 44.714 | 0.000 | 44.714 | 0.000 | method where |
| 80143 | 36.087 | 0.000 | 36.087 | 0.000 | method einsum |
| 1 | 31.236 | 31.236 | 405.910 | 405.910 | lettuce/simulation |
| 40239 | 18.219 | 0.000 | 18.219 | 0.000 | method sum |
| 10000 | 14.843 | 0.001 | 27.469 | 0.003 | lettuce/boundary |
| 100109 | 10.472 | 0.000 | 10.472 | 0.000 | method pow |
| 20002 | 10.136 | 0.001 | 10.136 | 0.001 | method tensordot |
| 20125 | 8.697 | 0.000 | 30.376 | 0.002 | lettuce/lattices |

### 3.2 RQ2: How is the efficiency of CPUs and GPUs on the simulation, which one is better?

**Approach:** To understand the efficiency between CPUs and GPUs, we generated the profile data for each input data. In order to reduce the environment's influence, we run all the data three times. Mainly, for each input data, we collect total running time, MLUPS (Million Lattice Updates Per Second), number

**Table 4** A composition of running profile on Obstacle1 by GPUs

| ncalls | tottime | percall | cumtime | percall | origin |
|---|---|---|---|---|---|
| 10000 | 24.945 | 0.002 | 27.124 | 0.003 | lettuce/boundary |
| 80143 | 2.650 | 0.000 | 2.650 | 0.000 | method einsum |
| 20002 | 2.088 | 0.000 | 5.527 | 0.000 | lettuce/equilibrium |
| 606 | 1.907 | 0.003 | 2.708 | 0.004 | pyevtk |
| 10000 | 1.440 | 0.000 | 4.435 | 0.000 | lettuce/streaming |
| 110000 | 1.436 | 0.000 | 1.436 | 0.000 | method where |
| 80000 | 1.301 | 0.000 | 1.301 | 0.000 | method roll |
| 16 | 1.184 | 0.074 | 1.207 | 0.075 | method tensor |
| 100109 | 0.920 | 0.000 | 0.920 | 0.000 | method pow |
| 20002 | 0.822 | 0.000 | 0.822 | 0.000 | method tensordot |

of function calls and obstacle area and we utilized them for each obstacle data.
**Findings:** To answer this questions, we introduce MLUPS, meaning Million Lattice Updates Per Second, is presented in **Table 1** and **Table 2** for CPUs and GPUs, respectively. Generally, ones with higher MLPUS have faster update speed, so they usually finish all the tasks in a shorter time comparing to the ones with lower MLPUS. According to the following information, on Obstacles_1, CPUs has 3.73 MLPUS, 408.06 seconds running time and 8644388.67 function calls in average, while GPUs has 33.99 MLPUS, 49.17 seconds running time and 8644872.67 function calls in average. For Obstacles_2, CPUs has 3.83 MLPUS, 397.41 seconds running time and 8644416.67 function calls in average, while GPUs has 33.78 MLPUS, 49.49 seconds running time and 8644773.0 function calls in average. For Obstacles_3, CPUs has 3.86 MLPUS, 394.49 seconds running time and 8644376.0 function calls in average, while GPUs has 33.63 MLPUS, 49.72 seconds running time and 8644684.67 function calls in average. CPUs MLPUS is 10.98%, 11.34% and 11.48% of that of GPUs, for Obstacles_1, Obstacles_2 and Obstacles_3, respectively. Now, comparing the the running time, which is represented in **Figure 4**, GPUs has a significantlly shorter running time in total. GPUs running time is 12.05%, 12.45% and 12.60% of that of CPUs, for Obstacles_1, Obstacles_2 and Obstacles_3, respectively.
It is interesting that with larger obstacles area, GPUs running time will have a higher percentage than that of CPUs. Our speculation is that this might be because more obstacles require more initial setting, which is not GPUs' speciality. For example, the Lettuce/boundary, which we talked about in the RQ1, is an initial setting in Lettuce and GPUs spent so much effort on that. Though it's obvious that GPUs finish the tasks much faster, CPUs can still do a better job in running some non-computation tasks, which was beyond our expectation when we first saw the running time difference. In general, GPUs did significantly faster than GPUs and this was almost the same as our expectation.

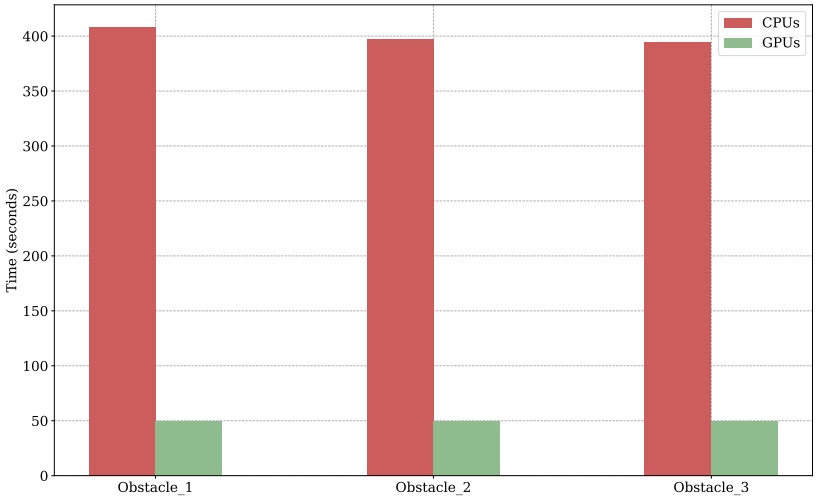


**Fig. 4** CPUs and GPUs Running time in Average

### 3.3 RQ3: Which factors can largely affect the efficiency?

**Approach:** To understand which factors can affect the efficiency, new dimension data are needed. The new data has the same obstacles information as Obstacles_1 and we introduced Obstacles_4, Obstacles_5, Obstacles_6 and Obstacles_7. By fixing the edge 300 and obstacles area, we further investigated the relation between the running time and dimension by linear regression.
**Findings:** To answer these questions, more data needed to be introduced. Due to the fact that Obstacle2D has one direction as flow direction, we decided to fix one side to make sure of the accuracy. Here, we introduced Obstacles_4, Obstacles_5, Obstacles_6 and Obstacles_7, with one of the sides is fixed for all these data, with 750, 1000, 1250, 1500×300, respectively. **Figure 5** shows the running time of Obstacles_1, Obstacles_4, Obstacles_5, Obstacles_6 and Obstacles_7, with width on the X axis. From **Figure 5**, we found out that the running time is almost on the same line, which means that it is possible to apply linear regression.
**Figure 6** includes the linear regression result and the actual running time. $Running\_time = width \times 0.0741796 + 15.568$ is what we got, indicating that it is possible that the running time is propagated to the area size of the dimension. Also, we suspected that the size changes in X direction and Y direction will have different influences on the running time.

## 4 Big Plan

Due to the time limitation, some of the questions don't have enough data and some require more readings and papers.

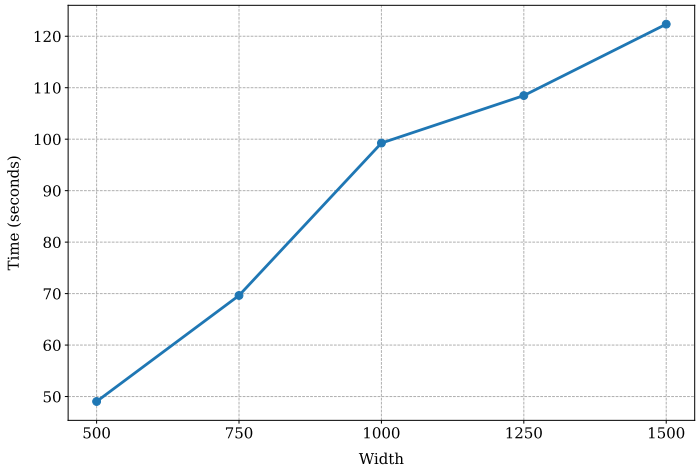


**Fig. 5** Running time of different dimensions on GPUs

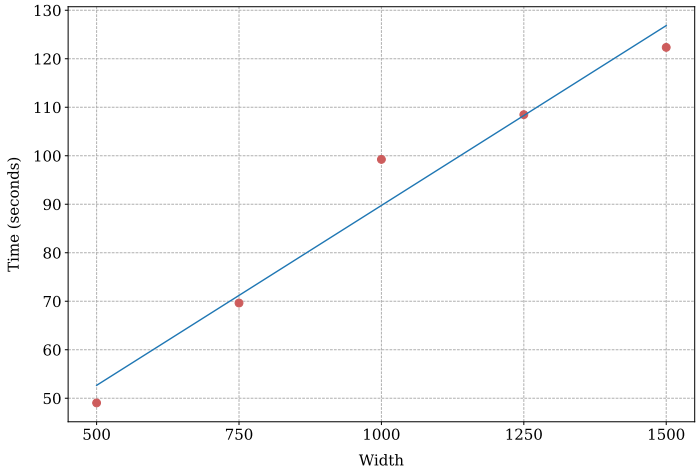


**Fig. 6** Linear Regression of running time of different dimensions on GPUs

## 4.1 Expand current Research Questions

For RQ2, we noticed that CPUs time spent on Obstacle_1, Obstacle_2 and Obstacle_3 are slightly decreasing as the obstacles area increasing. Our conjecture is that this is might because obstacle area doesn't need to do the calculation, which CPUs are not good at.
For RQ3, we figured out that it is possible that the dimension increasing in X

and Y directions have different increasing rate. The direction increment in different direction might cause different initial boundary setting up, which have a high probability of affecting the running time.

### 4.2 Other ideas

We also considered what is the relationship between 2D and 3D, but due to the time limitation, there are not enough 3D data to do the research.

## References

Bedrunka MC, Wilde D, Kliemank M, Reith D, Foysi H, Krämer A (2021) Lettuce: Pytorch-based lattice boltzmann framework. In: International Conference on High Performance Computing, Springer, pp 40–55

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. (2019) Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems 32:8026–8037

Wagner M, Llort G, Mercadal E, Giménez J, Labarta J (2017) Performance analysis of parallel python applications. Procedia Computer Science 108:2171–2179