

QDax: On the Benefits of Massive Parallelization for Quality-Diversity

Bryan Lim
Adaptive and Intelligent Robotics Lab
Imperial College London, U.K.
bryan.lim16@imperial.ac.uk

Maxime Allard
Adaptive and Intelligent Robotics Lab
Imperial College London, U.K.
m.allard20@imperial.ac.uk

Luca Grillotti
Adaptive and Intelligent Robotics Lab
Imperial College London, U.K.
luca.grillotti16@imperial.ac.uk

Antoine Cully
Adaptive and Intelligent Robotics Lab
Imperial College London, U.K.
a.cully@imperial.ac.uk

ABSTRACT

Quality-Diversity (QD) algorithms are a well-known approach to generate large collections of diverse and high-quality policies. However, QD algorithms are also known to be data-inefficient, requiring large amounts of computational resources and are slow when used in practice for robotics tasks. Policy evaluations are already commonly performed in parallel to speed up QD algorithms but have limited capabilities on a single machine as most physics simulators run on CPUs. With recent advances in simulators that run on accelerators, thousands of evaluations can be performed in parallel on single GPU/TPU. In this paper, we present QDax, an implementation of MAP-Elites which leverages massive parallelism on accelerators to make QD algorithms more accessible. We show that QD algorithms are ideal candidates and can scale with massive parallelism to be run at interactive timescales. The increase in parallelism does not significantly affect the performance of QD algorithms, while reducing experiment runtimes by two factors of magnitudes, turning days of computation into minutes. These results show that QD can now benefit from hardware acceleration, which contributed significantly to the bloom of deep learning.

CCS CONCEPTS

• **Computing Methodologies** → **Evolutionary Robotics**.

KEYWORDS

Quality Diversity, MAP-Elites, Robotics, Hardware Acceleration

ACM Reference Format:
Bryan Lim, Maxime Allard, Luca Grillotti, and Antoine Cully. 2022. QDax: On the Benefits of Massive Parallelization for Quality-Diversity. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3528927

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3528927

1 INTRODUCTION

Quality-Diversity (QD) algorithms [7, 17] have recently shown to be a very useful tool in the field of robotics. These algorithms can be used to generate a repertoire of diverse and high-performing robotic skills, which can then be used for rapid adaptation to unknown mechanical damage [3, 6, 13] and coupled with planning algorithms to perform long-horizon tasks [3]. QD algorithms are also powerful exploration algorithms. They have been shown to be effective in solving sparse-reward hard-exploration tasks in robotics and achieved state-of-the-art results on previously unsolved reinforcement learning(RL) benchmarks [9]. Lastly, QD can also be used as effective data generators for robotics RL tasks. A key reason for the success of modern machine learning come from the availability of large amounts of data. The early breakthroughs in supervised learning in computer vision have come from the availability of large diverse labelled datasets [1, 8]. The more recent successes in unsupervised learning have similarly come from methods that can leverage even larger and more diverse unlabelled datasets. As Gu et al. [12] highlighted, more efficient data generation strategies and algorithms are needed to obtain similar successes in RL. QD algorithms can play a significant role in this more data-centric view of robotics and RL by generating diverse and high-quality datasets of policies and trajectories.

QD algorithms consist of four parts (i) *selection*, (ii) *mutation* and (iii) *evaluation* and (iv) *addition*. The main bottleneck faced by QD algorithms are the large number of evaluations required that is on the order of millions. When using QD in the field of RL for robotics, this issue is mitigated by performing these evaluations in physical simulators such as Bullet [4], DART [14], and MuJoCo [19]. However, these simulators have mainly been developed to run on CPUs. Methods like MPI can be used to parallelise over multiple machines, but this requires a more sophisticated infrastructure (i.e., multiple machines) and adds some network communication overhead which can add significant run time to the algorithm. Additionally, the number of simulations that can be performed in parallel can only scale with the number of CPU cores available. Hence, both the lack of scalability coupled with the large number of evaluations required generally make the evaluation process of evolutionary based algorithms like QD, for robotics take days on modern 32-core CPUs. Our work builds on the advances and availability of hardware accelerators, high-performance programming frameworks [2] and simulators [11] that support these devices to scale QD algorithms.

Our goal in this paper is to bring the benefits of hardware acceleration to QD algorithms. For this, we present QDax, an accelerated python implementation of MAP-Elites which utilizes massive parallelism of evaluations on hardware accelerators (GPUs/TPUs). We study the effects of heavy parallelism on QD algorithms. Our experiments show that the final performance of QD algorithms remain relatively unaffected by the massive parallelization despite being up to two order of magnitudes faster. We show that this speed-up comes from a much larger throughput of evaluations per unit time and can be used to run QD algorithms in under 2 minutes. From this analysis, we propose some considerations when working in this regime of parallelism and highlight some benefits and limitations of this. An open-sourced python library of our implementation is available at https://github.com/adaptive-intelligent-robotics/QDax.

2 METHODS

We purposefully focus on a common and simple implementation of Quality-Diversity (QD) algorithms, MAP-Elites [15]. We use MAP-Elites as a baseline to study and show how QD algorithms can be scaled through parallelization. We leave other variants and enhancements of QD algorithms like CVT-MAP-Elites [20] and learnt behavioural descriptors [5] for future work; we expect these variants to only improve performance on tasks, and benefit from the same contributions of this work.

Acceleration of our implementation is enabled with: massive parallelization of evaluations, code compatible with just-in-time (JIT) compilation and fully on-device computation. The implementations of these different features present several challenges and we discuss the design and implementation decisions made to address them in the following section.

Conventional QD algorithms parallelize evaluations by utilizing multiple CPU cores, where each CPU separately runs an instance of the the simulation to evaluate a solution. We utilize Brax [11], a differentiable physics engine in Python which enables massively parallel rigid body simulations. By leveraging a GPU/TPU, utilizing this simulator allows us to massively parallelize the evaluations in the QD loop which is the major bottleneck of QD algorithms. This allows for higher throughput of evaluations each time. To provide a sense of scale, QD algorithms normally run on the order of several dozens of evaluations in parallel while Brax can simulate over 10,000 solutions in parallel. Brax is built on top of the JAX [2] programming framework, which provides an API to run accelerated code across any number of hardware acceleration devices This allows Brax to leverage the optimized hardware more easily to parallelize the operations in the core physics loop via vectorization.

Another key benefit of JAX is the JIT compilation which allows JAX to make full use of its Accelerated Linear Algebra (XLA) compiler to optimize and run code efficiently. Therefore, we also implement the entire MAP-Elites algorithm in a JIT compatible fashion. This poses some challenges as JIT compiling a function requires all array shapes to be static and known at compile time at the start of the algorithm.

Lastly, another bottleneck which slowed the algorithm down was the data transfer and marshalling across devices. As the Brax simulator uses the accelerator (GPU/TPU) to enable its massive parallelism, any operations outside of the evaluation phase would, by default, occur on the CPU. This gives rise to latency that is due to data being transferred across devices. To address this issue, we carefully consider our data structures and place all of them on-device. QDax places the QD algorithm components on the same device; components include selectors, mutations, the archive, and the physics simulator. This enables the entire QD algorithm to be run without interaction with the CPU.

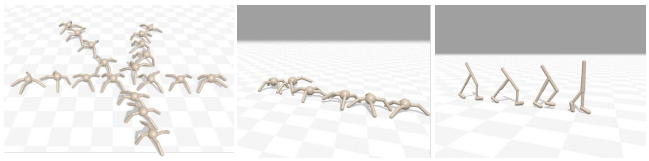


Figure 1: QD algorithms can discover diverse locomotion skills. Left: Omni-directional Ant task discovers ways to move in every direction. Centre and Right: Uni-directional Ant and Walker task respectively discover diverse gaits for moving forward.

3 EXPERIMENTS

3.1 Effects of parallelism on QD algorithms

We study the effect that massive parallelism has on the performance of the QD algorithms and the resulting repertoires. To evaluate this, we run the algorithm for a fixed number of evaluations. We use 5 million evaluations for the Ant environment and 15 million evaluations for the Walker environment (more details at the end of this section). In this experiment, we only use the A100 GPU.

We evaluate the performance of the different batch sizes on commonly used metrics in QD-literature of Coverage, Best Fitness and QD-score. The QD-score introduced by Pugh et al. [17] aims to capture both performance and diversity in a single metric. We plot the QD-score with respect to: the number of evaluations and the total run time.

Figure 2A shows the performance curves and importantly, the differences when plot against the number of evaluations and total run time. A key observation in the first row is that the QD-scores converge to the same final score after the fixed number of evaluations regardless of the batch size used. The Wilcoxon Rank-Sum Test for the final QD score across all the different batches results in p-values $p>0.05$ after applying the Bonferroni Correction. This shows that we do not observe statistically significant differences between the different batch sizes. Therefore, larger batch sizes and massive parallelism do not negatively impact the final performance of the algorithm. However, an important observation is that the larger batch sizes have a trend to be slower in terms of number of evaluations to converge. This can be expected as a larger number of evaluations are performed per epoch at larger batch sizes. Given this result, the second row of Figure 2A then demonstrates the substantial speed-up in total run time of the algorithm obtained from using larger batch sizes with massive parallelism while obtaining the same performances. We can obtain similar results but in the order of minutes instead of hours. The speed-up in total run-time of the algorithm by increasing batch size eventually disappears as we reach the limitations of the hardware.

We can also observe that in some cases (Ant-Uni), a larger batch size can even have a positive impact on the QD-score. However, an interesting observation is that the number of evaluations required

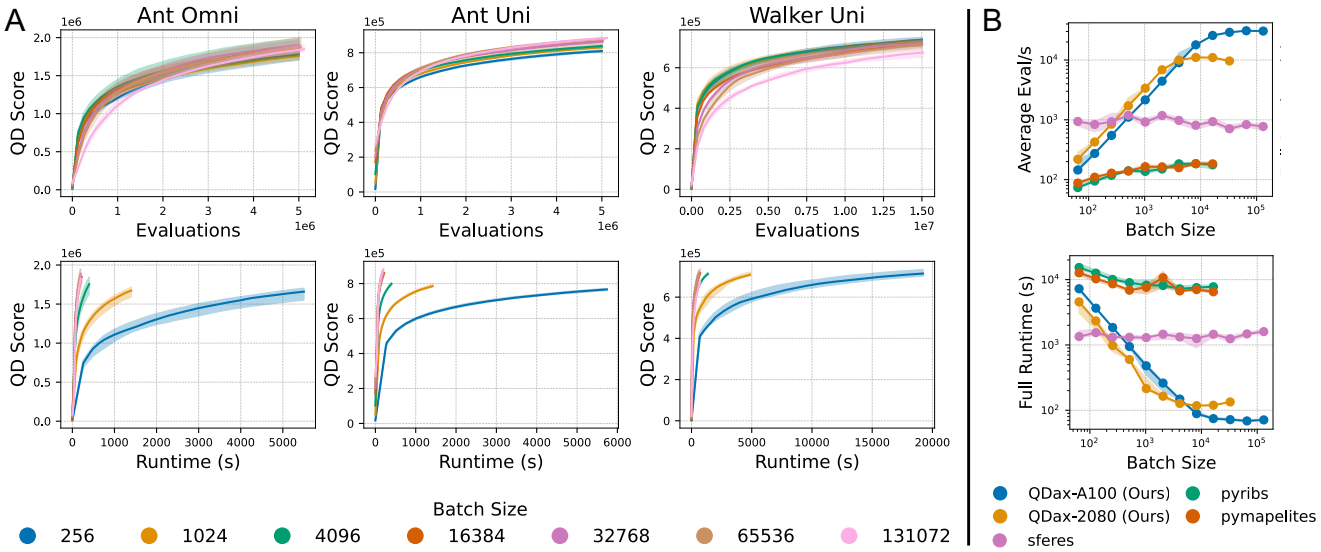


Figure 2: A: QD-Score for the omni-directional Ant, the uni-directional Ant and Walker. The plots show the QD Score w.r.t. run-times and the number of evaluations for each batch size. B: Average number of eval/s and full run time of algorithm across batch sizes and implementations. Note the log scales on both axes to make distinction between batch sizes clearer. The bold lines and shaded areas represent the median and interquartile range over 10 replications respectively.

for all the batch sizes to reach the same score are domain and task dependent. As noted in the previous figure, the larger batch sizes are slower to converge in terms of evaluations. In the Walker Uni-directional task, we run the algorithm for 15 million evaluations in order to see the same observation. This raises some interesting points on the need for sufficient epochs in QD. Given a fixed number of evaluations, a larger batch size would imply a lower number of epochs. Therefore, although our results show that a larger batch size has no negative impact on the QD algorithm and can significantly speed up the run-time of QD algorithm, the algorithm still needs to be allowed to run for a sufficient number of epochs to reach the similar final performance scores. The epochs are an important part of QD algorithms as new solutions that have been recently discovered and added to the archive from a previous epoch can be selected as a stepping stone for mutation. From our experiments, this required number of evaluations is domain and task dependent. However, given the significant speed-up in our algorithms, the termination criterion of the algorithm may no longer need to be a fixed number of generations or a fixed number of evaluations. The algorithm can be run until some performance convergence criteria instead.

3.2 Runtime speed of QD algorithms

As the increase in batch size allows QD algorihtms to scale, we also analyse the increase in evaluation throughout resulting from this. We start with a batch size $\mathcal{B}$ of 64 and double from this value until we reach a plateau and observe a drop in performance in throughput. In practice, a maximum batch size of 131,072 is used.

The number of evaluations per second (eval/s) is used to quantify this throughput. The eval/s metric is computed by running the algorithm for a fixed number of generations (also referred to as epochs) N (100 in our experiments). We divide the corresponding batch size $\mathcal{B}$ representing the number of evaluations performed in this epoch and divide this value by the time it takes to perform one epoch t_n. We use the final average eval/s across the entire run.

$$evals/s = \frac{1}{N}\sum_{n=1}^{N}\frac{\mathcal{B}}{t_n} \quad (1)$$

While the evals/s is an indication of the improvement in throughput from this implementation, we ultimately care about running the entire algorithm faster. To do this, we evaluate the ability to speed up the total runtime of QD algorithms. In this case, we run the algorithm to a fixed number of evaluations (1 million), as usually done in QD literature. Running for a fixed number of epochs would be an unfair comparison as the experiments with smaller batch sizes would have much less evaluations performed in total.

For this experiment, we consider the Ant Omni-directional task. We compare against common implementations of MAP-Elites and open-source simulators which utilize parallelism across CPU threads. We compare against both python and C++ implementations as baselines. Pymapelites [15] is a simple reference implementation from the authors of MAP-Elites that was made to be easily transformed for individual purposes. Pyribs [18] is a more recent QD optimization library maintained by the authors of CMA-ME [10]. In both python implementations, evaluations are parallelised on each core using the multiprocessing python package. Lastly, $\text{Sferes}_{\text{v2}}$ [16] is an optimized, multi-core and lightweight C++ framework for evolutionary computation, which includes QD implementations. It relies on template-based programming to achieve optimized execution speeds. The multi-core distribution for parallel evaluations is handled by Intel Threading Building Blocks (TBB) library. For simulators, we use PyBullet [4] in our python baselines and Dynamic Animation and Robotics Toolkit (DART) [14] for our C++ baseline. We perform 10 replications of each experiment using random seeds.

Figure 2B (Top) clearly shows that QDax has the ability to scale to much larger batch sizes which results in a higher throughput of evaluations. It is important to note the log scale on both axes to appreciate the magnitude of the differences. For QDax implementations (blue and orange), the number of evaluations per second scales as the batch size used increases. This value eventually plateaus once we reach the limit of the device. On the other hand, all the baseline implementations scales to a significantly lower extent. These results can be expected as evaluations using simulators which run on CPUs are limited as each CPU core runs a separate instance of the simulation. Therefore, given only a fixed number of CPU cores, these baselines would not scale as the batch size is increased. Scaling is only possible by increasing the number of CPUs used in parallel which is only possible with large distributed system with thousands of CPUs in the network. QDax can reach up to a maximum of 30,000 evaluations per second on an A100 GPU compared to maximum of 1,200 (C++) or 200 (python) evaluations per second in the baselines. This is a 30 to 100 times increase in throughput, turning computation on the order of days to minutes. The negligible differences between the pyribs (green) and pymapelites (red) results show that the major bottleneck is indeed the evaluations and simulator used, as both of these baselines use the PyBullet simulator. The performance of the $\text{Sferes}_{\text{v2}}$ (purple) implementation can be attributed to its highly optimized C++ code. However, the same lack of scalability is also observed when the batchsize is increased. When looking at run time of the algorithm for a fixed number of evaluations on Figure 2B (Bottom), we can see the effect of the larger throughput of evaluations at each epoch reflected in the decreasing run-time when larger batch sizes are used. We can run a QD algorithm with 1 million evaluations in just slightly over a minute when using a batch size of 65,536 compared to over 100 minutes taken by python baselines.

Our results also show that this scaling through massive parallelism is only limited by the hardware available. The experiments on both the RTX2080 (orange) and A100 (blue) show similar trends and increases in both evaluations per second and total runtime. The 2080 plateaus at a batch size of 8,192 capable of 11,000 eval/s while the higher-end A100 plateaus later at a batch size of 65,536 completing 30,000 eval/s.

4 DISCUSSION

In this paper, we presented QDax, an implementation of MAP-Elites that utilizes massive parallelism on accelerators that reduces the runtime of QD algorithms to interactive timescales on the order of minutes instead of hours or days. We evaluate QDax across a range of robotic locomotion tasks and show that the performance of QD algorithms are maintained despite the significant speed-up that comes with the massive parallelism.

Despite reaping the benefits of hardware in order to accelerate the algorithm, there are still some limitations that come with this implementation. This is mainly due to having to store the archive on the RAM of the device. As the archive stores the genotype among other things, the memory of the device becomes an issue preventing larger networks with more parameters from being used. Similarly, this memory limitation also prevent larger archives with more cells from being used. Through this work, we hope to see the community use QDax to iterate quicker when experimenting with new ideas and algorithms. We also hope to see new algorithmic innovations that could leverage the massive parallelism to improve performance of QD algorithms.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/V006673/1 project REcoVER, and by Google with GCP credits. We would like to thank the members of the Adaptive and Intelligent Robotics Lab for their very valuable comments.

REFERENCES

[1] Andrei Barbu, David Mayo, Julian Alverio, William Luo, Christopher Wang, Danny Gutfreund, Joshua Tenenbaum, and Boris Katz. 2019. Objectnet: A large-scale bias-controlled dataset for pushing the limits of object recognition models. (2019).

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[3] Konstantinos Chatzilygeroudis, Vassilis Vassiliades, and Jean-Baptiste Mouret. 2018. Reset-free trial-and-error learning for robot damage recovery. *Robotics and Autonomous Systems* 100 (2018), 236–250.

[4] Erwin Coumans and Yunfei Bai. 2016–2020. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

[5] Antoine Cully. 2019. Autonomous skill discovery with quality-diversity and unsupervised descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 81–89.

[6] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. 2015. Robots that can adapt like animals. *Nature* 521, 7553 (2015), 503–507.

[7] Antoine Cully and Yiannis Demiris. 2017. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation* 22, 2 (2017), 245–259.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[9] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. 2021. First return, then explore. *Nature* 590, 7847 (2021), 580–586.

[10] Matthew C Fontaine, Julian Togelius, Stefanos Nikolaidis, and Amy K Hoover. 2020. Covariance matrix adaptation for the rapid illumination of behavior space. In *Proceedings of the 2020 genetic and evolutionary computation conference*. 94–102.

[11] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. *Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation.* http://github.com/google/brax

[12] Shixiang Shane Gu, Manfred Diaz, Daniel C Freeman, Hiroki Furuta, Seyed Kamyar Seyed Ghasemipour, Anton Raichuk, Byron David, Erik Frey, Erwin Coumans, and Olivier Bachem. 2021. Braxlines: Fast and Interactive Toolkit for RL-driven Behavior Engineering beyond Reward Maximization. *arXiv preprint arXiv:2110.04686* (2021).

[13] Rituraj Kaushik, Pierre Desreumaux, and Jean-Baptiste Mouret. 2020. Adaptive prior selection for repertoire-based online adaptation in robotics. *Frontiers in Robotics and AI* 6 (2020), 151.

[14] Jeongseok Lee, Michael X. Grey, Sehoon Ha, Tobias Kunz, Sumit Jain, Yuting Ye, Siddhartha S. Srinivasa, Mike Stilman, and C. Karen Liu. 2018. DART: Dynamic Animation and Robotics Toolkit. *The Journal of Open Source Software* 3, 22 (Feb 2018), 500. https://doi.org/10.21105/joss.00500

[15] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).

[16] J.-B. Mouret and S. Doncieux. 2010. SFERESv2: Evolvin' in the Multi-Core World. In *Proc. of Congress on Evolutionary Computation (CEC)*. 4079–4086.

[17] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI* 3 (2016), 40.

[18] Bryon Tjanaka, Matthew C. Fontaine, Yulun Zhang, Sam Sommerer, Nathan Dennler, and Stefanos Nikolaidis. 2021. pyribs: A bare-bones Python library for quality diversity optimization. https://github.com/icaros-usc/pyribs.

[19] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 5026–5033.

[20] Vassilis Vassiliades, Konstantinos Chatzilygeroudis, and Jean-Baptiste Mouret. 2017. Using centroidal voronoi tessellations to scale up the multidimensional archive of phenotypic elites algorithm. *IEEE Transactions on Evolutionary Computation* 22, 4 (2017), 623–630.