Re-implementation and Optimization of a Scalable Spike Detection Algorithm for Large-Scale Extracellular Recordings

Kunchangtai "Rickey" Liang

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2022

Abstract

Extracellular recordings are one of the main techniques in neuroscience to record the activity of neurons in the brain. Recent advances in engineering have made it possible to record from thousands of channels simultaneously, creating large and complex data sets. A critical step is the extraction of the activity of single neurons from these recordings, a process called spike sorting. The first step in spike sorting is the detection of events (spikes) generated by neurons, which is time consuming as large amounts of raw data have to be processed. In this project, an existing algorithms for spike detection was rewritten in an object-oriented way, and, with the help of profiling, optimised for performance on large data sets. The final version of the new package `hs-detection` achieves both a 5.2x speedup from baseline and real-time performance on large-scale recordings. In addition, it was integrated as a component into the SpikeInterface package, a toolkit that unifies access to numerous algorithms used in spike sorting.

Acknowledgements

I would like to thank my project supervisor Dr Matthias Hennig, who patiently guided me throughout the process, discussing the implementation and providing advice for writing.

I would also like to thank my parents for their mental support during my study.

Table of Contents

1 Introduction **1**

1.1 Achievements 1

2 Background **3**

2.1 Extracellular Neural Recordings 3

2.2 Typical Spike Sorting Pipelines 4

2.3 Efforts of SpikeInterface 6

3 Herding Spikes Detection **8**

3.1 Pre-processing 8

3.2 Detection Loop 9

3.3 Spike Processing 11

3.4 Implementation Details 12

4 Implementation of `hs-detection` **15**

4.1 Python/Cython Implementation 17

4.2 C++ Implementation 20

4.2.1 Speedup Gained from Design 23

4.3 Further Optimization 26

5 Experimental Results **30**

5.1 Comparison with Other Implementations 31

5.2 Analysis on Parallelization 31

5.3 Performance Issue with Median 33

6 Discussion and Conclusions **35**

6.1 Future Work 35

Bibliography **37**

Chapter 1

Introduction

Analyzing the activity of neuron cells is essential to gaining a deeper understanding of the mechanism in the brain. Several approaches are available to neuroscientists to record the activity of neurons. Among those approaches, extracellular electrical recording is advantageous due to its non-invasiveness (the recorded neurons are not damaged) and the ability to record at scale. The recording is performed by putting an electrode outside of a neuron cell (in the extracellular space), and the voltage trace represents the extracellular membrane potential of the neuron [7]. The recent development of high-density multi-electrode arrays (HD-MEAs) allows the simultaneous recording of thousands of neurons to come true, but the scale of the data is beyond what humans could process manually.

Therefore, automatic spike sorters have been developed to analyze the recordings. SpikeInterface (SI) [2] summarizes the previous works by furnishing a unified toolkit for spike sorting with access to multiple data formats and spike sorters. The next stage in the toolkit development is modularization, that is, breaking down the spike sorters into sorting components that can be arbitrarily composed into a new spike sorter. This project aims to follow this direction and take out the detection part of Herding Spikes (HS) [15] to build it into a sorting component.

1.1 Achievements

The primary output of this project is the Python package `hs-detection`, released on Python Package Index (PyPI) [1] and open-sourced on GitHub [2]. This package is

[1] `https://pypi.org/project/hs-detection/`
[2] `https://github.com/lkct/hs-detection`

positioned as an optional dependency of SI, serving as the backend of a new sorting component. It is currently under discussion before being merged into the SI code base.

The code is completely rewritten to employ the object-oriented programming (OOP) paradigm while addressing code style for better readability and maintainability for future work. The optimization has led to a 5.2x speedup compared to the original HS implementation and outperforms the existing detection component in SI by 62%. Real-time performance has been gained on the test data with the large-scale recording setting, and a further speedup of up to 2.7x can be achieved through parallelization.

Chapter 2

Background

This chapter presents the background knowledge and the motivation for this project. A broader introduction is present in the Informatics Project Proposal [12] for this project, and here we will focus on the content that relates the most to the rest of this dissertation.

2.1 Extracellular Neural Recordings

Neuron cells interact through all-or-none firing events called spikes, which appear as potential changes with a specific pattern caused by ion flows through the cell membrane. This phenomenon suggests a way to capture a neuron's activity without invasive approaches that may affect the cell's functionality: an electrode can be placed in the vicinity of the neuron to record the extracellular potential change. A typical spike's waveform is shown in Figure 2.1. The first property of extracellular recordings, as in the figure, is the significant noise level. Unlike the intracellular voltage, which is influenced only by the neuron itself, the extracellular voltage may contain the effect of several nearby neurons. Also, the activities of distant neurons and synapse current can appear as background noise [7]. However, the extracellular spikes still have specific shapes standing out from the noise. During a spike, the extracellular voltage will first deviate considerably from the baseline level, and then an afterhyperpolarization (AHP) phase follows where the voltage goes in the other direction. This biphasic characteristic is often used as a criterion to distinguish spikes from noise. In the implementation of this project, the spike shape is flipped when processed, so the threshold for spikes is positive.

Ever since the first extracellular recording in 1962 [9], people have been trying to improve the recording device to provide more precise information in a larger amount.

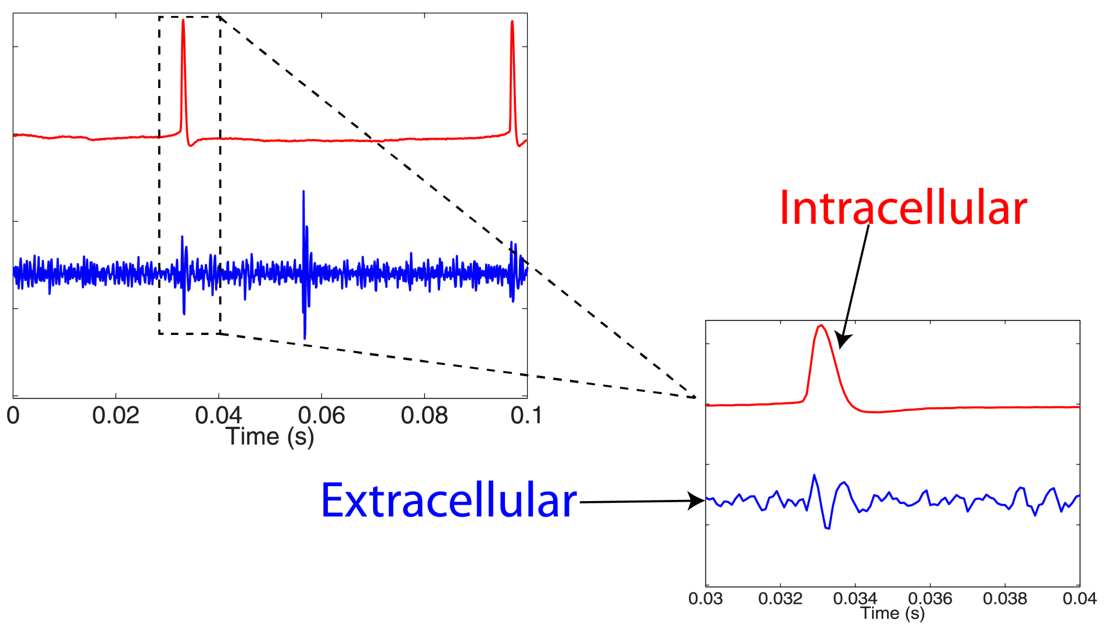


Figure 2.1: The typical waveform of a spike with both extracellular and intracellular voltages. Figure taken from [6].

With the rapid advancement of the semiconductor industry, arrays of micro-electrodes can be manufactured to record simultaneously at multiple locations (channels). Merely six years ago, people were studying “large, dense electrode arrays ” of 32 channels [18], but today, we have HD-MEA probes with 5120 channels to record in vivo [20] and near 20000-channel devices for cultured neurons [22]. The probes with an increasing number of channels are posing a challenge to data processing. For example, the Neuropixels 2.0 probe [20] can sample at 30 kHz at most, producing 14-bit raw data. This would give, on 5120 channels, a rate of 150 million data points, or 430 MB, per second (for comparison, a regular SATA SSD can work at 500MB/s). Some 1.5 TB of raw data could be generated during a typical recording experiment lasting one hour, which makes it impossible to analyze with human labour. Therefore, computer software for processing at such a large scale is in demand.

2.2 Typical Spike Sorting Pipelines

In order to identify spike activities from the recorded voltage trace, researchers have developed different algorithms to accomplish the process called spike sorting. Although the spike sorters may follow various configurations, they can generally be fitted into a typical pipelined design [4, 7]. Firstly, the raw electrical signal may contain

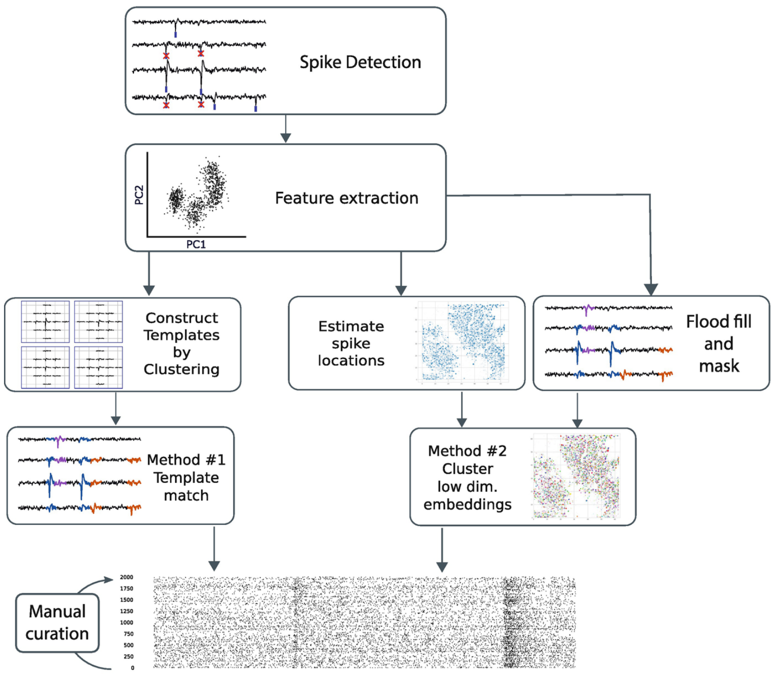


Figure 2.2: The standard structure of a spike sorting pipeline. Figure taken from [7].

noises and Local Field Potential (LFP) signals, which do not contain information about spikes and should be filtered [4]. It can be achieved with a band-pass filter, as shown in Figure 2.2. Other pre-processing modules can also be added to fit the needs of the following algorithm. Following is the spike detection module, which extracts spike candidates from the pre-processed data. As spikes always involve a notable voltage change, they can be detected by setting a threshold on the input signal [7]. The threshold can even be a dynamic value to compensate for the shift in signal-to-noise ratio (SNR). However, detecting spikes from a source relatively farther from the electrode can be tricky, as the spike amplitude can be close to that of noises. Therefore, some acceptance criteria can be established based on the shape properties of actual spikes, e.g. the existence of a biphasic shape [7], to guarantee a low false-positive rate even with a loose threshold. In addition, extra functionalities can be added to the detection module to cope with the fact that multiple channels may pick up the same spike event on an HD-MEA device. For example, in Herding Spikes [15], the detection part also includes de-duplication and spike localization.

After spike detection, the waveforms of the detected spikes are extracted from the voltage trace and projected into a low-dimensional feature space which will be fed to the clustering stage. The most commonly used method for feature projection is principal component analysis (PCA); at the same time, other approaches are also widely adopted, including wavelet decomposition, independent component analysis (ICA), and machine learning models [4, 7]. Next, the extracted features need to be clustered and assigned to neurons, and there are two major routes leading to that. The first is template matching (e.g. [21]). The spikes are first clustered in the feature space to produce a template corresponding to the siring pattern of one neuron. Then, the detected spikes are matched to the templates to be assigned to neurons. This approach can effectively handle spike collisions because the collisions can be easily identified by matching the waveform to the combination of different templates. The other way to clustering is the density-based method (e.g. [8]). The spike features are clustered by finding high-density regions separated by low-density spaces, and each region is considered yielded by the same neuron. This method can naturally incorporate spike locations (either from the localization in the detection step or an extra localization) by adding them to the spike features so that distant neurons creating similar spike patterns can be efficiently split. It is shown to be promising, especially on large-scale electrode arrays.

Aside from the traditional pipelines, the success of neural networks (NN) has led to the development of NN-based spike sorters, e.g., [11]. They either follow the same pipeline structure with functional stages implemented in NN, or build an end-to-end NN sorter with all stages fused together. The details are out of the scope of this paper.

2.3 Efforts of SpikeInterface

With different spike sorting algorithms keep emerging, a problem arises that there is no common standard to be followed by developers. Each lab uses its own probe, saves the recordings in its own file format, and develops spike sorters based on those. This has contributed to the fragmentation of the software ecosystem [2]. The issue prohibits the reproduction of results from other teams, making it tough to compare different algorithms. In this context, the SpikeInterface (SI) [2] was proposed aiming to provide a solution. SI acts as a layer of wrapper over different packages, providing a unified interface to handle recording files (`BaseRecording`) and run spike sorters (`BaseSorter`) along with various utilities for pre-/post-processing.

With the improved interoperability among different data and sorters using SI, new insights have been gained through benchmarks. The authors of SI found that different sorters mainly disagree on false-positive results, shedding light on the way to model ensembling [2]. Going further in this direction is the SpikeForest project [13]. A large amount of data were gathered with ground truth to benchmark commonly used spike sorters, and it is shown that different algorithms perform differently on different types of data. Therefore, it has become necessary to analyze the algorithms more in-depth to identify the cause of such differences.

One of the weaknesses of the old version of SI was that it was nothing more than a wrapper, treating the spike sorters as black boxes. Thus, it was challenging to analyze what was happening inside the algorithm. Fortunately, the SI code base [1] is under active development in an effort to address this issue. One of the current focuses of SI is to decompose spike sorters into some fundamental building parts. As introduced in the previous section, most spike sorters follow the same pipelined design, so the building components can be interchangeable among different pipelines [4]. Hence, SI tries to make available a set of sorting components that can be used to build new pipelines. This way, different stages of the algorithms can be compared and analyzed, enabling further understanding of the problem. However, this modularization work is still in process and demands continuous development, and the purpose of the project in this paper is to contribute a new sorting component to SI.

[1] `https://github.com/SpikeInterface/spikeinterface`

Chapter 3

Herding Spikes Detection

This chapter describes the detection algorithm in Herding Spikes (HS) [15] and its code base [1], which serves as the basis of the `hs-detection` package. Since there have been modifications and improvements after the publication of [15], this chapter will mainly refer to the code base and the wrapper in SpikeInterface (SI) [2].

The design of HS follows the typical spike sorting pipeline that can be divided into detection and clustering. Figure 3.1 shows the composition of the detection (with localization) stage in HS. Although there is no pre-processing stage in the HS code base, the wrapper in SI includes one, so it is still included in the diagram to enable opportunities for joint optimization with the rest of the algorithm. The detection stage that comes after pre-processing consists of two major parts separated by a queue of spikes. The detection loop iterates through the voltage trace looking for spike events and pushes them to the spike queue, while the spike processing part takes the spikes from the queue for de-duplication and localization and outputs the final result. The algorithmic details will be introduced in the following sections.

3.1 Pre-processing

The original HS algorithm is designed for a specific data source and does not need flexible pre-processing. However, SI has wrapped it with data pre-processors to apply it to a broader range of data. As in a typical spike sorting pipeline described in Section 2.2, a band-pass filter with configurable lower and higher cutoff frequencies is first added. Even though the HS algorithm has robustness mechanisms to cope with signal

[1]Specifically, the version at `https://github.com/mhhennig/HS2/tree/76c612f7`.

[2]Specific version of the wrapper at `https://github.com/SpikeInterface/spikeinterface/tree/280e2030/spikeinterface/sorters/herdingspikes/herdingspikes.py`.

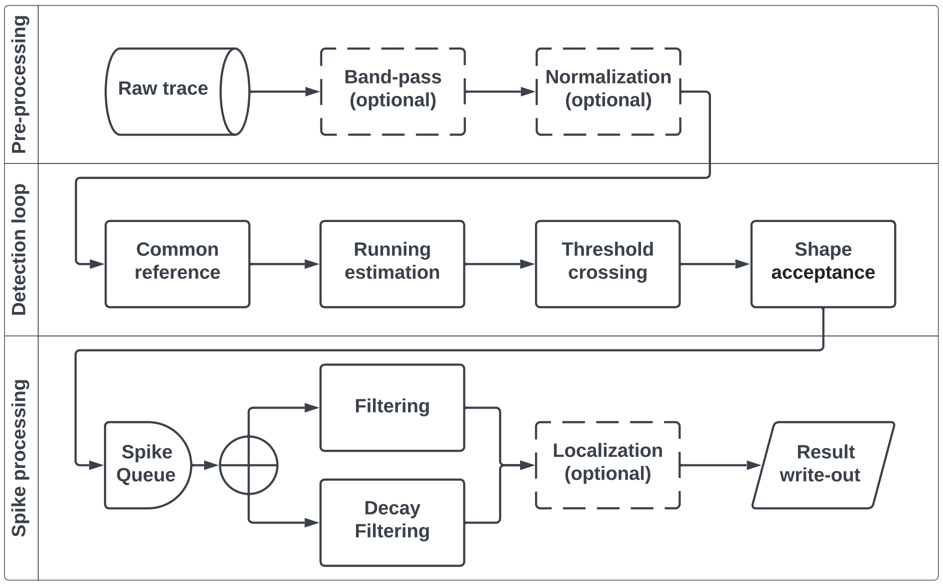


Figure 3.1: The composition of the pre-processing and detection stage in HS.

noises and voltage drifts, the band-pass filter has been proven helpful in some cases and added to the SI version.

The next point to deal with is the data range. HS expects `int16` data from proper on-probe signal amplifiers. However, it is not guaranteed that all the data will come in the same form, which could require adjustments to algorithm parameters if not rescaled. Therefore, after the band-pass filter, a normalization module will move the median of the input signal to `0` and scale the distance of the 5th and 95th percentiles to a given value. The median and percentiles are calculated from some randomly sampled chunks of the data. A type-casting to `int16` is then performed when appropriate because the calculations in the following parts can be faster than floating-point operations. Since the 16-bit dynamic range is more than enough for most neural recordings, a fixed multiplier of `64` is applied on the fly in the detection loop, providing 6 ($= \log_2 64$) bits more precision in integer division. However, aside from the efficiency issue caused by redundant scaling, it also introduces information losses as the lower 6 bits could have contained a more precise result from the normalization (if not omitted).

3.2 Detection Loop

Now that the input is in relatively good shape, it is possible to detect spike events in the trace. The detection loop iterates through the data by frames and runs through the following steps to locate spikes.

1. The common median/average reference (CMR/CAR) across all channels is subtracted from the input signal. By [15], CMR should give a better detection quality. However, in practice, CAR is preferred in the code base for a faster running speed (see also Section 5.3).

2. Now that the voltage traces of all channels are properly centred, running estimations will be generated on each channel for baseline b, which accounts for the change of local field, and deviation d, which represents the noise level. Instead of the median, the 67th percentile is chosen as the baseline level to compensate for a bias towards stronger negative fluctuations. The estimations are updated online based on the observed signal at the current frame:

   - b is increased by d/τ_b if the input exceeds $b+d$, and decreased by $d/2\tau_b$ if the input falls below $b-d$, where τ_b is time constant for b updates. This asymmetric update will provide an approximation of the 67th percentile.
   - d is increased by Δd if the signal falls between $b+d$ and $b+5d$, and decreased by the same amount if the signal either goes into $b \sim b+d$ or exceeds $b+6d$. The decrease of $6d$ prevents a large drift of d caused by spikes. A minimum value d_{min} is also set to guard the following normalization.

   After the estimations are updated for the current frame, the input voltage signal V will be normalized to $(V-b)/d$ so that the detection parameters below denote values relative to baseline and deviation.

3. With the normalized voltage trace, spikes are detected with the primary detection threshold θ. Whenever the voltage goes above this value, the corresponding channel is marked as observing a candidate firing event unless it has already been marked. However, when a higher voltage than the marked one is encountered before the event ends at τ_{spike} time after the marker, it will replace the current marker so that the marked value is always the peak of the spike, and its value is used as the amplitude of the spike.

4. For all the marked channels, the shape of the spike event will be inspected. First, there might be random fluctuations that briefly rise above θ. Therefore, the voltage is averaged from the first threshold crossing to τ_{avg} time after the peak. The

average amplitude of the signal should pass θ_{avg}. Candidates without a peak length too short will be discarded at this step. Then, afterhyperpolarization (AHP) must be present within τ_{spike} time. A voltage below θ_{AHP} must be observed to form a valid biphasic shape of the spike. Finally, the candidate events passing those criteria are taken as true spikes and pushed to the queue waiting for further processing.

3.3 Spike Processing

On HD-MEA devices, it is quite common that a spike is picked up by several nearby electrodes. Therefore, filtering is required to de-duplicate and output only one in the group of events on multiple channels. As the electrical signal from the spike decays by distance, it is a natural idea to pick the spike with the largest amplitude–closest to the source neuron–and discard the others. However, it is essential first to define the range that a spike event can affect. Given the setting of the recording experiment, it is possible to decide a radius r_{neighb} that a spike can be detected. At the same time, the transmission of the signal in the media and the processing on the probe can introduce jitter to the timing of an event, which can be estimated to be τ_{jitter}. The two parameters r_{neighb} and τ_{jitter} together define a spatial-temporal neighbourhood that a spike can reach at most. For each detected spike, the spike with maximum amplitude in the neighbourhood should be the centre of its group, and all detections in the spatial neighbourhood of the maximum spike are filtered out (note that it was found this function had a mistake in the original implementation). This is the easiest and fastest way to de-duplicate spikes. However, if there are two neurons that are very close (less than r_{neighb}) and they almost fire at the same time (within τ_{jitter}), one of the two true spikes will be filtered, which is undesirable. Thus, HS also implements another algorithm named decay filtering.

In decay filtering, the spatial neighbours of a channel are further divided into inner and outer channels, determined by a new radius parameter r_{inner}. There should only be one neuron covered by the inner neighbours, and the filtering can be done as before. For outer neighbours, transmission decay is considered. It is expected that for a duplicate spike at an outer neighbour channel, the observed amplitude should have decayed by at least a ratio of ρ compared to the maximum amplitude. To find a decaying path from the centre (maximum) spike, the minimum spanning tree algorithm is employed, with the edge defined as the inner neighbour relation. Should there be another neuron firing in the vicinity, its amplitude should stand out in the decayed amplitudes in the neighbourhood and will not be filtered. The decay filtering has the potential to achieve better results, but it has a higher algorithmic complexity and requires more effort in parameter tuning.

After the nearest channel to the true spike is identified, localization follows to interpolate a spike position more precise than merely electrode locations. By inspecting the signal amplitude on nearby channels, it is possible to obtain a better spike location based on the decaying pattern. HS assumes that the voltage roughly decays as $\sim 1/r$ to $\sim 1/r^2$, and a weighted Center of Mass (CoM) is utilized as the refined position. To decide the range of relevant channels, the concept of inner neighbours defined by r_{inner} is reused. The signals on inner neighbour channels of the current spike are barely affected by other events, and to cope with jittering, the voltage amplitude between $\pm\tau_{jitter}$ time is averaged. To avoid disturbance of the current spike to the running estimation, the voltages for averaging are re-centred on the baseline at τ_{rise} time before the peak when the voltage was yet to rise. Then, to counteract noise and other sources of fluctuations, the median of average amplitudes is subtracted from all amplitudes to form the weight. The CoM of all channel positions ignoring negative weights (i.e. half of the inner neighbours are ignored) is accepted as the result.

3.4 Implementation Details

As the final part of the chapter, this section covers not algorithmic but implementational aspects of HS because implementation details can have a great impact on code maintainability and performance.

The Herding Spikes sorter is supplied as a Python package published on PyPI [3]. It is a self-contained package with its own implementation of the classes for detection and clustering, data wrappers, along with helper functions to process electrode layout. On the other hand, in order to achieve higher efficiency, the aforementioned detection algorithms are mostly implemented in full C++ code (only pre-processing in Python but implemented in SI). The Python code interfaces with the C++ part through a layer of Cython [1] wrapper that allows Python to call C++ functions and converts between Python objects and C++ types. Some parameters of the algorithm are exposed to the Python detection class through Cython, while others are hardcoded in C++. The related parameters used in implementation are summarized in Table 3.1.

[3] https://pypi.org/project/herdingspikes/

When processing a large dataset, chunking is essential, as it is impractical to load the data into memory all at once. This involves all three layers of the code. In Python, the data wrapper needs to provide random access to the underlying data file specified by frame range. In Cython, a loop is needed to iterate through the data by a given chunk size C and invoke the C++ algorithm with the chunk. In C++, the implementation uses a rolling-array style that keeps critical information of the detection loop at the end of a chunk and carries it into the next chunk.

As of the implementation of detection in C++, the code basically observes the Procedure Programming paradigm. The code flow follows the procedure shown in Figure 3.1 and wraps each part of the algorithm in a function, with helper functions for parameters and input processing, probe geometry handling, and other self-contained functional code snippets. The parameters are saved to global variables, together with other information that needs to be shared among multiple blocks. The detected spikes are put into a structure which contains the frame and channel number where the spike comes from, its amplitude, and some containers (e.g. for the waveform) that provide information for spike processing. The processed results are finally written to a file containing the frame, channel, amplitude, position and waveform of the spikes, which is to be loaded later by the clustering stage.

Apart from setting the `-O3` compilation flag [4], the C++ code employs some tricks to accelerate the computations. Firstly, integers are used instead of floating-point numbers whenever possible, which are generally faster in computation, and the 16-bit integer (`int16`) is used for the input voltage trace, which can even better utilize the memory bandwidth and the vector cores (auto-vectorization enabled by `-O3`). To compensate for the inability to represent fractional numbers, extra scaling (e.g. the `64` factor on input) is applied to provide more precision for average (i.e. integer division) and the update of the running estimates (which would either be too coarse or need fractional numbers). Also, as division is a very expensive operation on most modern hardware, the $/d$ in normalization on running estimation is converted to $\times d$ on the thresholds. Even more, the thresholds are also passed in as a scaled value, which enables more precise adjustment of parameters but requires an extra scaling of the voltage during the comparison.

[4]For an example explanation, see `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. For a specific compiler, please refer to its documentation.

Type	Parameter	Value [a] [b] [c]	Description
Adjustable	C	100,000 fr.	The data chunk size.
	f_L	300 Hz	The lower cutoff frequency for band-pass.
	f_H	6000 Hz	The higher cutoff frequency for band-pass.
	s_{norm}	20	The desired scale of data.
	θ	10	The amplitude threshold of a spike.
	θ_{avg}	6	The threshold for average amplitude.
	θ_{AHP}	5.5 [d]	The upper threshold for AHP amplitude.
	τ_{spike}	1 ms	The spike length since the peak to expect AHP.
	τ_{avg}	0.4 ms	The time length to check average amplitude.
	τ_{jitter}	0.2 ms	The allowed jitter of spike timing.
	τ_{rise}	0.26 ms	The time for the baseline level before peak.
	r_{neighb}	90 µm	The spatial neighbour radius.
	r_{inner}	70 µm	The radius for inner neighbours.
	ρ	1.0	The expected decay ratio, should be ≤ 1.
	τ_{start}	0.3 ms	The time before peak for waveform cutout.
	τ_{end}	1.8 ms	The time after peak for waveform cutout.
Hardcoded	s_{volt}	-64	The additional scaling factor on voltage.
	τ_{base}	4 fr.	The time constant for baseline update.
	b_{init}	0	The initial value of baseline.
	d_{min}	200	The lower bound of deviation.
Magic number	s_θ	2	The scaling factor at threshold comparison.
	d_{init}	400	The initial value of deviation.
	Δd	1	The update on deviation each time.

[a] The values for adjustable parameters are the default settings in the SI wrapper.

[b] The values for voltage come with no units because they are relative to normalization.

[c] The time parameters in *ms* will be converted to *fr.* (frames) given the sampling rate.

[d] This value might be buggy as AHP is expected to be below the baseline ($\theta_{AHP} < 0$).

Table 3.1: The parameters in Herding Spikes divided into three categories. Adjustable: parameters exposed to the user. Hardcoded: parameters defined as constants in the source file. Magic number: parameters appearing in the code without explicit definition.

Chapter 4

Implementation of `hs-detection`

This chapter introduces the Python package of `hs-detection` that is developed starting from the detection part in Herding Spikes (HS) and especially highlights the differences between the original implementation and the newly developed package. The fundamental design goals of `hs-detection` are:

- Clean up the Python/Cython interface for seamless integration into SpikeInterface (SI). Since this package is dedicated to the invocation from SI, the data wrapper and utilities can be directly provided by SI in a standardized manner. The dependencies should also be minimized to make a lightweight package.

- Refactor the C++ code for an Object-Oriented Programming (OOP) design. The current implementation is Procedure-Oriented, which is naturally in line with the pipelined structure of the detection algorithm. However, the clear code flow comes with the price of opaque data flow. Therefore, the new implementation aims for an OOP design which is data-centric. Besides, wrapping data and code into classes also helps with maintenance and possible future extensions, as the data structure and the operations can be replaced as a whole without affecting other parts of the package, as long as the new class exposes the same interface. Some ideas from the Functional Programming paradigm may also be borrowed to focus the program on transformations of data.

- Optimize the implementation for better performance. As discussed before, the HS implementation has tried to improve the running speed of the algorithm. Still, there exists room for further optimization in several aspects, as shown through profiling. Moreover, the original implementation is sequential, and the opportunity for parallelization can also be explored.

An additional goal is portability, but it slightly conflicts with the above. On the software side, there is no major concern. The package does not contain any Operating System (OS) dependent code (the installation script can handle OS-dependent configurations); therefore, it should work anywhere with a Python environment and a proper C++ compiler. Although some old systems may not be shipped with a compiler supporting the C++17 standard, whose syntactic sugar is needed to improve the readability of the code, there should be a newer compiler available to all mainstream platforms. On the hardware side, optimization could become an issue. All the computational work in `hs-detection` is executed by the CPU, but different CPUs can have different computational capabilities. Although there are general optimization techniques, they usually do not lead to the best performance. Contrastingly, optimization tricks for a specific architecture can bring about more significant improvements, but they may cause troubles on another CPU. Here we assume that although the developers may be using a PC with x86, ARM or other CPUs, the large-scale computation task will be run on a dedicated server, where the x86-64 (x64) architecture is still the mainstream. Therefore, we devised a compromise that the optimization will target modern x64 systems while keeping the code runnable, though less efficient, on other platforms.

The environment for the development of `hs-detection` is characterized as follows:

- OS: Windows 10 21H2 with Ubuntu 20.04 in Windows Subsystem for Linux version 2 (WSL2).
- Language environment: Python 3.9 with CPython backend, Cython 0.29.30 [1], and g++ 10.4 supplied by conda-forge [1].
- CPU: 11th Gen Intel® Core™ i7-11800H @ 2.30GHz, featuring Willow Cove (Tiger Lake) microarchitecture [2]. It has 8 cores/16 threads and 48 KB L1-D Cache per core plus 24 MB L3 Cache shared.
- DRAM: dual-channel 32 GB DDR4-3200 SDRAM, of which 16 GB is allocated to the WSL2 Virtual Machine (VM).
- Hard drive: PCIe 4.0 SSD (note that this is mostly irrelevant because everything used can fit into the in-memory cache).

[1] `https://anaconda.org/conda-forge/cxx-compiler`.

[2] For full CPU capabilities, see `https://ark.intel.com/content/www/us/en/ark/products/213803/intel-core-i711800h-processor-24m-cache-up-to-4-60-ghz.html` and also *Ice Lake Client Microarchitecture* in [10].

In all of the following program runs with time cost measured, the band-pass filter is turned *off*, as there are running estimations to compensate for it. The normalization and localization are enabled to produce a complete result, while CAR (average) is used instead of CMR (median) and plain spike filtering instead of decay filtering for a faster speed. Further, the in-memory cache of the input data file will be warmed up before the time cost is measured, while the output file will be directed to the null device in memory so as to eliminate the impact of disk I/O speed to obtain a more reliable judgement of the algorithm.

The dataset from the experiments comes from [3], which is first introduced in SI. It contains a simulated 10-min trace of 250 neurons with independent Poisson firing events recorded using Neuropixels 1.0 (384 channels, 32 kHz). An extra Gaussian noise has been added to mimic the extracellular environment. To better fit the need of the experiment, the data is transformed and saved in the channel-dense layout with `float32` format.

4.1 Python/Cython Implementation

The development of `hs-detection` starts from the original code base of HS and goes on by gradually replacing the old pieces of code with the new implementation. A unit test is implemented and regularly executed throughout the process to guarantee that the program's behaviour does not change during the re-implementation unless there is a solid reason to do so. This way, it will be safe to apply any code changes without worrying about the correctness of detection.

The first step in the development is redesigning the Python code to provide a plainer interface to connect to SI. A the same time, we have decided to rewrite the Python part thoroughly, aiming for better readability and maintainability. It has been years since the first version of HS, and a number of new features have been added to Python with old bugs fixed. Therefore, we have based the new implementation on Python 3.9 with some helpful features. Also, it was released about two years ago, which should provide a reasonable breadth of deployment while still having years before end-of-life. Moreover, type annotations proposed in Python Enhancement Proposal 484 [19] [3] will be employed in the new implementation. Since Python is a dynamically typed language, the same variable can bind to different data types. Though it is convenient,

[3]Also incrementally enhanced after PEP-484, see `https://docs.python.org/3/library/typing.html#relevant-peps`

ambiguations may arise when working on interfaces. With type annotations, the code can hint to the developers about the type expected by an interface without adding extra runtime constraints, which can help to reduce errors without loss of convenience. Many modern Integrated Development Environments (IDEs) also have the ability to provide type hints with auto-completion.

With the basic direction in mind, we may get down to rewriting. The first thing to highlight is the `NeuralProbe` class in HS. It is used to handle the data reading from recordings of a probe as well as the geometry of the electrode layout. Most of the functionalities can now be supported by the `BaseRecording` interface in SI, which provides unified access to all kinds of data formats. Some remaining utilities around geometry can be merged into the main class of `HSDetection`. However, `hs-detection` will be used as a dependency of SI, so it cannot rely on `BaseRecording` to avoid circular dependency. This prevents the use of type annotations with `BaseRecording`. The solution here is to implement a `Protocol` class that can act as a virtual base class for `BaseRecording` without touching the inheritance hierarchy in SI. The `Protocol` can supply essential interface hints to the input data of `HSDetection`. With the `NeuralProbe` class removed, all the dependencies on data loading and visualization can then be dismissed. Now NumPy [16] is the only required dependency of `hs-detection`, which provides the infrastructure of `ndarray` (the n-dimensional array) and can also cover the calculation for geometry originally done with other packages.

The next step is about Cython. Without much work to handle, the central class `HSDetection` in Python is only a wrapper around the function in Cython with parameter parsing. Nevertheless, the Cython code is merely the entry point of the C++ extension, which converts the parameters and chunked data and the results between Python and C++ types. Hence, there is no need to have multiple layers of wrapping and scatter the code into multiple files. As it is very complicated to interface Python with C++ without the help of Cython, we may consider putting the Python code into Cython and letting Cython directly handle both parameter parsing and conversion. This is possible because Cython is a language extension of Python, and it can naturally compile Python code. Besides, there can be a minor performance gain as the code is compiled beforehand without invoking just-in-time (JIT) interpretation. Statically typed variables may also be used where applicable, saving time from dynamic binding. Another highlight of the `hs-detection` implementation is the use of pure Python mode [4] with `.pxd` augmentation and magic decorators, which allows the static analyzers in modern IDEs

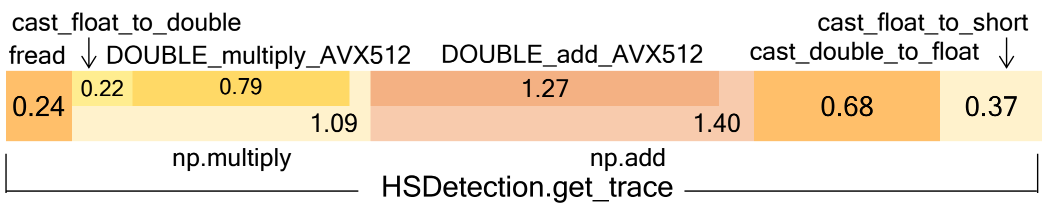
[4] `https://cython.readthedocs.io/en/stable/src/tutorial/pure.html`

to function fully for Cython code. However, there is a minor problem: even though NumPy furnishes native support for Cython, a deprecation warning is unavoidable with versioning issues (solutions only available in the unstable version of Cython).

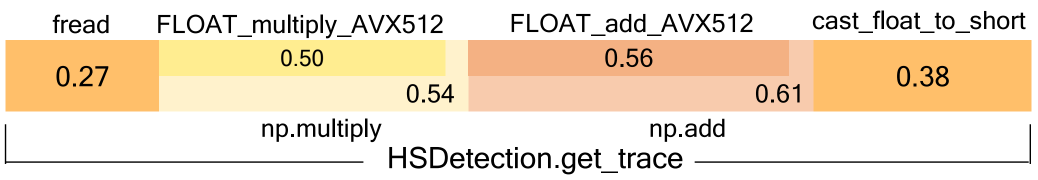
An extra enhancement added to `hs-detection` is the ability to handle multiple data segments. The `BaseRecording` class abstracts a dataset into one or more data segments recorded with the same probe settings, where each segment is a recording continuous in time. Previously, the SI directly wraps the HS package, which does not have the facility to handle multi-segment data. An auxiliary loop is needed to iterate the sorter through the segments. Now however, a `BaseRecording` is directly fed to `hs-detection`, so this functionality can be directly added to the new package.

Up to now, the changes only affect the wrappers, i.e. the part of code that does not include much actual work and does not have a noticeable influence on the performance. For guidance on performance-impacting changes, performance analysis facilities must be established first. Since most of the work now goes into pre-compiled Cython code, some function calls get hidden from Python profilers. Thus, a process profiler is chosen to profile the whole python process. Albeit the Python function/object names are not easily visible from the profiler on the system side (due to Python's dynamic binding), the most time-consuming functions are trackable from either the local C++ code or NumPy's shared library. Then, a sampling-based profiler is chosen instead of an instrumentation-based one so the profiling overhead can be reduced by the sampling rate and not impacted by the excessive number of function calls (again due to dynamic binding).

Therefore, the Intel® Vtune™ Profiler [5] is selected, which satisfies the requirements here and also suits for the following parts of the work. To balance the accuracy and overhead, the sampling rate used here is 100 samples/s. The profiling result of the `hs-detection` at the current version on 2,000k frames of data is shown in Figure 4.1a. An immediately visible problem in the result is that the data experiences several type conversions, which is definitely unnecessary with the `float32` input data. A more detailed analysis shows the conversion path in `float32->float64->float32->int16`, where the `float64` comes from `numpy.quantile` for the 5th/95th percentile needed for the scaling factor. As `float32` already provides sufficient precision, it can be used all along the way before being converted to `int16`. Therefore, instead of using the predefined pre-processor, the normalization is moved into the `hs-detection` package with `float32` calculation (apparently integer types are not appropriate for scaling). Figure 4.1b shows that this modification has approximately halved the data fetching



(a) The time costs with the pre-defined normalization in SI.



(b) The time costs with the new `float32` normalization.

Figure 4.1: The bar plots showing profiling results on 2,000k frames investigating type conversions. The lengths of bars are the proportion of CPU time taken by each function. Note that `np.multiply` and `np.add` have a small overhead the does not include actual computation.

time (`HSDetection.get_trace`).

4.2 C++ Implementation

On the C++ side, the most important objective is to facilitate the OOP paradigm. The design finally achieved in `hs-detection` is displayed in Figure 4.2. In the following, the ideas behind the design of each class are explained:

- `Detection`. This is the main class in the C++ code inherited from the original implementation, which interfaces with Cython but also runs the detection loop. In the HS code, the parameter set as in Table 3.1, along with other internal data, is scattered in global variables, class members, constants, and magic numbers. They are all cleaned up in the brand new `Detection` class. All the magic numbers are extracted from their initial places and defined as constants, which allows one-key replacement when adjusting the values. The precision scale s_θ is then increased from 2 to 256, providing 8 bits of extra precision. On the other hand, all of the adjustable parameters, as well as the internal states and buffers, are put into the member variables of `Detection` and supplied to the modules on demand, with nothing left in the globally shared scope. Therefore, it is possible to construct multiple `Detection` instances without conflicts.

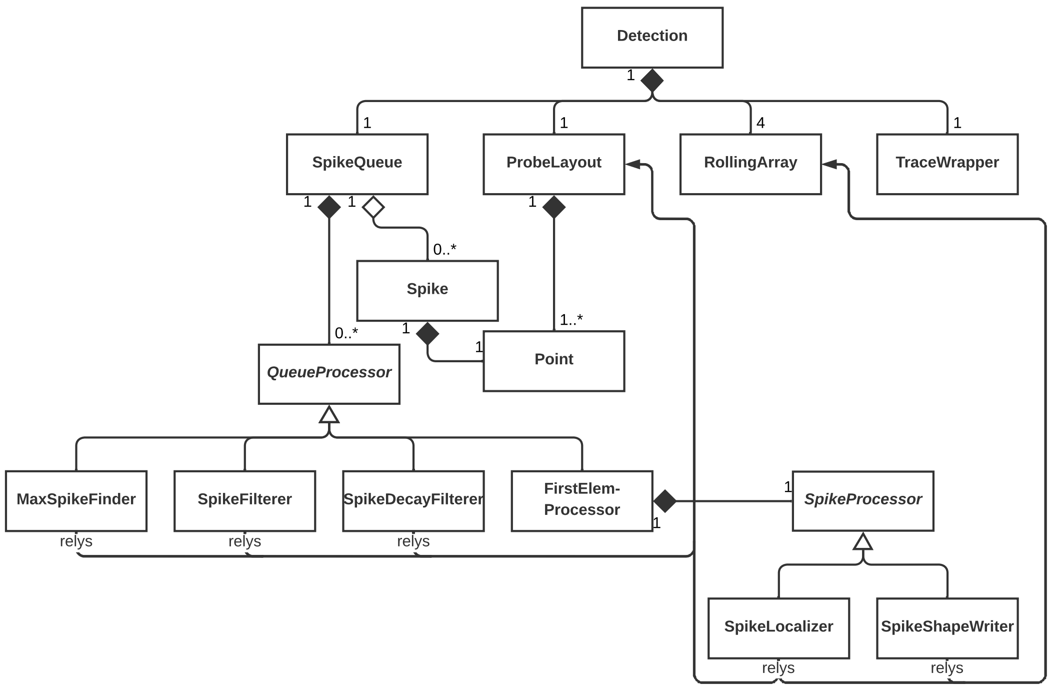


Figure 4.2: The UML class diagram for the classes and their relationship in `hs-detection`. Abstract classes are *italic*.

In addition from the parameter interface, the `Detection` class also takes care of results. The candidate events from the detection loop are pushed to the `SpikeQueue` for further processing, and the processed results are returned from the queue as a group of spikes, which will be handed back to Cython. This is different from HS kin that nothing is returned in memory in HS. Here, we assume that despite the considerable size of spike waveforms on a large dataset, it is possible to fit the metadata of spikes into memory. In cases where the waveforms are not needed, disk writing can be entirely avoided to improve performance.

- `ProbeLayout`. This class is in charge of the geometry calculations related to channel locations on the probe. One thing that we did not pay much attention to on the Python/Cython side is to retain the original `NeuralProbe` class, that is, some geometry utilities put together with parameter parsing in `HSDetection`. Now it is transferred here together with the neighbour checking implemented initially in C++ to the `ProbeLayout` class, which directly accepts the channel locations from the probe parameters in BaseRecording and hides the geometry details, and provides support for filtering and localization. Another issue worth mentioning is that the HS code base projects channel positions to integers when passing to C++, causing a slight bias (around 1 nm level) in localization, and this is fixed in `ProbeLayout`.

- `TraceWrapper`. This class wraps around the raw trace passed in as a pointer to the data chunk and also manages indexing with the global frame number during iteration through the chunks.
- `RollingArray`. This class is the container for trace-like internal data: scaled trace, common reference, and running estimations. As opposed to `TraceWrapper` which wraps a pointer from external code, `RollingArray` manages its own memory space. To carry essential information across chunk boundaries, it is a number of frames longer than the chunk size and always keeps the historical data available by updating in a rolling manner.
- `Spike`. This class contains essential information related to a spike, including frame, channel, amplitude, and position, and is instantiated when a spike is detected, with the position default-initialized waiting for localization. Different from the HS structure, the `Spike` here does not contain any extra information that can be retrieved from `RollingArray`. The lightweight spikes can help to reduce the time and space cost when working on the `SpikeQueue`.
- `SpikeQueue`. This class is a container of `Spike`s which temporarily holds the detected candidates and triggers a round of processing when an adequate amount of time has passed since the insertion of the head spike that the possible temporal neighbours are all present, and the processed result, still in the form of `Spike`s, will be handed back to `Detection` for returning. The underlying implementation employs the `list` (doubly linked list) in the C++ Standard Template Library (STL) without fast random access in favour of constant-time deletion.

  The processing in the queue borrows some ideas from Functional Programming for a paradigm centred on data transformations. the processing procedure can be viewed as a higher-order function (HOF) that takes a set of *`QueueProcessor`* as input to apply some transformations on the queue of spikes.
- *`QueueProcessor`*. This class is the abstract base for all processors working on the `SpikeQueue`. The subclass implementation should also follow the HOF design: algorithmic transformations from STL will be given a specific operator, usually implemented as `lambdas`, to finish the desired work on the queue.

- `MaxSpikeFinder`. All the processing on spike centres around the maximum spike (in amplitude) in the neighbourhood. Thus, the processing pipeline follows the design that the maximum spike is first found and moved to the front of the queue, and then all other processors will assume that the head element is the special spike as the centre.

- `SpikeFilterer/SpikeDecayFilterer`. These two classes just implement the plain spike filtering and the decay filtering described in Section 3.3. The only thing to highlight is a bug in the original implementation hidden in this sentence: *'all detections in the spatial neighbourhood of the maximum spike are filtered out'*. Unarguably, the correct implementation should use the *spatial-temporal* neighbourhood for filtering, as implemented in `hs-detection`.

- `FirstElemProcessor`. This class connects the *`SpikeProcessor`* with the *`QueueProcessor`*. In cope with the special-first-element model, a realization of *`QueueProcessor`* is needed to apply the operator defined by some *`SpikeProcessor`* to the special-first-element of the queue.

- *`SpikeProcessor`*. This class is the abstract base for the processors that only work on a single spike (normally the one that survives filtering). This, however, will be wrapped by some *`QueueProcessor`* to be uniformly invoked by the processing function of `SpikeQueue`.

- `SpikeLocalizer`. This class handles the localization of the spike in question given the `ProbeLayout`, and the signal amplitudes required are delivered by `RollingArray`. Here, unlike in the HS that returns the location from the localization module, the localized result is decoupled from the procedure and updated to the position field of the spike.

- `SpikeShapeWriter`. This class simply outputs the shape of the spike waveform to the file, which is to be loaded by other stages in the spike sorting pipeline. As mentioned before, other information about spikes is not written but kept in memory, but it is easy to also write it out without affecting other modules.

4.2.1 Speedup Gained from Design

Now that the design has been discussed, the rest of this section will cover some performance issues. As shown by the profiling, the spike processing is expected to account only for a small fraction of the time cost. Hence, the focus will be shifted to the detection loop, which has more potential for optimization.

An interesting observation is that merely sorting out the parameters can lead to a significant speedup. After careful inspection, the cause is identified to be τ_{base}, the time constant for baseline update. As the time constant, it is inevitable to be used with a division like `baseline += deviation / tauBase`. Yet as a constant parameter, it is possible to implement integer division with a power of 2 (e.g. 4) with a bit-shift, and there are more advanced tricks [14] available for arbitrary constant values that are utilized by many modern compilers (note that floating-point operations are unnecessary with the scaling of s_{volt}). However, the problem is with the `const` specifier. When used in a class, as with the original HS implementation, it specifies that this value cannot be modified after initialization. Nevertheless, it is essentially a class member bound to an instance and can be re-initialized for each instance at construction. Thus, it only results in a constant to the programmer but not the compiler and requires a true division operation at baseline update. As no division is available in the vectorized form, it also prohibits the auto-vectorization in this part conducted by `-O3` optimization. Alternatively, `constexpr` specifies the value as a constant expression at compile time. Since it cannot bind to a specific instance, the `static` keyword must be used in conjunction. Together, they specify a value as a compile-time constant, and compilers can perform all kinds of optimizations with the specific value (e.g. generate a 2-bit right shift for division by 4 and then vectorize) and remove the member from the actual class memory layout. It is worth noting that to enable auto-vectorization for the running estimations, they must be split from the loop that detects threshold crossings, as the check on spikes is too complex for vectorization. An ablation experiment is conducted using the facilities in Section 4.3 to benchmark the speed for calculating running estimations and is exhibited in Table 4.1. This is the most significant speedup gained from re-implementation except for parallelization.

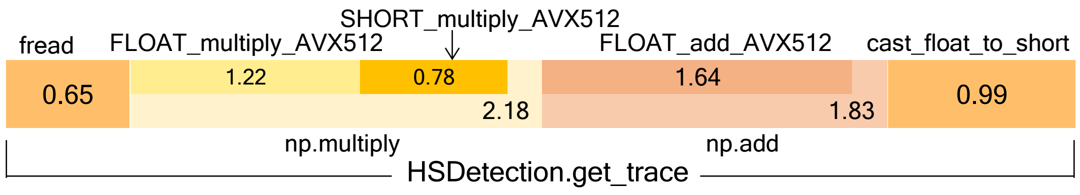
```cpp
class Detection
{
    // Original one. This is an instance member constant.
    const short tauBase = 4;

    // Updated one. This is a compile-time constant.
    static constexpr short tauBase = 4;
}
```

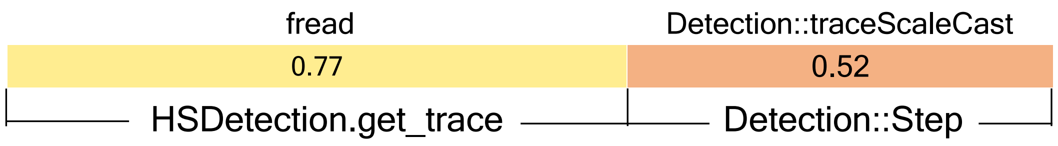
Listing 4.1: Two kinds of specifiers for `tauBase`.

Time cost of	`const`	`static constexpr`	Speedup
Running est.	161.248 s	11.899 s	13.55x
Total	224.271 s	65.304 s	3.43x

Table 4.1: Ablation study for the constant-division issue with the same settings as in Section 4.3, but not averaged on three runs because of the large margin. The total time includes approximately 3.2 s overhead related to file buffers.



(a) The time costs with Cython/NumPy implementation.



(b) The time costs with C++ implementation.

Figure 4.3: The bar plots showing profiling results on 6,000k frames for moving the normalization and quantization into C++.

From Figure 4.1b, it can be seen that multiplication, addition and type conversion from `float32` to `int16` are three standalone functions which read in the data, perform the respective operations, and write out the result. This is how a Python expression is interpreted. It can be seen in the figure that NumPy has tried to make use of the widest AVX-512 instructions to improve performance, but the three pairs of load/store introduce considerable memory access. As an alternative, a function for normalization-and-quantization can be implemented that reads in the data, performs the multiplication, addition and type conversion, and writes out the result. Another pair of bars are presented in Figure 4.3 to visualize the speedup on 6,000k frames. In addition, though with little performance impact, the global scaling s_{volt} is merged into the normalization scale s_{norm}, as they actually have exactly the same effect.

An additional improvement in C++ is the enforcement of memory alignment. The start addresses of all memory buffers are aligned to the multiple of a power of 2, and the number of channels is padded accordingly so that the address for each channel is also aligned (most probes are designed with a multiple of 64 channels, so in most cases, this has no effect). On the one hand, the address of each memory chunk in the size of a frame is aligned to the multiple of 64 bytes (the typical cache line size and the width of AVX-512), facilitating better vectorization assembly generated by the compiler and better memory access pattern in memory-dense regions. On the other hand, the start address of the large memory blocks allocated in `RollingArray`s aligns to 4 KB, the typical page size, for better paging in virtual memory. Also, the rolling length of the arrays is set to some power of 2 based on chunk size to support efficient indexing with bitwise operations.

4.3 Further Optimization

The code in `hs-detection` has been dramatically optimized through the above process, but up to now, we have only looked into some general strategies for optimization. In the final section of this chapter, we will explore some techniques to look into the microarchitecture of the CPU. Although the profiling results demonstrated in this section depend heavily on the architecture of the development platform, the optimizations taken ought to work across platforms. Finally, the possibility for parallelization will also be inspected.

A different profiling strategy from the above is adopted in this section, aiming to concentrate on possibly small improvements in the C++ code. Firstly, a C++ wrapper is developed in place of the `HSDetection` class in Cython to make the code runnable as a pure C++ program, relieving the time measurement from all the Python overheads. Secondly, the normalization factor for pre-processing is pre-computed and directly loaded when used, so the fixed overhead to sample the data and calculate the factor is eliminated. The only overhead remaining is the reading of the raw data. Therefore thirdly, each benchmark includes consecutive 10 repeated runs on the 6,000k frames (approximately 9 GB) pre-loaded into memory in order to highlight even small speed changes, and the results shown are averaged on three rounds of benchmarking after a warm-up round. The profiler is still Vtune™, which also provides event-based profiling that makes use of the hardware performance counters in the CPU to provide an extremely detailed profiling report at a low overhead. However, to grant access to the hardware events, the profiling here runs on bare-metal Windows OS instead of the WSL2 VM.

The first thought for further optimization is to look into the most time-consuming part of the algorithm, the detection loop. Through the previous changes, the so-called

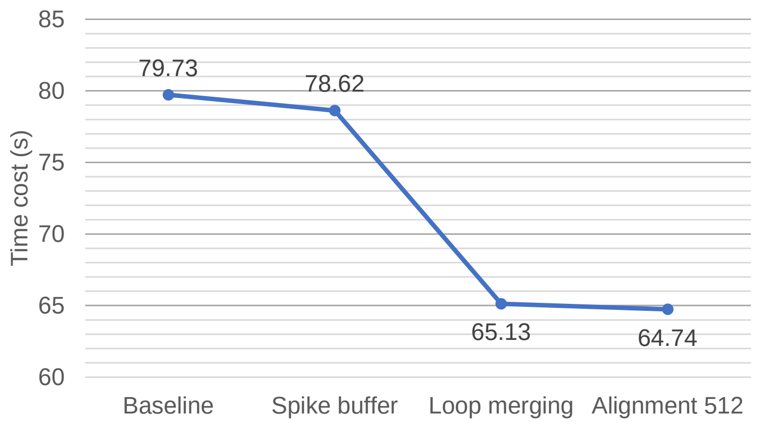


Figure 4.4: The speedup gained from inspection of microarchitecture usage. overead around 3.2s

detection loop actually contains 4 loops, each iterating through the data once. The scaling loop contains the normalization from the Python fused with scaling for precision s_{volt}; the common-reference loop calculates the CMR/CAR across all channels; the running-estimation loop is split from the main loop for better vectorization (see below, this is over-split but the vectorization brings more gain); the main detection loop does the actual work of checking threshold and shape criteria. Of the four loops, the main detection loop takes more than half of the time. Therefore, the hotspots in this loop will be analyzed for bottlenecks. Although the profiling summary does not show a large portion of the CPU time stalled by memory access (8.9% for the loop), when looking into the decomposed result for machine instructions, more than 40% of the clockticks are spent in a small group of instructions with dense memory access, which is related to the fetching of voltage trace and the pushing of spikes to queue. Based on this result, it can be speculated that there are conflicts (e.g. break of locality) between the memory access patterns of trace fetching and queue pushing, which can be proved by removing the queue pushing. A reasonable explanation is that the underlying `list` for the `SpikeQueue` needs complicated procedures to insert an element, breaking the sequential access pattern of trace reading. The solution chosen is to allocate a contiguous array to temporarily hold the detections from the detection loop and dump them into the queue later. However, to avoid reallocations that again break memory access, a huge array is needed to hold all the detected spikes from a data chunk, thus exchanging space for time. The speed achieved is illustrated in Figure 4.4.

Next, what is drawing attention is the metrics on memory. Vtune™ shows the

Version	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Memory Bound
Before	0.7%	2.1%	0.3%	0.8%	8.7%
After	0.9%	0.1% [a]	0.0% [a]	0.0% [a]	2.4%

[a] These metrics may not be reliable due to a low number of events sampled.

Table 4.2: The comparison of changes caused by loop merging in memory-related profiling metrics on the main detection loop. Note that the value of Memory Bound is not the sum of preceding values as they are calculated from different hardware events.

fraction of CPU pipeline slots wasted waiting for memory access to finish through Memory Bound, which is further broken up into L1/L2/L3/DRAM Bounds for the proportion of clockticks spent waiting for each level in the cache hierarchy. The metric values are listed in row *Before* of Table 4.2 (metrics for the main detection loop are listed as they are the most typical). The low value in L1 Bound and higher values in the following levels imply an insufficient utilization of the fastest L1 cache. A considerable ratio of memory requests even passes through the whole hierarchy and are served by the main memory. This phenomenon is plausible considering that the iteration on the lengthy voltage trace can easily outrun the caches. Similar patterns can be observed in other loops, and the solution becomes apparent: there is no necessity to read the data four times. Each frame can undergo all the processing before the next frame is read (not possible to split further as calculating the common reference requires all channels in a frame). However, for the sake of the following parallelization, the four loops are not all merged together. The implementation carried out is to merge the scaling and common-reference loops into one and the running-estimation and detection loops into the other. The vectorization is not affected as the channels in each frame still form vectors. The change in memory bounds metrics for this optimization is compared in Table 4.2. Moreover, one sub-metric under the L1 Bound, 4K Aliasing, suggests that it may not be the best practice to align several large blocks to the 4K boundary as in `RollingArrays`, which may cause conflicts in the L1 Cache. However, Vtune™ also hints that it is usually not a major problem with optimized hardware. Yet it can still deliver a minor speedup in the memory-dense regions. Therefore, a smaller alignment of 512 B is still tested, and it is shown in Figure 4.4 that there is indeed a slight performance gain.

As the final effort for additional speedup, parallelization is applied to the loops using the multi-threading feature of OpenMP [17] built-in to most compilers. One option is to parallelize everything by channel since the running estimation and detection are time-dependent. Nevertheless, the common reference must be calculated on all channels and introduces a serialized region with two additional synchronizations. Thus, an alternative is taken for full parallelization with only one synchronization point added, that is, to parallelize the scaling and common reference by time and the common reference and detection by channel. This is why we do not merge everything into one loop in the previous step. As a side effect of parallelization, insertion to the `SpikeQueue` needs modifications. Some spike processors on the queue count on the fact that the spikes are detected and pushed to the queue in time order, but it is not guaranteed because the thread for part of the channels may run faster than others. Luckily, the temporary linear buffer for the spikes introduced before can be sorted before dumping its contents into the queue. However, it requires thread safety through atomic operations when saving the spikes to the buffer. Fortunately, as spikes are expected to occur sparsely, the extra atomic locking and sorting will not cost too much overhead.

Chapter 5

Experimental Results

This chapter presents the results achieved by the implementation of the `hs-detection` package. From the previous chapter, we can already conclude that the goals of the coding style have been met. Therefore, this chapter will focus on the analysis of performance. Although memory consumption is also a crucial criterion for evaluating a program, it is not emphasized here because we have exchanged space for time during optimization.

Here are the baseline settings of the experiments: The band-pass filter is turned off as its time consumption is irrelevant to the detection algorithm, while the scaling (normalization in previous versions) is on for the general use case. Nextly, CAR (common average reference) is chosen over CMR (common median reference) to save time and facilitate a valid benchmark (further analyzed in Section 5.3). On the spike processing side, simple spike filtering is used instead of decay filtering, but they have only negligible differences in speed and filter ratio (the number of output spikes can influence processing time in subsequent modules). The localization and waveform write-out processors are both turned on for a complete result, but the output file is redirected to the null device to reduce disk I/O cost. However, other parameter settings are all kept default due to the lack of tools for parameter tuning.

The dataset for the experiment has been introduced in Section 2.1. Aside from the original form in 384 channels, it is also transformed into a 6144-channel (=384$\times$16) recording in the following way: the whole length of the original recording is sliced into 16 chunks, and they are concatenated in the channel dimension. The geometry of the probe is copied and translated to other locations. This transformed recording can serve as an example of the large-scale data produced by modern HD-MEAs.

The experiments are conducted on a computational server instead of the development machine since we expect the processing of large-scale data will take place on servers that may not be equipped with the most up-to-date hardware (e.g. with the AVX-512 instructions). The platform used, which should be representative for most users, is specified below:

- OS: Scientific Linux 7.9 (access to hardware performance counters disabled).
- CPU: 2x Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz, featuring Haswell microarchitecture [1]. Each CPU has 12 cores/24 threads and 32 KB L1-D Cache per core plus 30 MB L3 Cache shared.
- DRAM: 64 GB DDR4 SDRAM.
- Hard Drive: rotational HDD (note that the main memory is large enough to fit the cache of the whole dataset).

5.1 Comparison with Other Implementations

To demonstrate the performance gain through the re-implementation based on the Herding Spikes (HS) code base, we benchmark the output of each section in Chapter 4 and compare them with the original HS implementation. Also, to get an understanding of how the program performs compared to other algorithms, the `peak_detection` from the `sortingcomponents` in SpikeInterface (SI) is taken, with the default parameter settings [2] (not enabling spike localization due to mechanism issues). All of the algorithms are invoked form the user-faced wrapper functions in SI. The result on the 384-channel data is illustrated in Figure 5.1, and `hs-detection` offers a 5.2x speedup compared to HS and outperforms `peak_detection` by 62% even with an extra localization module.

5.2 Analysis on Parallelization

To test the scalability of the parallel implementation of `hs-deteciton`, we run the pure C++ mode on the multi-core test platform with the 6144-channel data with the

[1] For full CPU capabilities, see https://ark.intel.com/content/www/us/en/ark/products/81908/intel-xeon-processor-e52680-v3-30m-cache-2-50-ghz.html and also *Haswell Microarchitecture* in [10].

[2] https://github.com/SpikeInterface/spikeinterface/blob/e17d2ff4/spikeinterface/sortingcomponents/tests/test_peak_detection.py#L30-L32

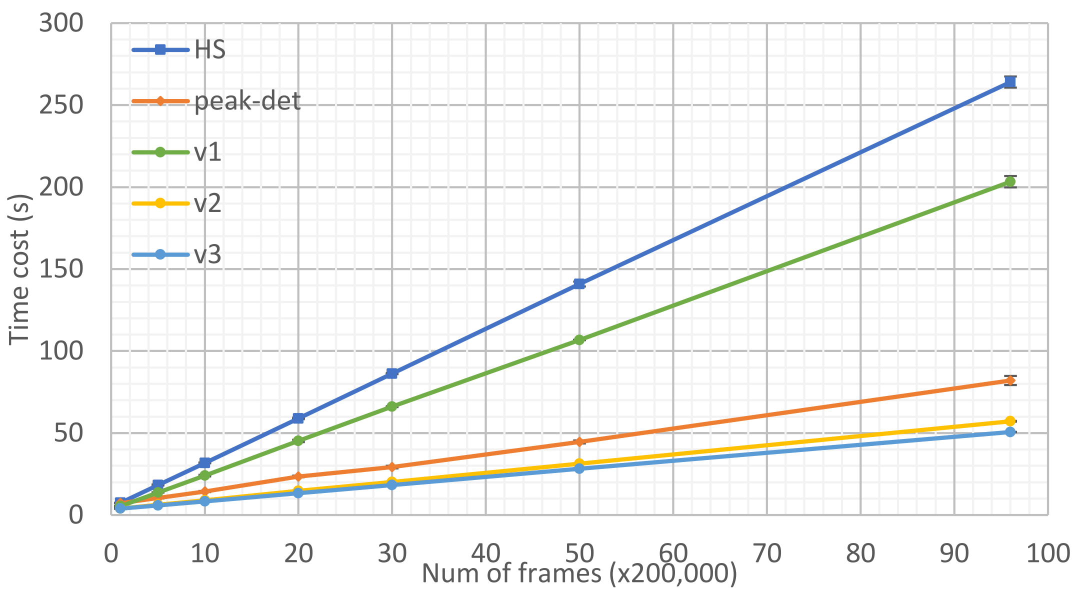


Figure 5.1: The performance comparison among versions of `hs-detection` (v1-v3 for Section 4.1-4.3), Herding Spikes (HS), and SI's `peak_detection` (peak-det). The data is in 384 channels with a sampling rate of 32 kHz.

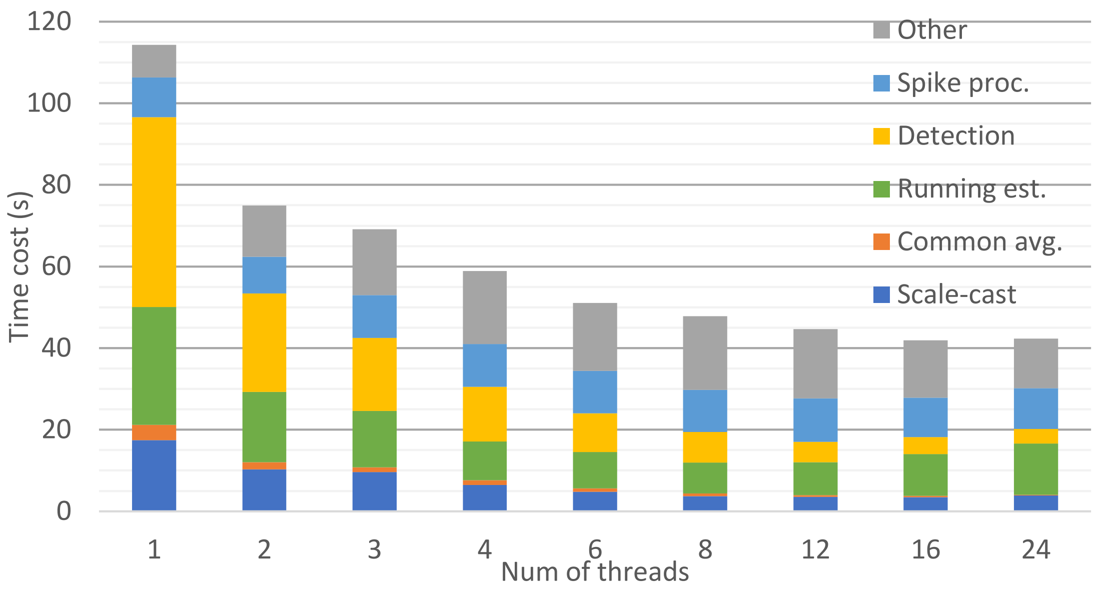


Figure 5.2: The scalability of `hs-deteciton` on multiple cores. The *Other* category includes data loading and some initialization/finalization work. The data has 6144 channels at a 32 kHz sampling rate. The corresponding real-time speed is at 112.5 s time cost.

pre-computed normalization factor to save initialization time. To further reduce the time taken by file loading, only 120k frames (10%) of the test data are used, repeated 30 times. The equivalent recording length is 112.5 s. The number of threads utilized ranges from 1 to 24, and the corresponding time costs are depicted in Figure 5.2. It can be seen clearly that the speedup quickly saturates, with the best performance only of 2.7x achieved with 16 cores. Among the major components displayed in the figure, the main detection loop scales the best, while the scaling/type-casting and running estimation can degrade with too many cores. Those two parts are among the most memory-dense regions and cannot benefit from loop merging to hide the memory latency. On the development platform where hardware performance events are available, profiling results show that the memory bandwidth would quickly saturate with only a few threads, which would begin to compete for resources. Adjusting the chunk size parameter may help resolve this issue by improving caching yet introducing more chunking overhead. However, when it comes to the demand for real-time processing, it can be approximated even with only one core available, and four to six cores will be sufficient to serve a doubled number of channels with the same sampling rate.

5.3 Performance Issue with Median

It has been mentioned before the common median reference (CMR) is too slow to be useful, and the profiling report from Vtune™ in Figure 5.3 shows why. The algorithm to calculate median (`nth_element` in C++ STL) is actually a partial sort, which includes a bunch of conditions and irregular memory access. In a modern super-scalar CPU with out-of-order execution, the inability to predict the next instruction to execute significantly harms performance. As shown in the figure, a lot of instructions executed are not retired (Bas Speculation), wasting resources on useless work. Meanwhile, each branch misprediction requires a re-steer of instruction fetching, causing the instruction decoding to perform poorly (Front-End Bound).

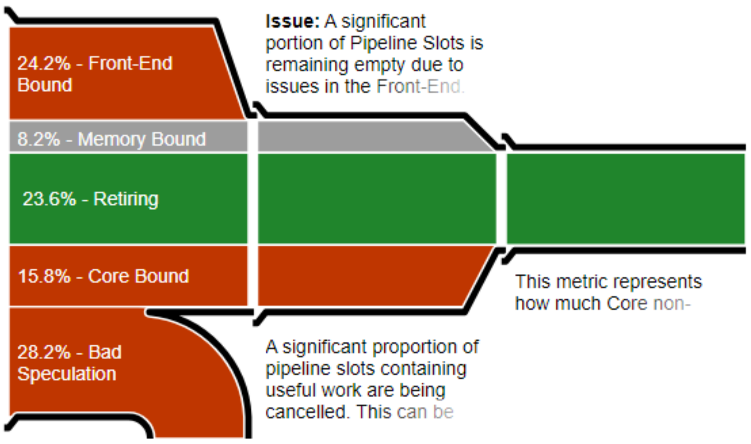

24.2% - Front-End Bound
8.2% - Memory Bound
23.6% - Retiring
15.8% - Core Bound
28.2% - Bad Speculation
Issue: A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.
This metric represents how much Core non-
A significant proportion of pipeline slots containing useful work are being cancelled. This can be

Figure 5.3: The CPU pipe diagram showing the microarchitecture usage with CMR. Interpretation of the diagram: this shows the inefficiencies in the CPU pipeline, just like a flow going through a pipe. Flowing in from the left is the full capability of the CPU pipeline slots. The pipe gets narrowed when some slots are not doing useful work, e.g., flushed due to branch misses or waiting for hardware resources. The output pipe to the right represents actual instructions executed by the CPU. For details on the five components, see *Top-down Microarchitecture Analysis Method (TMAM)* in [10].

Chapter 6

Discussion and Conclusions

In this project, we have taken the detection algorithm from Herding Spikes (HS) and re-implemented it into a new package `hs-detection`. We have set the following goals: 1. seamless integration into SpikeInterface (SI) as a sorting component, 2. OOP design in the C++ code, and 3. performance optimization and parallelization. To address the first two goals, we have entirely rewritten the Python/Cython and C++ code starting from the HS code base while keeping code readability and maintainability in mind. An additional outcome is that some bugs in the original implementation are identified and fixed in the new package.

As to the performance, several opportunities are exploited to make full use of the popular x86-64 architecture while preserving compatibility on other platforms. The final version has gained more than 5x speedup compared to the original implementation and is also 60% faster than the existing detection component in SI. With the outstanding processing speed, the algorithm can achieve real-time detection on the large-scale dataset with 6144 channels and a 32 kHz sampling rate, i.e., near 200 million data points per second, with a typical computational server. Although the implementation cannot scale linearly with multi-threading, a further 2x speedup may be obtained with up to 6 cores.

6.1 Future Work

With the achievements stated above, we can conclude that the established goals have been accomplished. However, it is not yet the time to celebrate, as not all problems are solved by this single piece of work. We expect the output of this project could serve as a new foundation for future works in different directions.

One possible direction in contributing to the modularization work in SI is to develop tools to evaluate the detection (possibly with localization) result. All existing evaluation tools for spike sorting apply to the whole sorting pipeline. Nevertheless, when decomposing spike sorters into components, testing the quality of results produced by each module is of vital importance. The research community demands such a tool that helps understand the pros and cons of different algorithms.

Another interesting track is the development of real-time, online spike sorters. In most scenarios today, the recording generated by a probe is stored in files waiting to be analyzed later. However, with the continuing development of HD-MEA devices that can record on a larger and larger scale, the space and time spent to save the raw data keep increasing. Therefore, it will be helpful if the data recorded can be processed before being saved to external storage. Some software is needed for this job, designed to accept the data stream from the recording device, process it on-the-fly, and write out the results. Although `hs-detection` is specialized for processing the data that resides readily in the memory, its real-time speed makes it suitable as the starting point of such an online toolkit.

There are also some specific points leading to further optimization of `hs-detection`:

- The current parallelization model in `hs-detection` only applies to the loops on the voltage trace due to strong data dependency in spike processing. However, the processing does not necessarily need to be performed by the main thread and can be dispatched to worker threads given enough cores. Here the producer/consumer model can be employed to be built upon the `SpikeQueue`. The threads for the detection loops act as producers that put the detected spikes into the queue, while the workers for spike processing fetch the spikes asynchronously from the queue to process them. With proper workload balancing, it can achieve better parallelism than the current implementation.

- Computational accelerators (e.g. GPUs) can sometimes facilitate better parallelism with suitable task characteristics. Most of the detection loops can be fully parallelized by channel, and the channel-dependent common reference may have an efficient implementation with collective operations on many-core systems. Therefore, it is possible to assign each channel to a core in a many-core processor, and it only needs to handle simple computations. Moreover, many-core platforms often supply a higher bandwidth to memory, exactly solving the memory starving issue with the parallelized `hs-detection`.

Bibliography

[1] Robert Bradshaw et al. *Cython: C-Extensions for Python*. Version 0.29.30. May 2022. URL: https://cython.org/.

[2] Alessio P Buccino et al. "SpikeInterface, a unified framework for spike sorting". In: *eLife* 9 (Nov. 2020). Ed. by Laura L Colgin, Sonja Grün, and Fabian Kloosterman, e61834. ISSN: 2050-084X. DOI: 10.7554/eLife.61834.

[3] Alessio P. Buccino et al. *sub-MEAREC-250neuron-Neuropixels*. 2021. URL: https://dandiarchive.org/dandiset/000034/0.211030.0713/files?location=sub-MEAREC-250neuron-Neuropixels.

[4] Alessio P. Buccino, Samuel Garcia, and Pierre Yger. "Spike sorting: new trends and challenges of the era of high-density probes". In: *Progress in Biomedical Engineering* 4.2 (Apr. 2022), p. 022005. DOI: 10.1088/2516-1091/ac6b96.

[5] *Fix Performance Bottlenecks with Intel® VTune™ Profiler*. Version 2022.2.0. Intel Corporation, Mar. 2022. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[6] Franz Hamilton, Tyrus Berry, and Timothy Sauer. "Tracking intracellular dynamics through extracellular measurements". In: *PLOS ONE* 13.10 (Oct. 2018), pp. 1–13. DOI: 10.1371/journal.pone.0205031.

[7] Matthias H. Hennig, Cole Hurwitz, and Martino Sorbaro. "Scaling Spike Detection and Sorting for Next-Generation Electrophysiology". In: *In Vitro Neuronal Networks: From Culturing Methods to Neuro-Technological Applications*. Ed. by Michela Chiappalone, Valentina Pasquale, and Monica Frega. Cham: Springer International Publishing, 2019, pp. 171–184. ISBN: 978-3-030-11135-9. DOI: 10.1007/978-3-030-11135-9_7.

[8] Gerrit Hilgen et al. "Unsupervised Spike Sorting for Large-Scale, High-Density Multielectrode Arrays". In: *Cell Reports* 18.10 (2017), pp. 2521–2532. ISSN: 2211-1247. DOI: https://doi.org/10.1016/j.celrep.2017.02.038.

[9] D. H. Hubel and T. N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of Physiology* 160.1 (1962), pp. 106–154. DOI: 10.1113/jphysiol.1962.sp006837.

[10] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Version 248966-045. Intel Corporation. Feb. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[11] Jin Hyung Lee et al. "YASS: Yet Another Spike Sorter". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/1943102704f8f8f3302c2b730728e023-Paper.pdf.

[12] Rickey K. Liang. "Towards Modularized Framework for Spike Sorting". Informatics Project Proposal. Apr. 2022.

[13] Jeremy Magland et al. "SpikeForest, reproducible web-facing ground-truth validation of automated neural spike sorters". In: *eLife* 9 (May 2020). Ed. by Markus Meister, Ronald L Calabrese, and Markus Meister, e55167. ISSN: 2050-084X. DOI: 10.7554/eLife.55167.

[14] Niels Möller and Torbjorn Granlund. "Improved Division by Invariant Integers". In: *IEEE Transactions on Computers* 60.2 (Feb. 2011), pp. 165–175. ISSN: 1557-9956. DOI: 10.1109/TC.2010.143.

[15] Jens-Oliver Muthmann et al. "Spike Detection for Large Neural Populations Using High Density Multielectrode Arrays". In: *Frontiers in Neuroinformatics* 9 (2015). ISSN: 1662-5196. DOI: 10.3389/fninf.2015.00028.

[16] Travis E. Oliphant et al. *NumPy: The fundamental package for scientific computing with Python*. Version 1.21.6. Apr. 2022. URL: https://numpy.org/.

[17] *OpenMP Application Programming Interface*. Version 4.5. OpenMP Architecture Review Board, Nov. 2015. URL: https://www.openmp.org/.

[18] Cyrille Rossant et al. "Spike sorting for large, dense electrode arrays". In: *Nature Neuroscience* 19.4 (Apr. 2016), pp. 634–641. ISSN: 1546-1726. DOI: 10.1038/nn.4268.

[19] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints*. standard. Python Enhancement Proposals, 2015. URL: https://peps.python.org/pep-0484/.

[20] Nicholas A. Steinmetz et al. “Neuropixels 2.0: A miniaturized high-density probe for stable, long-term brain recordings”. In: *Science* 372.6539 (2021), eabf4588. DOI: 10.1126/science.abf4588.

[21] Carsen Stringer et al. “Spontaneous behaviors drive multidimensional, brain-wide activity”. In: *Science* 364.6437 (2019), eaav7893. DOI: 10.1126/science.aav7893.

[22] Xinyue Yuan et al. “Versatile live-cell activity analysis platform for characterization of neuronal dynamics at single-cell and network level”. In: *Nature Communications* 11.1 (Sept. 2020), p. 4854. ISSN: 2041-1723. DOI: 10.1038/s41467-020-18620-4.