Supplementary Table 1 – Example datasets used in this publication.

Label	Description	Data Shape	Link
a549_mt	Fixed A549 cells immunolabelled against ß-tubulin with Alexa Fluor™ 647, acquired with the Oxford Nanoimager (ONI)	10000x283x283	Available in Zenodo: https://zenodo.org/record/8318395
huvec_nuclei	HUVEC stained with DAPI, acquired in a Marianas spinning-disk confocal microscope.	100x512x512	

Supplementary Note 1: How computational acceleration for an algorithm implementation written with Cython, PyOpenCL and Numba is achieved.

Python is an interpreted, high-level programming language that allows rapid development and easy code readability. However, Python's flexibility and dynamic nature come at the cost of performance speed. Operations in pure Python are generally slower compared to compiled languages like C/C++. There are several methods to accelerate Python code by bypassing the Global Interpreter Lock (GIL) and compilation to machine code. Three popular approaches are Cython[1], PyOpenCL[2] and Numba[3]:

- Cython[1] is a static compiler that converts Python code into optimised C/C++ code that can be compiled into a Python extension module. It provides Python-like syntax while supporting calling C functions and declaring C types. Cython[1] code runs significantly faster than Python because it bypasses the GIL to allow multi-threading and performs low-level optimisations like loop unrolling. One limitation is that Cython[1] requires explicit type declarations, which removes some of Python's dynamism. Overall, Cython[1] can accelerate Python code 5-1400x faster (Supplementary Figure 1).
- PyOpenCL[2] allows Python code to execute parallel computations on Graphical Processing Units (GPUs) through the OpenCL[4] framework. Computational tasks are offloaded to the GPU, which has thousands of tiny processing cores suited for data-parallel operations. PyOpenCL[2] translates Python functions into OpenCL[4] kernels that run efficiently on GPUs. This offers massive parallelism and speedup compared to Python limited by single-CPU execution. Despite PyOpenCL[2] not requiring a physical GPU to run, the best and easiest performance improvement requires one, which can be a limiting factor for some users. Overall, PyOpenCL[2]p can accelerate some workloads up to 150x by harnessing GPU parallelism[5] (Supplementary Figure 1).
- Numba[3] is a just-in-time (JIT) compiler that converts Python functions into optimised machine code using the LLVM compiler framework. It is designed to accelerate numerical and scientific workloads using NumPy[6] arrays and math operations. Numba[3]-compiled code avoids interpreter overhead and leverages vectorisation, loop-unrolling and parallel execution on multicore CPUs. But Numba[3] has compilation overhead on first run. Overall, Numba[3] can speed up math-heavy Python code by 100-400x by generating optimised machine code specialized for CPUs (Supplementary Figure 1).

In summary, all these three tools can significantly accelerate Python code by bypassing interpreter overhead and utilizing efficient compilation, parallelism, and hardware optimisation. Cython[1] translates Python to C/C++ code that can multi-thread and leverage CPU efficiency. PyOpenCL[2] taps into massively parallel GPU hardware. Numba[3] optimises machine code for numerical workloads on CPUs. Typical speedups depend on the methods used, nature of the code, size of the input data and hardware used.

Supplementary Note 2: Metaprogramming in the Liquid Engine

Metaprogramming is a programming technique where a program can manipulate or generate code during runtime. In NanoPyx, metaprogramming is used to generate optimised implementations of the same task automatically.

The Liquid Engine uses two metaprogramming tools: tag2tag and c2cl.

The tag2tag tool enables developers to generate multiple implementations of the same algorithm written as C or Python code snippets. Effectively, tag2tag transcribes these snippets into single-threaded and multi-threaded versions of the code, generally then called by Cython[1]. Specifically, developers can write a single version of the code and delimit the "tag" to be propagated to the other implementations (e.g., a function). The tag2tag tool is able to read the content of the file, identify the tags, and store them in a dictionary, where the tag name is the key, and the associated code snippet is the value. After choosing the "tag", in the same script, the developer can create a tag-copy, where they specify what part of the original tag should be replaced, and what to replace it for. tag2tag will identify the tag placeholders and apply the specified replace commands to the associated tag code. It then replaces the tag placeholder in the file with the modified tag code. This tool is intended for use with Python (.py), Cython[1] (.pyx), and OpenCL[4] (.cl) files. In practice, for most tasks implemented in the Liquid Engine, we used a single-threaded version of the code as the original tag and create multi-threaded versions of the code by replacing the range in the for loops with prange. Additionally, we added implementations with different schedulers for the parallelisation. This allows the developer to easily create as many code variations as they find necessary. This approach streamlines the process of maintaining consistency across various code implementations, as altering the code in one version ensures that the modification is seamlessly and consistently applied to all other relevant versions. As a result, developers can effectively manage code updates and improvements, as these are propagated into the other implementations effortlessly, reducing redundancy and enhancing code maintainability.

Another meta-programming tool used in the Liquid Engine is the c2cl tool. c2cl is analogous to the tag2tag tool, but specifically designed to extract C functions and propagate them into .cl files, so the C functions can be used in OpenCL kernels. Manual conversion of these adaptations would be time-consuming and prone to errors, which is where c2cl comes in. The tool automates the process of porting C code to OpenCL[4] by extracting reusable code blocks from the C functions, converting the C code into valid OpenCL[4] kernels, and inserting the modified kernels back into the OpenCL[4] file.

The Liquid Engine also supports Numba[3] as an alternative performance-boosting option for Python code snippets. With all these implementations, NanoPyx can be run and used by users with diverse hardware configurations.

Overall, the Liquid Engine extensively uses metaprogramming techniques to avoid manual coding. Code generation and transformations are used to automatically create specialised implementations, resulting in a simple and flexible architecture.

Supplementary Note 3: The machine learning basis of the Liquid Engine

The NanoPyx Liquid Engine presents a straightforward machine learning technique for performance self-tuning. To do so, it logs execution times of operations across multiple implementations like Python, Numba[3], Cython[1] and OpenCL[4]. These benchmarking times are used to train basic regression models to predict the occurrence of delays. If delays are detected the trained models are used at runtime to estimate the magnitude and occurrence of a delay in a specific implementation. Over time, the benchmarking data is aggregated to refine the models continuously. This allows the Liquid Engine to "learn" the optimal implementations for a given platform, device, and data shape to maximize performance. It's important to note that these principles do not use neural networks, rather focusing on looped data-driven optimisations. In summary, fundamental machine learning principles are applied in the engine itself for auto-tuning - the methods exposed to users are traditional image processing functions using optimised implementations under the hood.

Supplementary Note 4: Fuzzy logic

The Liquid Engine employs fuzzy logic[7] to match a specific function call to its most similar past benchmark.

The Liquid Engine adaptive nature is highly dependent on the existence of appropriate benchmarks for each implementation. As previously demonstrated, the time it takes for a particular image analysis task to execute is greatly influenced by the size of the input image and by the parameters chosen to perform the given task. To address this variability, benchmarks are stored separately for each unique set of parameters and data size. This approach allows the Liquid Engine to dynamically adjust its performance based on the specific conditions of each task, resulting in more accurate and optimised outcomes.

When the Agent receives a request to execute a method using the Liquid Engine, it checks its records for historical runtime data. However, when dealing with a new method executed with a unique combination of parameters, no existing benchmarks are available. To address this, the Agent employs the following strategy: it searches for the most similar set of parameters that has been previously used. This is because each set of parameters is associated with a runtime score, which was saved along with the historical runtime data in benchmark files. This score is determined by considering various factors, such as the dimensions of the input image and other relevant parameters. By finding the most similar parameters with known benchmark data, the Agent leverages this score to estimate and adapt the expected runtime performance for the new method, even in the absence of specific benchmarks. This adaptive approach allows the Liquid Engine to intelligently adjust its behaviour and make informed decisions when executing methods with varying input conditions.

Finally, if no appropriate benchmarks exist for a specific run type, the score of the current parameters is compared to the score of all other benchmarks, and the benchmarks with the closest score are used.

Supplementary Note 5: Example datasets in NanoPyx

NanoPyx provides users a wide range of example datasets. These datasets not only draw from our previous publications but also tap into publicly available datasets[8]. These were integrated into the NanoPyx framework to facilitate testing and development, offering users an opportunity to gain hands-on experience and explore the capabilities of the library.

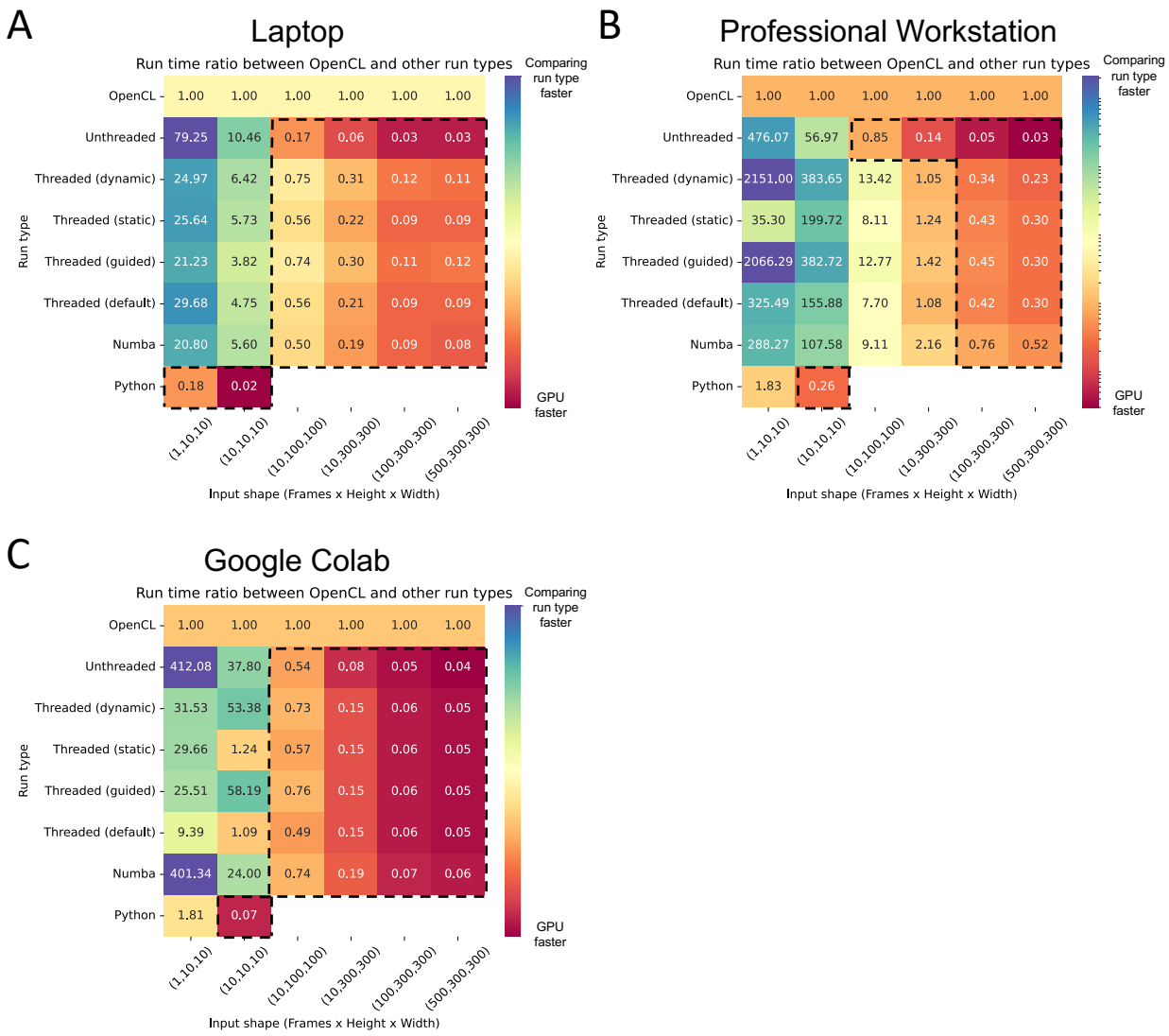
These include single-molecule localization microscopy data of Cos7 cells expressing Utrophin-GFP[9]; U2OS with microtubules labelled with AF647; Jurkat T cells expressing LifeAct-GFP[10]; Structured Illumination Microscopy data of VACV A4 virions[11]; among others.

The management and loading of datasets within NanoPyx are orchestrated through a class specifically designed for data management, facilitating efficient access and use. The class provides functions to list datasets and retrieve their information, enabling users to effortlessly identify and choose relevant datasets.

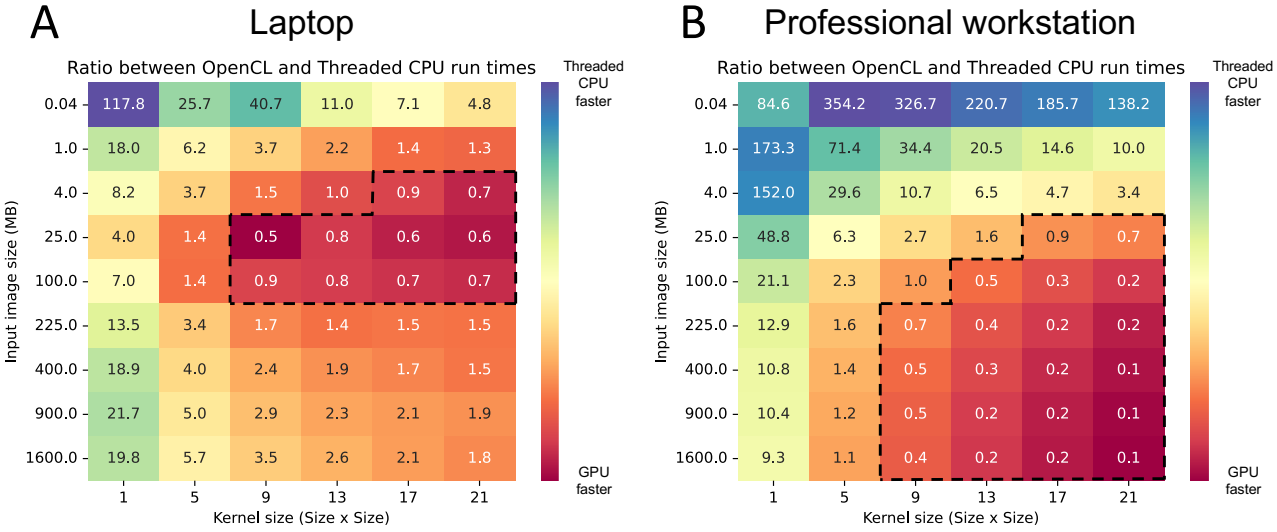
Most of the datasets are stored and accessed via Google Drive and can be automatically downloaded as zip files.

Once downloaded, these zip files can be effortlessly converted into numpy arrays, a process that seamlessly manages the complexities of image retrieval and manipulation.

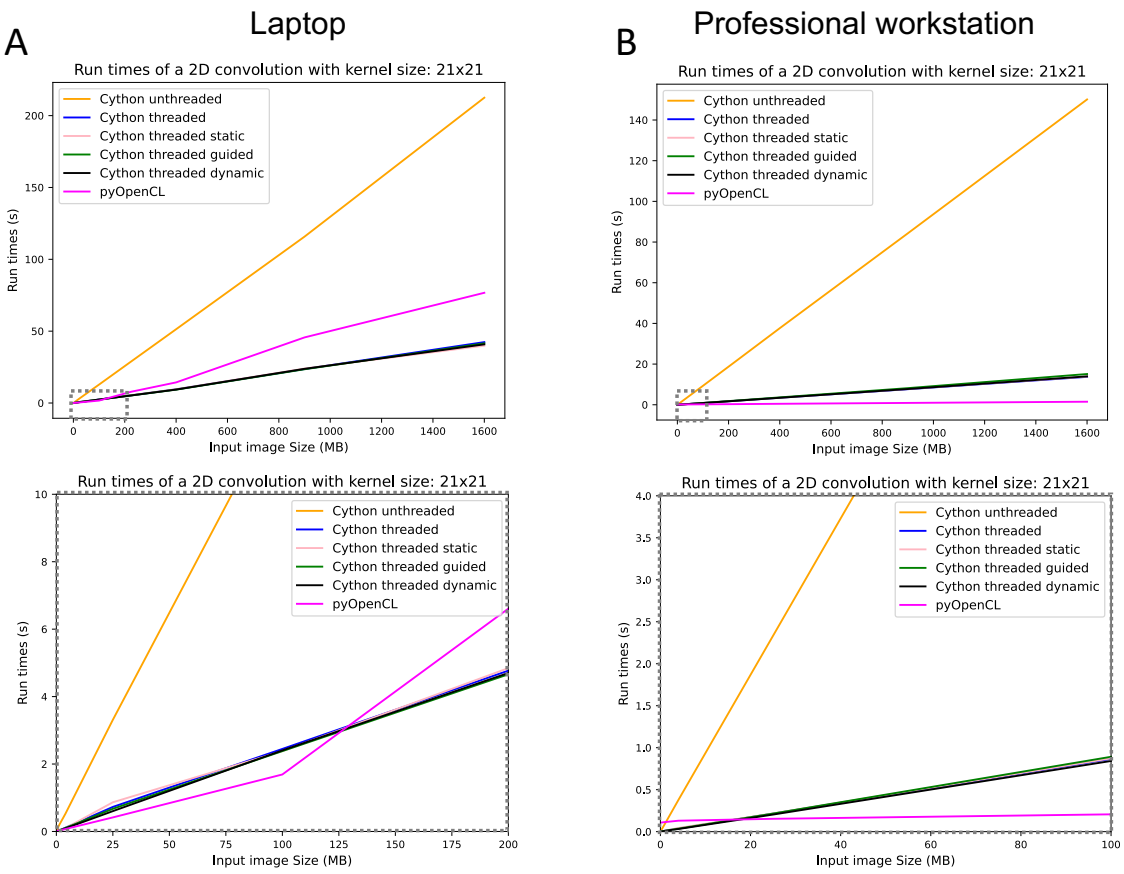
In Jupyter or Colab notebooks, users can easily access the datasets through a user-friendly graphical interface (GUI) that has been developed to streamline the process. This interface empowers users to select datasets from a diverse array of options, all of which are named to provide clear context. This intuitive approach ensures efficient dataset selection and integration into analysis workflows.



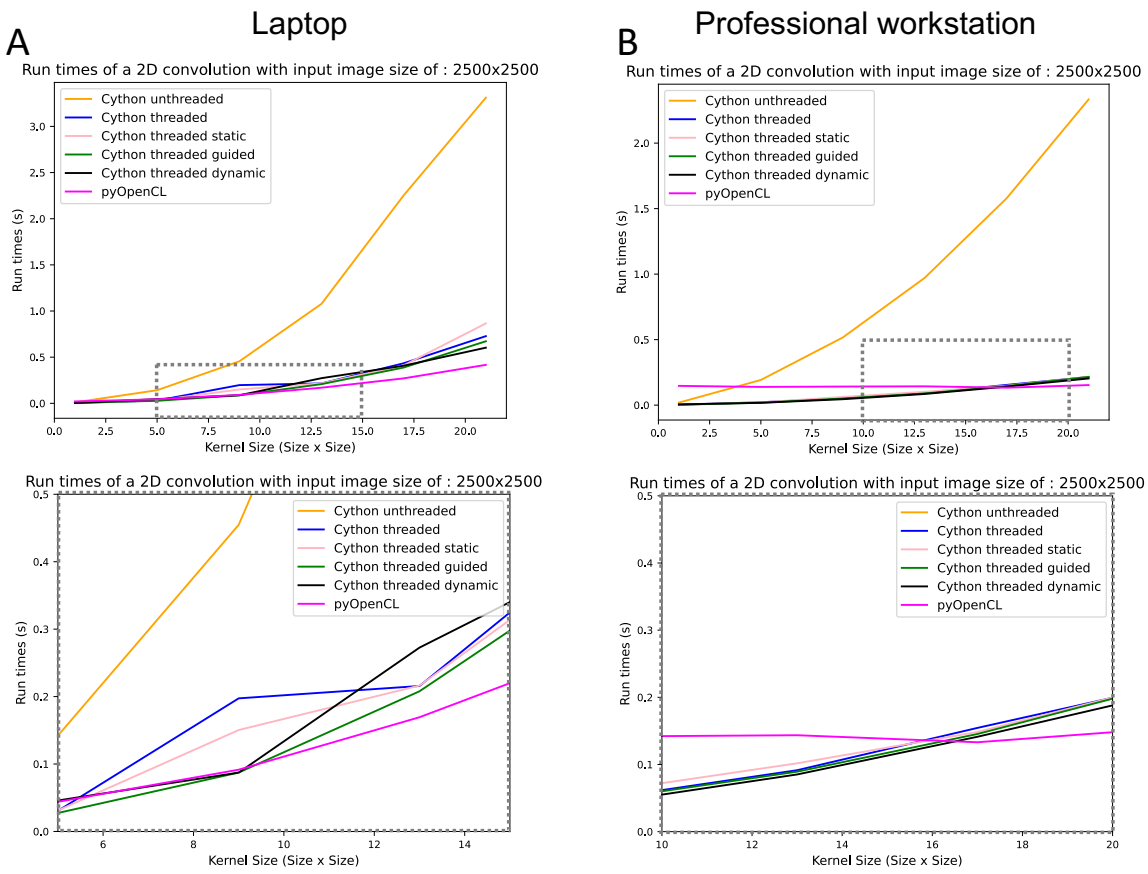
Supplementary Figure 1 – Ratio between the run times of OpenCL and other implemented run types. Run times of a 5x Catmull-rom[12] interpolation were measured across multiple input data sizes using either a MacBook Air M1 (A), a Professional Workstation (B) or Google Colaboratory (C). Using the fastest implementation can lead to up to 10x faster code execution. Area within dashed lines correspond to kernel and image sizes where OpenCL is faster than other implementations.



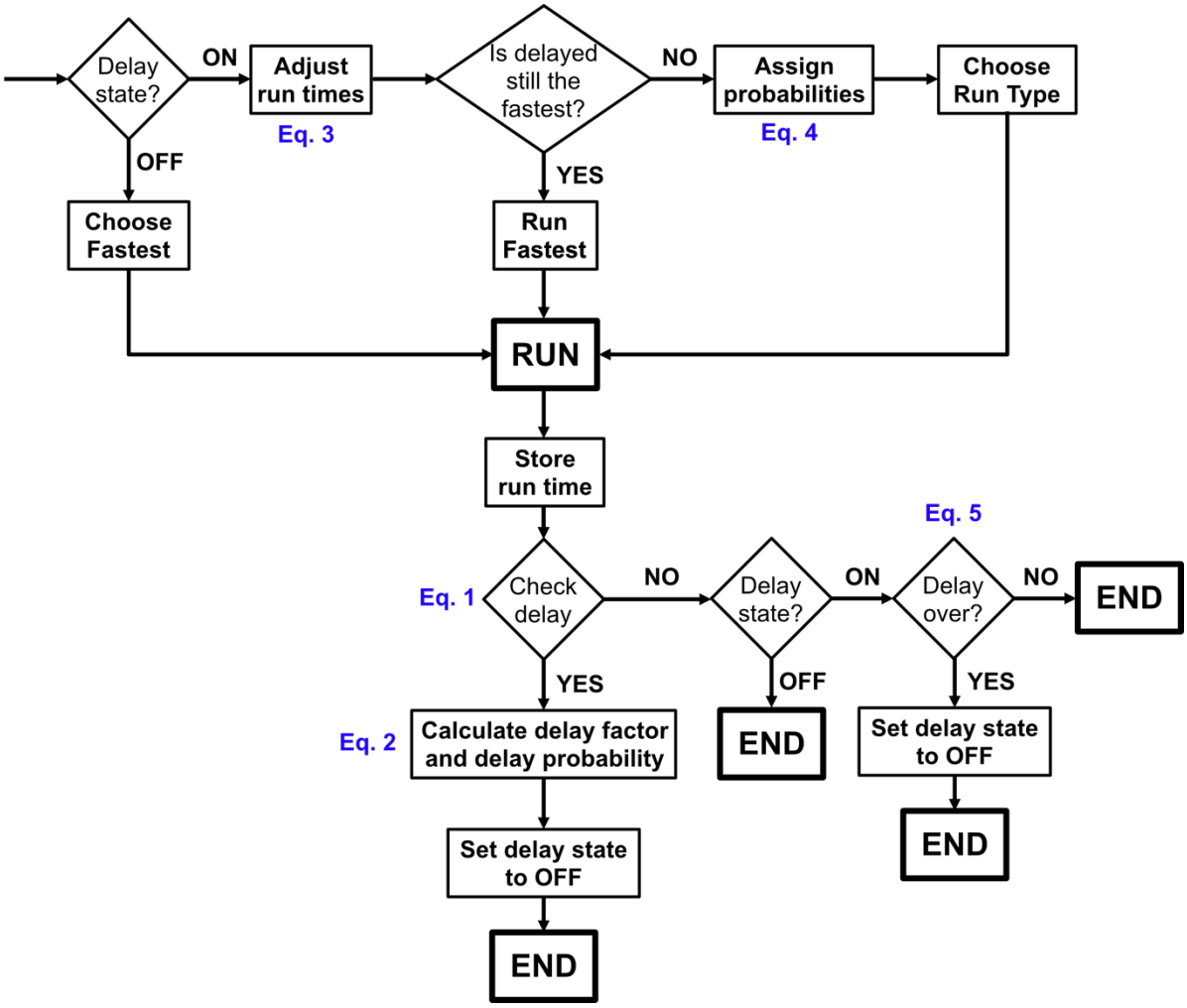
Supplementary Figure 2 – Ratio between the run times of a 2D convolution. Run times were measured across multiple input data sizes and kernel sizes using either a MacBook Air M1 (A) or a Professional Workstation (B). Areas within dashed lines correspond to kernel and image sizes where OpenCL is faster than threaded CPU.



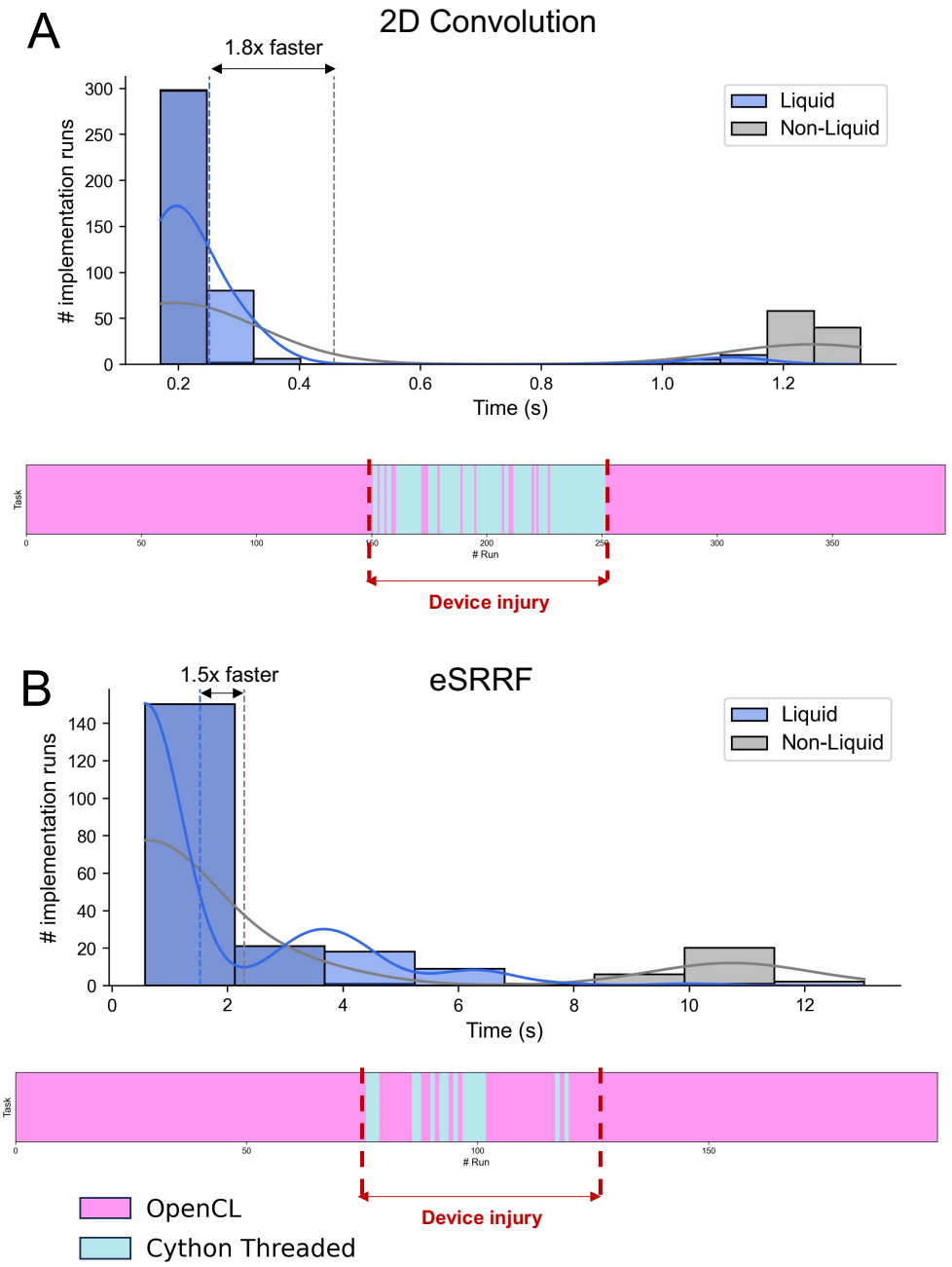
Supplementary Figure 3 – Run time of each implementation is highly dependent on the shape of input data. A 2D convolution was performed on images with increasing size using either a MacBook Air M1 (A) or a professional workstation (B). A 21 by 21 kernel was used in all operations. When using the MacBook laptop, interestingly the PyOpenCL implementation is the fastest until 125MB after which the Cython threaded implementations become significantly faster. In the professional workstation, while unthreaded is virtually always the slowest implementation, the threaded implementations are only the fastest until the size increases to 20MB, after which PyOpenCL becomes the fastest.



Supplementary Figure 4 – Kernel size impacts which implementation is the fastest. A 2D convolution was performed on images with varying kernel sizes, ranging from 1 to 21 (every 4) using either a MacBook Air M1 (A) or a professional workstation (B). A 21 by 21 kernel was used in all operations While unthreaded is virtually always the slowest implementation, the threaded implementations are only the fastest until the size increases to 20MB, after which PyOpenCL becomes the fastest. Bottom panels correspond to zoomed in windows of top panels, indicated by dotted boxes.

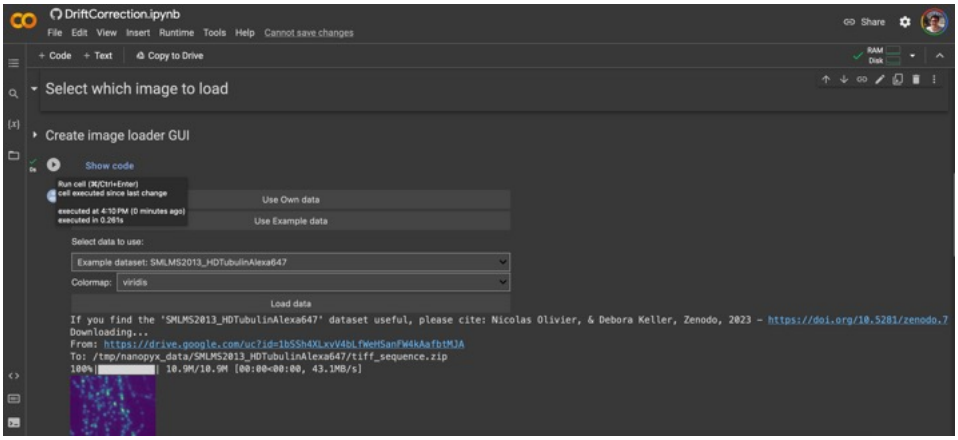


Supplementary Figure 5 – Schematic of the Agent decision making for delay management.

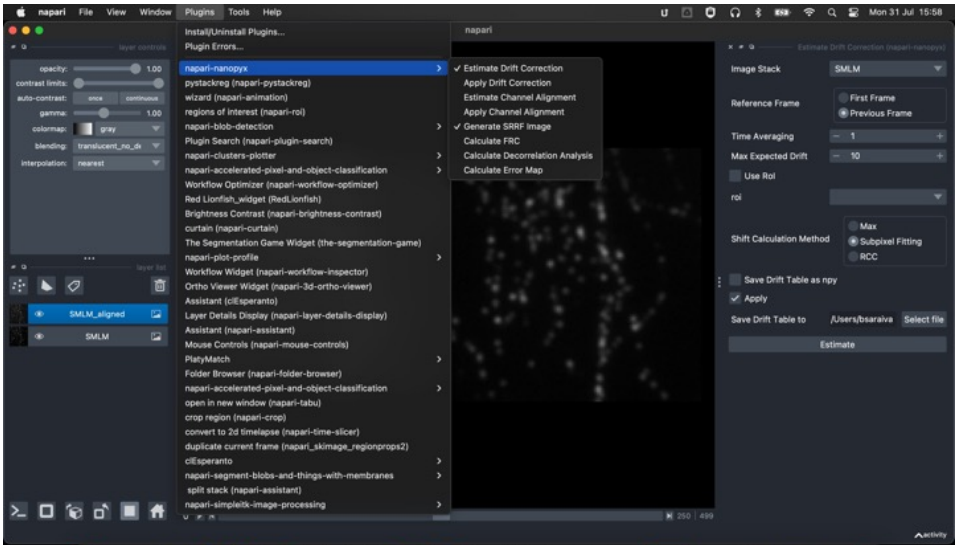


Supplementary Figure 6 – Example of delay management by the Liquid engine. Multiple two-dimensional convolutions (A) and eSRRF analysis (B) were run sequentially in a professional workstation. Starting from two initial benchmarks, the Agent is responsible for informing the Liquid Engine on the best probable implementation. An artificial delay was induced by overloading the GPU with superfluous calculations in a separate Python interpreter. Dashed lines represent average run times.

A



B



Supplementary Figure 7 – NanoPyx is available to users independently of their coding expertise. Besides using NanoPyx as a Python library, users also have access to Jupyter notebooks[13] (A) that can either be run locally or through Google Colaboratory. Additionally, users have access to a napari[14] plugin (B) with the implemented methods.

References

1. Behnel, S. *et al.* Cython: The Best of Both Worlds. *Comput. Sci. Eng.* **13**, 31–39 (2011).
2. Kloeckner, A. *et al.* PyOpenCL. (2022) doi:10.5281/zenodo.7063192.
3. Lam, S. K., Pitrou, A. & Seibert, S. Numba: a LLVM-based Python JIT compiler. in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* 1–6 (ACM, 2015). doi:10.1145/2833157.2833162.
4. Stone, J. E., Gohara, D. & Shi, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Comput. Sci. Eng.* **12**, 66–73 (2010).
5. Haase, R. *et al.* CLIJ: GPU-accelerated image processing for everyone. *Nat. Methods* **17**, 5–6 (2020).
6. Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585**, 357–362 (2020).
7. Novak, V., Perfiljeva, I. & Mockor, J. Mathematical Principles of Fuzzy Logic. in (1999). doi:10.1007/978-1-4615-5217-8.
8. Olivier, N. & Keller, D. STORM Vectashield datasets (Tubulin). (2023) doi:10.5281/zenodo.7620025.
9. Culley, S., Tosheva, K. L., Matos Pereira, P. & Henriques, R. SRRF: Universal live-cell super-resolution microscopy. *Int. J. Biochem. Cell Biol.* **101**, 74–79 (2018).
10. Gustafsson, N. *et al.* Fast live-cell conventional fluorophore nanoscopy with ImageJ through super-resolution radial fluctuations. *Nat. Commun.* **7**, 12471 (2016).
11. Gray, R. D. M. *et al.* VirusMapper: open-source nanoscale mapping of viral architecture through super-resolution microscopy. *Sci. Rep.* **6**, 29132 (2016).
12. Catmull, E. & Rom, R. A CLASS OF LOCAL INTERPOLATING SPLINES. in *Computer Aided Geometric Design* (eds. Barnhill, R. E. & Riesenfeld, R. F.) 317–326 (Academic Press, 1974). doi:10.1016/B978-0-12-079050-0.50020-5.
13. Kluyver, T. *et al.* Jupyter Notebooks – a publishing format for reproducible computational workflows. in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* 87–90 (IOS Press, 2016). doi:10.3233/978-1-61499-649-1-87.
14. Sofroniew, N. *et al.* napari: a multi-dimensional image viewer for Python. (2022) doi:10.5281/zenodo.7276432.