# ACCELERATED QUALITY-DIVERSITY FOR ROBOTICS THROUGH MASSIVE PARALLELISM

**Anonymous authors**
Paper under double-blind review

## 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 first demonstrate the improvements on the number of evaluations per second that parallelism using accelerated simulators can offer. More importantly, 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.

## 1 INTRODUCTION

Quality-Diversity (QD) algorithms (Pugh et al., 2016; Cully & Demiris, 2017) 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 (Cully et al., 2015; Kaushik et al., 2020) and coupled with planning algorithms to perform long-horizon tasks (Chatzilygeroudis et al., 2018). 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 (Ecoffet et al., 2021). Additionally, 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 (Deng et al., 2009; Barbu et al., 2019). The more recent successes in unsupervised learning and pre-training of large models have similarly come from methods that can leverage even larger and more diverse datasets that are unlabelled and can be more easily obtained by scraping the web (Devlin et al., 2018; Brown et al., 2020). As Gu et al. (2021) 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 both with supervision and in an unsupervised setting (Cully, 2019; Paolo et al., 2020). Finally, QD algorithms are also useful in the design of more open-ended algorithms (Stanley et al., 2017; Stanley, 2019; Clune, 2019) and have shown promising signs in this direction (Stanley et al., 2017). They have been used in pioneering work for open-ended environment generation (Wang et al., 2019; 2020) and open-ended discovery of diverse skills in unsupervised manner (Cully, 2019; Paolo et al., 2020). Addressing the computation scalability of QD algorithms (focus of this work) offers a hopeful path towards open-ended algorithms that endlessly generates its own challenges and solutions to these challenges.

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 Reinforcement Learning (RL) for robotics, this issue is mitigated by performing these evaluations in physical simulators such as Bullet (Coumans & Bai, 2016–2020), DART (Lee et al., 2018), and MuJoCo (Todorov et al., 2012). 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 (Bradbury et al., 2018) and simulators (Freeman et al., 2021) that support these devices to scale QD algorithms.

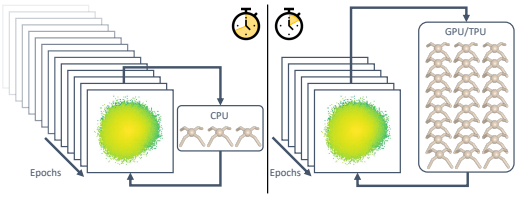


Figure 1: QDax uses massive parallelism on hardware accelerators like GPUs/TPUs to speed up runtime of QD algorithms by orders of magnitude.

Historically, significant breakthroughs in algorithms have come from major advances in computing hardware. Most notably, the use of Graphic Processing Units (GPUs) to perform large vector and matrix computations enabled significant order-of-magnitude speedups in training deep neural networks. This brought about modern deep-learning systems which have revolutionized computer vision (Krizhevsky et al., 2012; He et al., 2016; Redmon et al., 2016), natural language processing (Hochreiter & Schmidhuber, 1997; Vaswani et al., 2017) and even biology (Jumper et al., 2021). Even in the current era of deep learning, significant network architectural breakthroughs (Vaswani et al., 2017) were made possible and are shown to scale with larger datasets and more computation (Devlin et al., 2018; Shoeybi et al., 2019; Brown et al., 2020; Smith et al., 2022).

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 show that this leads to a much larger throughput of evaluations per unit time and can speed up run-time of QD algorithms to run in under 2 minutes. We also study the effects of heavy parallelism on QD algorithms. From this analysis, we propose some considerations when working in this regime of parallelism and highlight some benefits and limitations of this. Our experiments show that the final performance of QD algorithms remains relatively unaffected by the massive parallelism despite being up to two order of magnitude faster. An open-sourced python library of our implementation is available at REDACTED[1].

## 2 Related Work

### 2.1 Quality-Diversity

Quality-Diversity (QD) (Pugh et al., 2016; Cully & Demiris, 2017) is a family of optimization algorithms which searches for a large collection of both diverse and high-performing solutions, where each solution in this collection differs from another across certain attributes. This is in contrast to conventional optimization algorithms which search for a single global high-performing solution. QD algorithms were derived from interests in divergent search methods (Lehman & Stanley, 2011a) and behavioural diversity (Mouret & Doncieux, 2009; 2012) in evolutionary algorithms and hybridizing such methods with the notion of fitness (Lehman & Stanley, 2011b). Interest in QD algorithms has increased since; it has shown to be useful across many applications such as robotics (Cully & Mouret, 2013; Cully et al., 2015), video games (Gravina et al., 2019; Fontaine et al., 2020a), aerodynamic design (Gaier et al., 2018) and many more. They have also shown state-of-the-art performance in damage recovery for robotics (Cully et al., 2015) as well as hard-exploration tasks in reinforcement learning (Ecoffet et al., 2021).

[1]link will be made available after review period.

While QD algorithms are promising solutions to robotics and RL, they remain computationally expensive and take a long time to converge due to high sample-complexity. The move towards more complex environments with high-dimensional state and action spaces, coupled with the millions of evaluations required for the algorithm to converge, makes these algorithms even more inaccessible to regular hardware devices. Progress has been made towards lowering the sample complexity of QD algorithms and can generally be categorized into two separate approaches. The first approach is to leverage the efficiency of other optimization methods such as evolution strategies (Colas et al., 2020; Fontaine et al., 2020b; Cully, 2020) and policy-gradients (Nilsson & Cully, 2021; Pierrot et al., 2021). The other line of work known as *model-based quality-diversity* (Gaier et al., 2018; Keller et al., 2020; Lim et al., 2021), reduces the number of evaluations required through the use of surrogate models to provide a prediction of the behavioural descriptor (BD) and fitness. Our work takes an approach orthogonal to sample-efficiency and focuses improvement on the runtime of QD algorithms instead.

Additionally, despite algorithmic innovations that improve sample-efficiency, most QD implementations still rely on evaluations being distributed over large compute systems. These often give impressive results (Colas et al., 2020) but such resources are mainly inaccessible to most researchers and yet still take significant amount of time to obtain. Our work aims to make QD algorithms more accessible by running quickly on more commonly available accelerators, like cloud available GPUs.

### 2.2 Hardware Acceleration for Machine Learning

Machine Learning, and more specifically Deep Learning methods, have benefited from specialised hardware accelerators that can parallelize operations. In the mid-2000's researchers started using GPUs to train neural networks (Steinkrau et al., 2005) because of their high degree of parallelism and high memory bandwidth. After the introduction of general purpose GPUs, the use of specialized GPU compatible code for Deep Learning methods (Raina et al., 2009; Ciresan et al., 2012) enabled deep neural networks to be trained a few orders of magnitude quicker than previously on CPUs (Lecun et al., 2015). Very quickly, frameworks such as Torch (Collobert et al., 2011), Tensorflow (Abadi et al., 2016), PyTorch (Paszke et al., 2019) or more recently JAX (Bradbury et al., 2018) were developed to run numerical computations on GPUs or other specialized hardware.

These frameworks have led to tremendous progress in deep learning. In other sub-fields such as deep reinforcement learning (DRL) or robotics, the parallelization has happened on the level of neural networks. However, RL algorithms need a lot of data and require interaction with the environment to obtain it. Such methods suffer from a slow data collection process as the physical simulators used to collect data were mainly developed for CPUs, which results in a lack of scalable parallelism and data transfer overhead between devices. More recently, new rigid body physics simulators that can leverage GPUs and run thousands of simulations in parallel have been developed. Brax (Freeman et al., 2021) and IsaacGym (Makoviychuk et al., 2021) are examples of these new types simulators. Gradient-based DRL methods can benefit from this massive parallelism as this would directly correspond to estimating the gradients more accurately through collection of larger amounts of data at each optimization step. Consequently, better gradient estimations are shown to result in better performances (i.e. higher reward) in a faster amount of time (Rudin et al., 2021). Gu et al. (2021) and Rudin et al. (2021) both show that it is possible to train state-of-the-art control policies in minutes on single GPU stations. However, unlike gradient-based methods, it is unclear how evolutionary approaches like QD would benefit from this massive parallelism since the implications of massive batch sizes has not been studied to the best of our knowledge. It is not clear how the mutation operators and addition mechanisms present in QD algorithms would behave in the large batch size regime, calling for a more detailed study. Our work shows how massive parallelism impacts QD algorithms, and its limitations in performance.

## 3 Background: MAP-Elites

We purposefully focus on a common and simple implementation of Quality-Diversity (QD) algorithms, MAP-Elites (Mouret & Clune, 2015). 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 (Vassiliades et al., 2017) and learnt behavioural descriptors (Cully, 2019) for future work; we expect these variants to only improve performance on tasks, and benefit from the same contributions of this work.

The MAP-Elites algorithm first requires a user to define: (i) performance measure $f$ and (ii) $N$ dimensions of interest that make up the behavioural space. Each dimension is then discretized

according to user-preference which creates a grid-like structure of the behavioural space consisting of cells. The role of the cells is to store elites, highest performing solutions of each behavioural niche. This discretization can be tuned further, but heavily depends on the computational resources available. A smaller resolution discretization would create more cells in the behavioural space. MAP-Elites consists of an archive $\mathcal{X}$ which has the same structure as the behavioural space grid which stores solutions. Each solution is represented by its genotype, $x$. Our work focuses on the reinforcement learning setting, where the genotype is the parameters $\theta$ of a neural network policy $\pi_\theta$. The genotype $x$ of each solution can be evaluated to obtain its corresponding performance $f$ and behavioural descriptor $bd$.

At the start, an empty archive $\mathcal{X}$ is created. The archive is then randomly initialized by evaluating random policies. Random initialization can be done through multiple initialization schemes. A common method is to uniformly sample from the genotype parameter space. However, this requires the parameter space bounds to be defined. In the case of neural networks, we can adopt commonly used weight initialization methods such as He (He et al., 2015) or Xavier (Glorot & Bengio, 2010) initialization procedures to generate these random policies.

At every generation, MAP-Elites follows a (i) selection, (ii) variation, (iii) evaluation and (iv) addition procedure. A population $\mathcal{P}$ of batch size $\mathcal{B}$ is selected uniformly from the archive. The selected individuals (also called "parents") are then mutated to obtain $\mathcal{B}$ child individuals. Our open-source implementation consists of Gaussian noise, polynomial mutation and iso-line variation operators (Vassiliades & Mouret, 2018). We use the iso-line variation operator for all our experiments. The iso-line variation requires two parents as it is a crossover operator. Therefore, we select $2 \times \mathcal{B}$ individuals as parents during the selection phase. The $\mathcal{B}$ child individuals are then evaluated to obtain their corresponding $f$ and $bd$. In the archive addition phase, each individual is placed in their corresponding cell in the behavioural grid according to the $bd$. If the cell is empty, the individual is added to the archive. If the cell is occupied by an existing elite, the individual with the higher fitness is kept while the other is discarded.

# 4 METHODS

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. In the following section, we provide an overview of our method and discuss the design decisions made to address these challenges. A detailed explanation of the implementation details core to the method can be found in Appendix A.1.

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 (Freeman et al., 2021), 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 (Bradbury et al., 2018) programming framework, which provides an API to run accelerated code across any number of hardware acceleration devices such as CPU, GPU or TPU. 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. The next subsection details how our implementation addresses this challenge.

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.

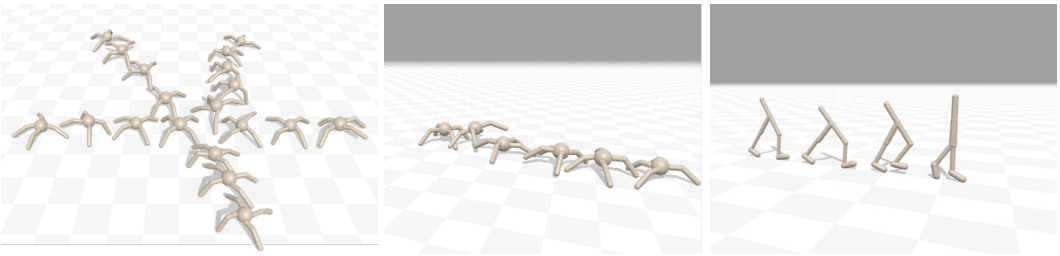

Figure 2: QD algorithms can discover diverse locomotion skills. Left: Omni-directional Ant task discovers ways to move in every direction. Centre and Right: Uni-directional Ant and Walker task respectively discover diverse gaits for moving forward.

# 5 EXPERIMENTS

Our experiments aim to answer the following questions: (1) What magnitude of speed-up does massive parallelism offer over existing implementations? (2) How does this differ across different hardware accelerators? (3) How does parallelism affect performance of Quality-Diversity (MAP-Elites) algorithms?

*Tasks and Environments.* We perform experiments on two different continuous control tasks; *omni-directional* locomotion and *uni-directional* locomotion. In the omni-directional task, the goal is to discover locomotion skills to move in every direction while minimizing energy used. The behavioural descriptor (BD) is defined as the final $x$-$y$ position of the centre of mass of the robot at the end of the episode while the fitness is defined as a sum of a survival reward and torque cost.

In contrast, the goal in the uni-directional tasks is to find a collection of diverse gaits to walk forward as fast as possible. In this task, the BD is defined as the average time over the entire episode that each leg is in contact with the ground. For each foot $i$, the contact with the ground $C_i$ is logged as a Boolean (1: contact, 0: no-contact) at each time step $t$. The BD definition of this task was used to allow robots to recover quickly from mechanical damage (Cully et al., 2015) and in recent benchmark QD tasks (Nilsson & Cully, 2021). The fitness of this task is a sum of the forward velocity of the robot's centre of mass, survival reward and torque cost. Detailed equations describing the BD and fitness both these tasks can be found in Appendix A.2.

We use the Ant and Walker2D gym locomotion environments made available on Brax (Freeman et al., 2021) on these tasks. We run the Ant experiments for an episode length $T$ of 100 steps and Walker experiments for 250 steps. In total, we report results on a combination of three tasks and environments; Omni-directional Ant, Uni-directional Ant, Uni-directional Walker. Figure 2 illustrates examples of the types behaviours discovered from these tasks. We use a behavioural grid shape of [100,100], [10,10,10,10] and [40,40] respectively for the corresponding tasks. We use fully connected neural network controllers with two hidden layers of size 64 and tanh activation functions as policies across all environments and tasks.

## 5.1 RUNTIME SPEED OF QD ALGORITHMS

We first evaluate the ability of our implementation of MAP-Elites across increasing batch sizes to see the increase in evaluation throughput we can obtain. 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 value of this metric across the entire run $\frac{1}{N}\sum_{n=1}^{N}\frac{\mathcal{B}}{t_n}$. While the evaluations per second can be 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.

| Implementation | Simulator | Resources | Eval/s | Batch size | Runtime (s) | Batch size |
|---|---|---|---|---|---|---|
| QDax (Ours) | Brax | GPU A100 | **30,846** | 65,536 | **69** | 65,536 |
| QDax (Ours) | Brax | GPU 2080 | 11,031 | 8,192 | 117 | 8,192 |
| pyribs | PyBullet | 32 CPU-cores | 184 | 8,192 | 7,234 | 4,096 |
| pymapelites | PyBullet | 32 CPU-cores | 185 | 8,192 | 6,509 | 16,384 |
| $\text{Sferes}_{\text{v2}}$ | DART | 32 CPU-cores | 1,190 | 512 | 1,243 | 32,768 |

Table 1: Maximum throughput of evaluations per second and fastest run time obtained and their corresponding batch sizes across implementations. The medians over the 10 replications are reported.

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 (see Table 1). All baseline algorithms used simulations with a fixed number of timesteps (100 in our experiments). We compare against both python and C++ implementations as baselines. Pymapelites (Mouret & Clune, 2015) is a simple reference implementation from the authors of MAP-Elites that was made to be easily transformed for individual purposes. Pyribs (Tjanaka et al., 2021) is a more recent QD optimization library maintained by the authors of CMA-ME (Fontaine et al., 2020b). In both python implementations, evaluations are parallelised on each core using the multiprocessing python package. Lastly, $\text{Sferes}_{\text{v2}}$ (Mouret & Doncieux, 2010) 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 (Coumans & Bai, 2016–2020) in our python baselines and Dynamic Animation and Robotics Toolkit (DART) (Lee et al., 2018) for our C++ baseline.

We also test our implementation on two different GPU devices, a more accessible RTX2080 local device and a higher-performance A100 on Google Cloud. We only consider a single GPU device at each time. QDax was also tested and can be used to perform experiments across distributed GPU and TPU devices but we omit these results for simplicity.

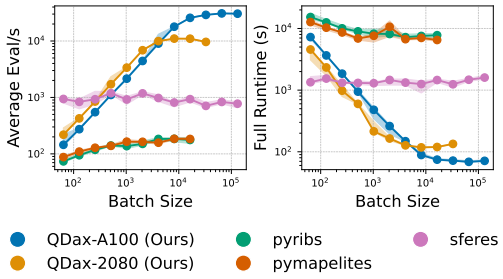


Figure 3: Average number of evaluations per second and full run time of algorithm across batch sizes and implementations. Note the log scales on both axes to make distinction between batch sizes clearer. The bold lines and shaded areas represent the median and interquartile range over 10 replications respectively.

Figure 3 (Left) 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 (see Table 1). 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 3 (Right), 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 (See Table 1) 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.

### 5.2 Effects of parallelism on QD algorithms

In the previous section, we showed that QDax has the ability to significantly speed up QD algorithms through massive parallelism of the evaluations. We now study the effect that this 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 environments and 15 million evaluations for the Walker environment (more details at the end of this section). We evaluate the performance of the different batch sizes on commonly used metrics in QD-literature of Coverage, Best Fitness and QD-score. The coverage is used as a measure of the diversity of skills discovered without considering the fitness of individuals. It is computed by counting the number of cells of the archive filled. The Best Fitness metric captures the highest performing individual in the archive regardless of behavioural descriptor. Lastly, the QD-score introduced by Pugh et al. (2016) aims to capture both performance and diversity in a single metric. This metric is computed as the sum of fitness of all the individuals in the archive, where the fitness must be strictly positive. We plot these metrics with respect to three separate factors: number of evaluations, epochs and total run time. In this experiment and all following experiments, we only use the A100 GPU.

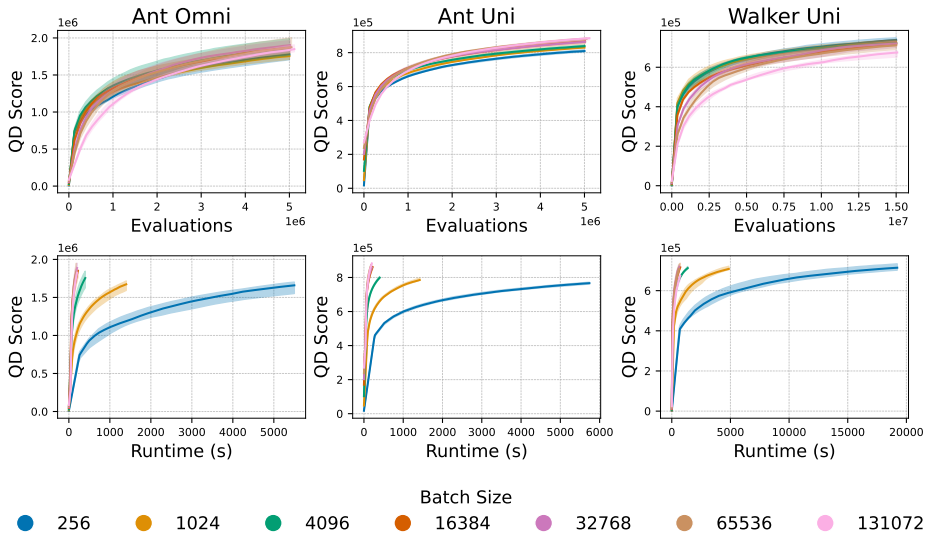


Figure 4: Performance Metrics for three different environments, namely the omni-directional ant, the uni-directional Ant and Walker. The plots show the QD Score with respect to run-times and the number of evaluations for each batch size.

Figure 6 (Appendix A.3) shows examples of the BD-spaces generated on each of the different tasks. We first focus on the results and observations from just a single environment - Omni-directional Ant, (Figure 5) and later show that this result translates across all the environments tested (Figure 4). Figure 5 (Appendix A.3) shows the performance curves across all the metrics and more importantly the differences when plot against the number of evaluations, epochs and total run time. A key observation in the first row is that all the metrics 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 third (last) row of Figure 5 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. An expected observation we see when comparing the plots in the second and third row of Figure 5 is how the total run time is proportional to the number of epochs. As we are able to increase the evaluation throughput at each epoch, it takes a similar amounts of time to evaluate both smaller and larger batch sizes. The speed-up in total run-time of the algorithm by increasing batch size eventually disappears as we reach the limitations of the hardware. This corresponds to the results presented in the the previous section 5.1 (Figure 3) where we see the number evaluations per second plateauing.

These observations and trends are also reflected across all the environments as seen in Figure 4. 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 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. This can be observed in the second row of Figure 5. 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. This may of course have its limitations in open-ended applications where no convergence is expected.

## 6 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. An important point to note is also the comparison across different simulators will never be perfect. Each simulator has its individual pros and cons in terms of fidelity, execution time and parallelism capabilities.

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.

## References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementa-*

*tion (OSDI 16)*, pp. 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

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

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. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

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

Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012. URL http://arxiv.org/abs/1202.2745.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

Cédric Colas, Vashisht Madhavan, Joost Huizinga, and Jeff Clune. Scaling map-elites to deep neuroevolution. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 67–75, 2020.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. 2011. URL http://infoscience.epfl.ch/record/192376.

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2020.

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

Antoine Cully. Multi-emitter map-elites: Improving quality, diversity and convergence speed with heterogeneous sets of emitters. *arXiv preprint arXiv:2007.05352*, 2020.

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

Antoine Cully and Jean-Baptiste Mouret. Behavioral repertoire learning in robotics. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 175–182, 2013.

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

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

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

Matthew C Fontaine, Ruilin Liu, Ahmed Khalifa, Jignesh Modi, Julian Togelius, Amy K Hoover, and Stefanos Nikolaidis. Illuminating mario scenes in the latent space of a generative adversarial network. *arXiv preprint arXiv:2007.05674*, 2020a.

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

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

Adam Gaier, Alexander Asteroth, and Jean-Baptiste Mouret. Data-efficient design exploration through surrogate-assisted illumination. *Evolutionary computation*, 26(3):381–410, 2018.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)*, pp. 1–8. IEEE, 2019.

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. Brax-lines: Fast and interactive toolkit for rl-driven behavior engineering beyond reward maximization. *arXiv preprint arXiv:2110.04686*, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596 (7873):583–589, 2021. doi: 10.1038/s41586-021-03819-2.

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

Leon Keller, Daniel Tanneberg, Svenja Stark, and Jan Peters. Model-based quality-diversity search for efficient robot learning. *arXiv preprint arXiv:2008.04589*, 2020.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature 2015 521:7553*, 521: 436–444, 5 2015. ISSN 1476-4687. doi: 10.1038/nature14539. URL https://www.nature.com/articles/nature14539.

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

Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011a.

Joel Lehman and Kenneth O Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 211–218, 2011b.

Bryan Lim, Luca Grillotti, Lorenzo Bernasconi, and Antoine Cully. Dynamics-aware quality-diversity for efficient learning of skill repertoires. *arXiv preprint arXiv:2109.08522*, 2021.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance gpu-based physics simulation for robot learning. *CoRR*, abs/2108.10470, 2021. URL https://arxiv.org/abs/2108.10470.

J.-B. Mouret and S. Doncieux. SFERESv2: Evolvin' in the multi-core world. In *Proc. of Congress on Evolutionary Computation (CEC)*, pp. 4079–4086, 2010.

J-B Mouret and Stéphane Doncieux. Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation*, 20(1):91–133, 2012.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Jean-Baptiste Mouret and Stéphane Doncieux. Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity. In *2009 IEEE Congress on Evolutionary Computation*, pp. 1161–1168. IEEE, 2009.

Olle Nilsson and Antoine Cully. Policy gradient assisted map-elites. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 866–875, 2021.

Giuseppe Paolo, Alban Laflaquiere, Alexandre Coninx, and Stephane Doncieux. Unsupervised learning and exploration of reachable outcome space. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2379–2385. IEEE, 2020.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

Thomas Pierrot, Valentin Macé, Geoffrey Cideron, Karim Beguir, Antoine Cully, Olivier Sigaud, and Nicolas Perrin. Diversity policy gradient for sample efficient quality-diversity optimization. 2021.

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

Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 873–880, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553486. URL https://doi.org/10.1145/1553374.1553486.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. *arXiv preprint arXiv:2109.11978*, 2021.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model, 2022.

Kenneth O Stanley. Why open-endedness matters. *Artificial life*, 25(3):232–235, 2019.

Kenneth O Stanley, Joel Lehman, and Lisa Soros. Open-endedness: The last grand challenge you've never heard of. *While open-endedness could be a force for discovering intelligence, it could also be a component of AI itself*, 2017.

Dave Steinkrau, Patrice Y. Simard, and Ian Buck. Using gpus for machine learning algorithms. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition*, ICDAR '05, pp. 1115–1119, USA, 2005. IEEE Computer Society. ISBN 0769524206. doi: 10.1109/ICDAR.2005.251. URL https://doi.org/10.1109/ICDAR.2005.251.

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

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

Vassilis Vassiliades and Jean-Baptiste Mouret. Discovering the Elite Hypervolume by Leveraging Interspecies Correlation. In *GECCO 2018 - Genetic and Evolutionary Computation Conference*, Kyoto, Japan, July 2018. doi: 10.1145/3205455.3205602. URL https://hal.inria.fr/hal-01764739.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Poet: open-ended coevolution of environments and their optimized solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 142–151, 2019.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth Stanley. Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In *International Conference on Machine Learning*, pp. 9940–9951. PMLR, 2020.

# A Appendix

## A.1 Implementation Details

This section follows Section 4 and provides a detailed description of the implementation details of each component of the algorithm to enable acceleration.

In order to leverage the benefits of vectorised operations, JAX requires all the data on the device to be in the form of a Tensor with immutable size (classes and objects cannot be used directly).

*Archive Structure.* The first data structure that we have to place on the device is the archive of MAP-Elites. However the number of individuals stored in the archive is meant to change overtime as new individuals are added to it and other individuals being replaced. To accommodate these requirements, we implemented MAP-Elites with a fixed grid-sized archive in the form of a preallocated static array containing individuals with NaN values. In MAP-Elites, the algorithm needs to store three

attributes per individual as mentioned in section 3: (i) the fitness, (ii) the BD and (iii) the genotype. The fitness values for all the individuals are initialized as an array of NaN values and the genotype for all the individuals as an array of zeros. The size of these arrays are defined as the number of cells the behavioural space contains. The behavioural descriptors of the individuals are then directly encoded as indexes of the array storing the corresponding genotypes and fitness values. Defining the fitness and genotype as static arrays allows JAX to execute the addition and replacements of the new solutions directly on the device. This prevents the overhead caused by the transfer of data between the device and the CPU at each epoch to perform these operations.

*Selection and Mutation.* As the archive is already completely preallocated with zero-outed individuals, sampling from the archive requires some considerations. To find cells that are actually filled, the stored fitness values are used to check for valid, non-NaN entries. The indexes of these valid entries are stored at the top of a new static array of the size of the total number of cells. The bottom of this array is padded with zero values. To select individuals from the archive, the algorithm samples uniformly with replacement the indexes of valid content of this array to generate the indexes of the selected individuals. Consequently, this allows the algorithm to use these indexes to form a static array containing the genotypes of the selected parents by copying the corresponding genotypes from the archive. Finally, the "Iso+LineDD" variation operator is mapped to the static batch of parents to generate the offspring. This operation is performed in parallel, since every pair of individuals used for the cross-over are independent.

*Evaluation.* The evaluation of the solutions is done in parallel as a batch with Brax (Freeman et al., 2021) as described in the previous section 4. Brax systematically simulates a static number of steps as it is built upon JAX and static arrays. Due to the constraint of static arrays, Brax has to simulate all the steps in the specified episode length even if the agent fails during this episode. This differs from other simulators where the simulation stops as soon as an agent fails. For this reason, we capture the behavioural descriptor from the trajectory up until the time step just before the agent fails the task. This is in contrast to taking the behavioural descriptor from the trajectory up until the final time step. Alternatively, we can also flag failed individuals with a "death flag" to discard them from being considered for addition.

*Archive Addition.* Adding individuals to the archive is challenging since the batch of evaluated individuals can contain several individuals with the same behavioural descriptor but with different fitness values. This causes a conflict when trying to apply the addition condition to all the individuals in a single JAX operation, as new individuals have to be compared to the content of the archive and between themselves. To overcome this issue, the archive addition follows four steps: (i) a boolean mask is applied to remove the new individuals with a death flag, (ii) the fitness function of all new individuals having the same $bd$ is set to $-\infty$ except for the best individual which keeps its fitness value, (iii) we compare the fitness value of existing individuals in the archive to the offspring and mask the new individuals having a worse fitness, (iv) the boolean mask and the death flags determine together which individual will be added by making JAX ignore individuals that haven't met the criteria to enter the archive.

All of these steps are JIT compatible since all the array sizes are static. As a result, the comparison and addition can be executed in a single set of operations (one for each of the four steps above) regardless of the actual batch size and directly on the device.

### A.2 Fitness and Behavioural Descriptor Definition

This section provides the equations for computing the behavioural descriptor $bd$ and fitness $f$ for the tasks performed in our experiments, as described in Section 5. For the omni-directional tasks, the $bd$ and $f$ are given by:

$$bd_{omni} = \begin{pmatrix} x_T \\ y_T \end{pmatrix} \tag{1}$$

$$f_{omni} = \sum_{t=0}^{T} r_{survive} + (-r_{torque}) \tag{2}$$

For the uni-directional tasks, the $bd$ and $f$ can be computed using:

$$bd_{uni} = \frac{1}{T}\sum_{t}^{T}\begin{pmatrix} C_1(t) \\ \vdots \\ C_I(t) \end{pmatrix}, \text{ with } I \text{ the number of feet.} \tag{3}$$

$$f_{uni} = \sum_{t=0}^{T} r_{forward} + r_{survive} + (-r_{torque}) \tag{4}$$

### A.3 Supporting Results

This section provides supporting figures that are referenced by the text in the experiments section 5. Figure 5 focuses on the performance curves on the omni-directional Ant task across batch sizes. This is analysed and explained in Section 5.2.

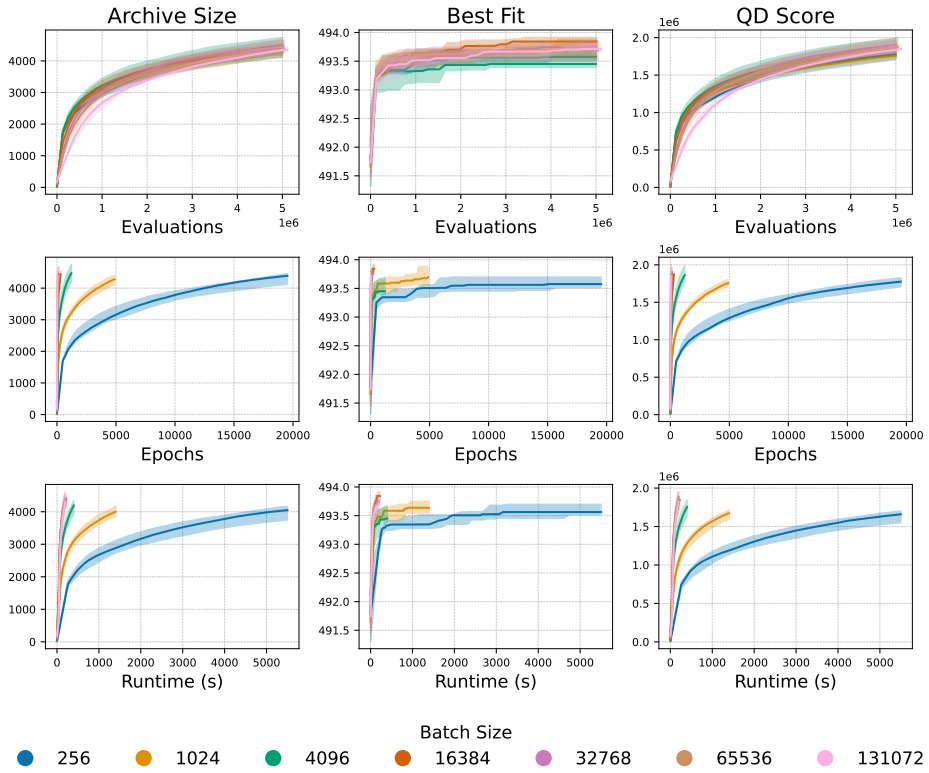


Figure 5: Performance curves of QDax for different batch sizes across generations of the algorithm on the omni-directional Ant task. The metrics are plotted against evaluations (top row), epochs (middle row) and runtime (bottom row) to demonstrate the effect of massive parallelism.

Figure 6 shows examples of the resulting behavioural repertoires obtained using QDax across the different environments. QDax illuminates the behaviour space resulting in a repertoire of diverse and high-performing policies.

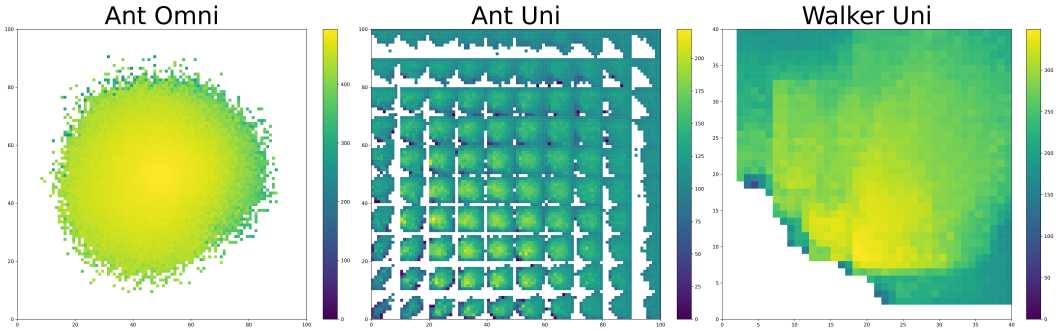


Figure 6: Example of behavioural repertoires obtained from the omni-directional Ant, the uni-directional Ant and Walker. A randomly selected replication is used for this visualization. Each cell represents a policy where each dimension of plot corresponds to one dimension of the BD space. The color represents the fitness (lighter has higher fitness).