Parallel implementation of a spike detection method for high density electrophysiological recordings

Albert Puente Encinas

Master of Science

Cognitive Science

School of Informatics

University of Edinburgh

2016

Abstract

Neuronal electrophysiology is paramount to the understanding of brain function and its underlying mechanisms. However, obtaining reliable and rich data of the electrical activity of multiple neurons has not been feasible until very recently thanks to the use of dense micro-electrode arrays (MEA), allowing to record massive amounts of multi-channel electrical data extracellularly. Due to the increased spatial and temporal resolution achieved by such platforms, the amount of data that spike (action potentials originated in neurons) detection algorithms has to process has become challenging to be efficiently processed.

This project presents the parallel implementation of two spike detection algorithms introduced by Muthmann et al. (2015) using modern *C++* features and the new standard for heterogeneous computing SYCL, a cutting-edge OpenCL abstraction layer.

The developed software yields a quasi-linear scaling of the processing time when comparing sequential, 2-threaded, and 4-threaded executions. Furthermore, the implementation is capable of real-time processing of the data on both algorithms.

Acknowledgements

I would first like to thank my thesis supervisor Dr Matthias Hennig (University of Edinburgh) for all the support, encouragement and trust I have received.

I would also like to extend my most sincere gratitude to Jens-Oliver Muthmann (Manipal University, India), Martino Sorbaro (University of Edinburgh) and Duncan McBain (Codeplay®) for the selfless advices that help this venture reach its destination.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Albert Puente Encinas*)

Table of Contents

Chapter 1

Introduction

One of the keys to understanding the underlying mechanisms of the brain and provide accurate models of its functioning is to obtain reliable and extensive data. At the heart of the modelling resides the field of neural coding, concerned with defining the relationship that exists between stimulus and the neuronal response in the form of electrical activity, ultimately represented by action potentials (i.e. spikes) of neurons (see section 2.1). In order to obtain information about the spikes from neuronal tissue, electrophysiologists have historically relied on single or simple multichannel recordings. Amongst these methods, the most common are the intracellular recordings such as the patch clamp technique (accurate but limited to single neurons), or the extracellular positioning of multiple electrodes together (e.g. polytrodes, Blanche et al., 2005), which dramatically improves the task of identifying spikes (Gray et al., 1995; Harris et al., 2000).

During the last decades, recent advances in miniaturisation and material engineering have allowed high-density micro-electrode arrays (MEA) to become a reality. MAEs allow to record a large number of channels simultaneously (see Figure 1.1), a number that has been exponentially increasing over the latest years. However, as the number and density of channels increases, the algorithms which process the signals have to deal with quadratically larger and more complex data. Indeed, the algorithms responsible for identifying action potentials have to handle noisy, massive and potentially redundant[1] data.

[1]The dense disposition of the electrodes (e.g. 42 - 81 μm for the 3BRAIN BioChip 4096 series) in MAEs allows to capture the signals of individual neurons on multiple channels.

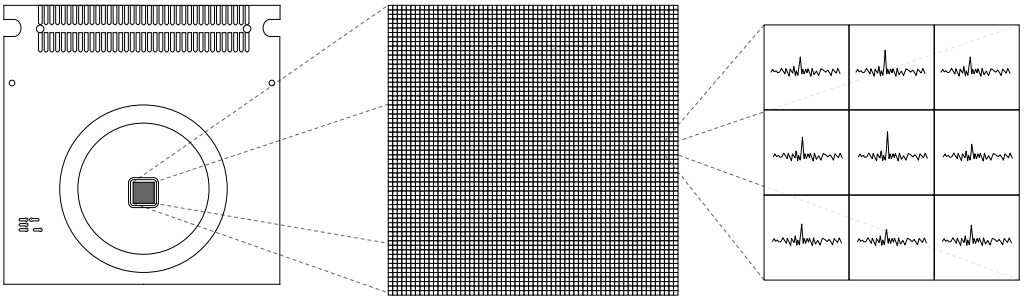

Figure 1.1: Schematic of a micro-electrode array (approx. BioCam 4096 by 3Brain Inc.) with a CMOS sensor, similar in principle to the one used in digital cameras. After placing neuronal tissue in a compatible solution, the chip is capable of recording 4096 electrical signals positioned in a 64x64 grid of channels with a total working area of 3x3 mm. The right diagram shows different signals for a unique spike (i.e. depolarisation).

1.1 Motivation

Modern MAE platforms used in research have a particularly high sampling rate frequency. Using fast recordings together with large sets of channels can provide a better discrimination of spikes but can also yield massive bandwidths[2]. In Muthmann et al. (2015), the authors developed a set of algorithms aimed at performing the task of spike detection (finding individual spikes on data) and spike sorting (assigning the detected spikes to their source neurons). The efficiency of the first one is of a special importance. Since the amount of memory required to store spikes is immensely smaller compared to the memory required to store whole recordings, it becomes crucial to process the input signals fast enough so that spikes can be stored in real-time. This allows not only to handle daylong recordings but also empower the implementation of solutions which provide real-time feedback within the experimental steps. Specifically, the paper introduces two algorithms: an online algorithm that performs spike detection on each channel independently, and an interpolation-based algorithm that uses multichannel data to improve the detection sensitivity and spatial accuracy (see chapter 3 for an outline of the algorithms).

It is at this point that modern parallel computer architectures offer a scalable and rela-

[2]Frequencies of approximately 7 KHz for 4096 channels with a resolution of 12 bits per sample (3Brain BioCam4096S) generate an output of 2.5 GB/minute. The newest systems by 3Brain can record using sampling frequencies up to 18 KHz.

tively simple solution to the aforementioned problems. Multi-core architectures such as modern CPUs or GPUs provide a way of simultaneously executing programs on multiple threads provided the implementation coordinates their work flow. The degree of parallelistation that can ultimately be achieved (speed-up) is usually linked to the constitution and data dependencies of the algorithms as well as the the kind of platform in which it is executed. For this reason, and in search of a thorough analysis of the parallelisation capabilities of the algorithms explained by Muthmann et al., this project details the development and assessment of parallel implementations in multiple forms. Specifically, the developed software has employed the parallelisation features of modern *C++* and the new Single-source Heterogeneous Programming (SYCL[3]) based on OpenCL (see section 2.4).

1.2 Goals of the project

The objective of this MSc thesis is to research and implement a parallel version of the algorithms explained in Muthmann et al. (2015) (chapter 3), which can be executed in heterogeneous platforms (CPUs or GPUs). In detail, this can be separated into two major tasks:

1. Implement a working parallel version of the online detection algorithm and detail the corresponding speed-up.
2. Research and study how the interpolation-based spike detection algorithm can or cannot be parallelised due to its intrinsic structure and explain the reasons as well as explore feasible solutions.

In addition, documenting any findings regarding the applicability of parallel architectures to spike detection algorithms is also a part of the scope of the project.

1.3 Achieved results

Overall, the project has produced several parallel implementations for the online spike detection algorithm, with a maximum parallel speed-up of approximately x5 compared

[3]The Khronos Group Inc. *SYCL - C Single-source Heterogeneous Programming for OpenCL* `https://www.khronos.org/sycl`

to a new sequential implementation in *C++*. When comparing with the previous implementation of the algorithm in *C#*, the total speed-up is of approximately 20x.

Regarding the interpolation-based spike detection algorithm, a sequential port to *C++* has yielded a speed-up of approximately 3x comparing with the original *C#*. In addition, a new parallel implementation designed as a proof of concept, replicating the base functionality of the original code but optimised for a parallel execution capable of performing the interpolation-based algorithm in real time, has been developed and tested. For more details, please examine chapter 5.

Chapter 2

Context and background research

2.1 Neuronal electrophysiology

Neurons are arguably the most important cells concerning the processing and transmission of information in almost all animals. Their electrical and chemical properties provide them with the ability to react to potentials sent by other neurons and transmit signals in the form of synapses, which can be electrical or chemical (Purves et al., 2001). In the chemical synapse, the most common, different kinds of neurotransmitters (e.g. amino acids) released at the end of the axon of another cell trigger the flow of ions (mainly, Na+, K+, Ca2+ and Cl-) across ion channels which, in turn, create voltage fluctuations in the soma. After the combination of both excitatory and inhibitory synapses make the voltage of the post-synaptic cell exceed a certain threshold, a large depolarisation (i.e. a spike) is unchained and sent along the axon to trigger another synapse at its end.

Analysing where and when spikes are triggered can yield invaluable information on how the brain encodes and decodes stimuli in the form of spikes (Dayan and Abbott, 2001). The consequences of developments in this field span across almost every aspect of human life. For instance, understanding how the motor cortex encodes the motor control has empowered the creation of brain-computer interfaces to control robotic prosthetic limbs. Another example of successful application is the cochlear implant, where the sound is directly encoded and transmitted to the auditory nerve bypassing the defective inner ear.

The most precise way to obtain the electrical activity of neurons is by measuring their

voltage using electrodes. In our case, this is achieved by using thousands of electrodes packed in a very dense array (i.e. micro-electrode array). However, getting from the signals of the electrodes to the spike trains of each neuron involves a multistage process named spike sorting (see Figure 2.1). Within this process, a first and crucial step is to detect the spikes from the raw data (i.e. spike detection), where every signal is in fact a mixture of electrical sources. The algorithms implemented in this project only deal with this process and provide the input for the next steps (feature extraction and clustering). It is worth mentioning that, for some applications (e.g. Ventura and Gerkin, 2012), unsorted spikes provide enough information about the neural activity on the regions captured by the array.

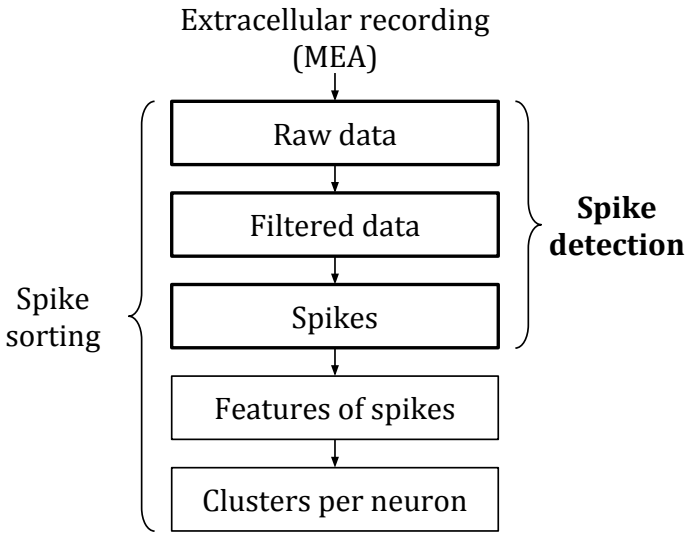

Figure 2.1: Spike sorting process. The spike detection phase, which is the part within the scope of this project, has been highlighted.

2.2 Data involved in the project

The recordings used to test and compare the performance of the software developed in the project are the same as in Hilgen et al. (2016). They were performed with light-stimulated retinas obtained from adult mice and placed in a MEA BioChip 4096S (3Brain GmbH., Switzerland), recording at 7-8 KHz.

2.3 Previous work on spike detection algorithms

3Brain provides a complete software suite to interface the data input from the chip and perform spike detection with three different methods i.e., a simple threshold, a differential threshold (Novellino et al., 2009) and the Precise Time Spike Detection (Maccione et al., 2009). All these methods have been efficiently implemented using multithreaded code and are capable of performing detection on real time in modern workstation CPUs (Maccione et al., 2015). This project intends to parallelise and optimise two more complex spike detection algorithms (see chapter 3) based on online estimates of percentiles of the voltage recordings used to determine the baseline and variability of each signals. Naturally, in all these algorithms there is a trade-off between missing spikes due a high threshold and getting false positives due to noise exceeding low thresholds. Thus, choosing the correct threshold, and adapting it through time depending on the characteristics of the signals, becomes the goal of every algorithm of this kind.

Various types of algorithms can be found in the literature on spike detection (Rey et al., 2015). In order to set a threshold that changes through time according to the data characteristics, one of the easiest ways is to define it as a multiple of the standard deviation of the noise (assuming that it follows a Gaussian distribution). For example, a noise threshold-based detection and spike sorting by template matching was proposed in Quiroga et al. (2004), as the preceding step to spike clustering. In this paper, the authors proposed using an standard deviation of the noise of $\hat{\sigma}_n = \frac{\text{median}(|X|)}{0.6745}$. Here, X is the filtered signal and the denominator is defined as the cumulative distribution function for the standard normal distribution at 0.75. With this estimation of σ_n, the threshold can be defined as $k \cdot \hat{\sigma}_n$ where k is a constant.

Additionally, other methods have been presented, such as using wavelet transformations to detect and localise spikes (Nenadic and Burdick, 2005), a Hilbert transform with probabilistic and fuzzy theory analysis (Azami et al., 2015), or derivatives of the traditional pattern matching algorithm applied to spike detection with predefined spike shapes (De Oliveira et al., 1983). For an early review on spike detection and sorting algorithms, see Lewicki (1998).

Nonetheless, all the former methods suffer from the unavoidable problem of multiple elements signalling through a shared media. That is, multiple neurons firing at the same time and each electrode receiving a mixed signal containing more than one source.

However, it has been shown that higher sampling rates increase the resolution enough so that the amount of information available to discern between overlapping spikes is sufficient to solve the majority of cases (Muthmann et al., 2015).

2.4 Parallelism

2.4.1 Multi-core processors

Since the release of the first commercially available microprocessors in the early 1970s, the performance of processors has increased exponentially following the renowned prediction by Goordon Moore (i.e. Moore's law). That is, that the number of transistors on integrated circuits, and by extension CPUs, would be doubled every two years approximately (Schaller, 1997). However, at the start of the 21st century, and as the size of transistors shrank to less than 100 nm, the CPU manufacturers (mainly Intel, AMD and the AIM alliance) encountered two serious problems: First, the frequency of the processors could not be incremented any more because the heat would not be able to dissipate in such small surfaces. Secondly, the power required to work with an exponentially increasing number of transistors became prohibitively large. While the former problem was addressed and gradually solved through time (thanks to new materials and better instructions per cycle, i.e. IPC, ratios in general), the first problem was not so easy to address due to physical constraints. In order to provide a solution and continue increasing the performance of computers, the decision of putting multiple processors (named cores) in a single die was made (see Figure 2.2). This way, multiple programs would be executed at the same time, and the number of total transistors would still increase thanks to smaller transistors[1], and even single processes could benefit from this parallelism if appropriately coded.

Multi-core general purpose processors not only introduced a game changing computing paradigm with an scalable future, but also produced various restrictions on how any program could be accelerated by using multiple threads on different cores to ensure correct executions (e.g. avoiding race conditions, deadlocks, etc.). Moreover, while incrementing the frequency of CPUs implied a symmetric increase in the per-

[1]Currently, Intel is developing processors with 10 nm transistors. However, other problems are starting to appear when implementing transistors at such a small scale (e.g. quantum tunnelling). This could imply the end of silicon-based processors in the near future.

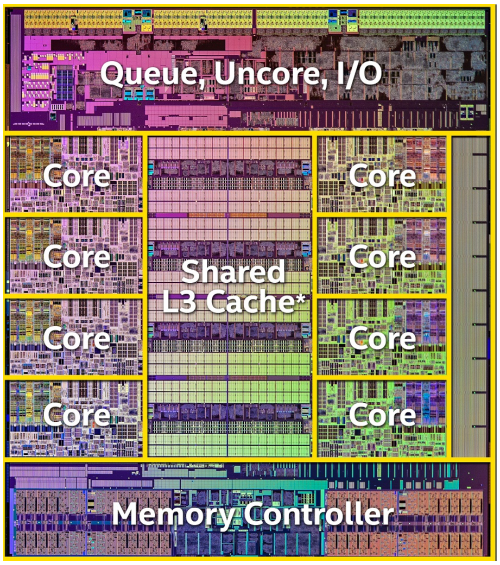

Figure 2.2: Modern 8-core CPU (i7-5960X) with its different components (Intel ©).

formance of the software (assuming same IPC ratios), parallelisation generaly only accelerates the specific parts[2] that are suitable for a multi-threaded execution and it usually introduces an operating overhead.

2.4.2 Graphics processing units and OpenCL

Graphic processing units (i.e. GPUs) provide the hardware necessary to visualise graphical interfaces with 2D and 3D renderings. This is internally achieved by using a pipeline in which one the most relevant steps is the use of shaders. Shaders are small routines that are typically executed for every point or pixel to be rendered[3]. Since computing the shaders for all geometry and pixels is a repetitive and simple task, early GPUs already incorporated several simple processors designed with the sole purpose of computing those shaders. In time, GPUs became massively parallel architectures capable of performing simple tasks related with geometric transformations with a very large throughput (in terms of floating point operations per second) when compared to CPUs. In order to use such architectures for computing, developers started manipulating the input of the graphics cards to contain general data instead of only textures or geometry,

[2]The total performance gain (speed-up) in such application is computed using Amdahl's law: speed-up $= \frac{1}{(1-p)+\frac{p}{s}}$ where p is the percentage of the software subjected to parallelism, and s is the speed-up of that part.

[3]Shaders can be separated into vertex (per point) and fragment shaders (per target pixel). More recently, geometry shaders have been included to the pipeline to allow arbitrary transformation to geometric primitives.

starting the General-Purpose computing on GPUs (GPGPUs). Soon, manufacturers of graphics cards developed tools (on software and hardware) to allow programmers to accelerate their software using GPUs without having to rely on tricks (e.g. CUDA by Nvidia, released in the 2007). Two years later (2009), the open source standard OpenCL was created.

Since SYCL is based on OpenCL, some of its main standard conventions will now be explained in order to provide some background before the description of the implementations of the algorithms in chapter 4.

OpenCL defines a **device** (a compatible GPU or CPU) as a collection of **compute units**, were **kernels** (i.e. functions/procedures) are executed when a **host** (typically a CPU) enqueues a **command**. Each compute unit executes an undefined number of **workgroups**, but a workgroup is always executed in a single compute unit. In OpenCL, a workgroup is a group of kernels that are executed together (not necessarily at the same time, though), and it provides a mechanism to distribute the work of the algorithm into equally sized groups of tasks with shared and faster memory (local memory). All workgroups are composed of a number of **workitems** which is limited by each device (e.g. the AMD FirePro 5100W has 12 compute units that can execute workgroups with up to 256 workitems each). Figure 2.3 shows a device in OpenCL terms.

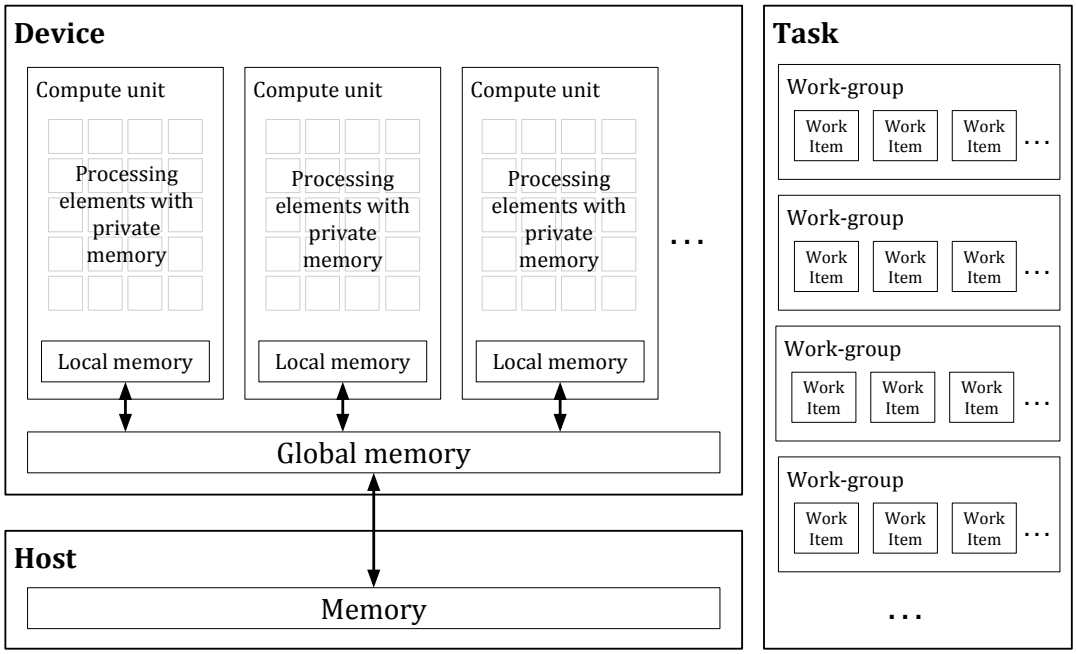

Figure 2.3: Architecture the OpenCL paradigm including a device, a host and a task.

Usually, kernels are executed in a workgroup at the same time using a compute-unit with the maximum number of processing units[4] available to maximise the computing performance (the maximum number of workitems specified by its OpenCL implementation is, at least, the number of processing units per compute unit). In addition, such compute units only reach their full potential when all kernels involve the same type of operations and there are no different work-flows through control statements between elements of the same workgroup. Otherwise, the execution flow will be executed sequentially for every group of workitems sharing the same branch, and reunited when all threads in the workgroup return to the same scope.

2.4.3 Single-source Heterogeneous Programming (SYCL)

SYCL is the new open-source standard for implementing cross-platform *C++* software using an abstraction layer on top of *OpenCL* designed to run on heterogeneous devices. Its single source design implies that, instead of compiling kernels and host code separately (as it is usually done), the new implementations of SYCL allow to write host and device code within the same files using only *C++11* syntax.

A simple example using *SYCL* syntax following the specification 1.2 is presented below to illustrate a recurrent work-flow used in this project:

First, a *SYCL* device queue must be created to enqueue any command.

```
using namespace cl::sycl;
queue Q();
```

Then, in order for *SYCL* to know which variables will be required in the device (and potentially copy them), buffers must be created using the memory directions in the host and the range of elements. In this example, the one dimensional array *V* of *N* elements will required in the device.

```
buffer<int, 1> VBuffer (&V, range<1> (N));
```

The next step is to define the command that is going to be enqueued, which will define a scope at the workgroup level.

```
Q.submit([&] (handler& cgh) {
```

[4] Processing units are called streaming multiprocessors (NVIDIA) or stream processors (AMD) depending on the vendor.

Inside this context, the first thing to do is determine the kind of access required for the buffers defined above. *VPtr* will be an accessor to *V* in the device memory with read and write permissions, visible inside the kernel scope.

```
auto VPtr = VBuffer.get_access< access::mode::read_write >(cgh);
```

In addition, we will create a local variable of size *M* shared between workitems in each workgroup. In this case, *M* will also be the number of workitems per workgroup.

```
accessor<int, 1, access::mode::read_write,
                 access::target::local> localV(range<1>(M), cgh);
```

Once the accessors are defined, the kernel code can be specified with an item variable, which will provide an unique id to each kernel (to perform different tasks) within the range of workitems for which the kernel is called. Since this is a simple one dimensional workspace, the index will return the global index when consulted with the `get_global(dimension)` function. Furthermore, the position of the workitem inside all workgroups can be checked with `get_local(dimension)` and the workgroup number can be accessed by calling `get_group(dimension)`. In the following code, each workgroup will copy the elements to local memory, wait in a barrier for all the other workitems in the workgroup to finish their copy, perform several additions to a private variable accessing the local array (assume that *M* is substituted by its value), and finally copy back the values to the global buffer.

```
auto exampleKernel = ([=](nd_item<1> it) {
    auto i = it.get_global(0);
    auto local_i = it.get_local(0);
    localV[local_i] = V[i]; // Copy to local memory
    int privateVariable = 0; // Initialise private variable

    // Wait all workitems within the same workgroup
    it.barrier(access::fence_space::local_space);

    // Perform computation
    for (int i = 0; i < 10; i++)
        privateVariable += localV[(local_i + i)%M];

    // Copy back to global memory
    V[i] = privateVariable;
}
```

Before calling the kernel with `parallel_for`, the range of elements for which it will be executed must be defined. In this case, the total number of workitems (elements in the array) is N and the workgroup size is M (both one dimensional spaces).

```
auto workSpaceRange = nd_range<1>{ range<1>(N), range<1>(M) };
```

Finally, the kernel can be called and the command scope can be closed.

```
cgh.parallel_for<class example>(workSpaceRange, exampleKernel);
} // Close Q.submit
```

Chapter 3

Spike detection algorithms

This section outlines the two algorithms described in Muthmann et al. (2015). It is worth noting that since the scope of the project does not include the tuning of parameters in connection with the accuracy of the algorithms, their values will not be justified here. Nonetheless, their name, value and function is presented below (Table 3.1) for the sake of clarity.

Table 3.1: Default parameters for the algorithms described in Muthmann et al. (2015).

Parameter	**Value**	**Description**
θ	6	Detection threshold
θ_{ev}	10.5	Minimum depolarisation area
θ_b	0	Repolarisation threshold
f_s	7022 Hz	Sampling rate
f_v	0.03125 $f_s \mu V$	Variability update rate
f_b	0.5 $f_s v$	Baseline update rate, where v is the estimation of the variability of the baseline per channel
τ_{event}	1 ms (7 frames)	Maximum depolarisation width
τ_{ev}	0.27 ms (2 frames)	Interval for depolarisation (and window for coincident events in the interpolation based algorithm)
τ_{pre}	1 ms (7 frames)	Frames included in the output before the spike peak
τ_{post}	2.2 ms (15 frames)	Frames included in the output after the spike peak
w_{cs}	4	Ratio between center and surrounding signals for the 5-channel interpolation

3.1 Online spike detection algorithm

The first variant of the algorithms addressed in the project detects spikes over the raw data of each signal or channel independently. The output contains the channel number and time-step as well as the amplitude for each detected spike.

Specifically, the algorithm detects a spike when the voltage exceeds a specific threshold. In order to account for global variations across all channels, the average voltage per time-step (frame) is subtracted from all channels. Then, spikes are detected using a threshold that is computed with respect to an estimated baseline of the signal for each channel (see Figure 3.1). This estimation of the baseline (i.e. b) represents the local 33rd percentile of the signal and, together with an estimation of its variability (i.e. v), is updated on each channel at every time-step as the algorithm advances through the signals (i.e. s). Thus, it does not require all the data in advance (online algorithm). The baselines and variabilities of the channels are updated as follows:

- b is increased by $\frac{1}{2}\frac{f_b}{f_s}$ if $s \in (b+v, \infty)$ or decreased by $\frac{f_b}{f_s}$ if $s \in (-\infty, b-v)$.
- v is increased by $\frac{f_v}{f_s}$ if $s \in (b-v, b] \cup (-\infty, b-6v]$ or decreased by $\frac{f_v}{f_s}$ if $s \in (b-v, b]$.

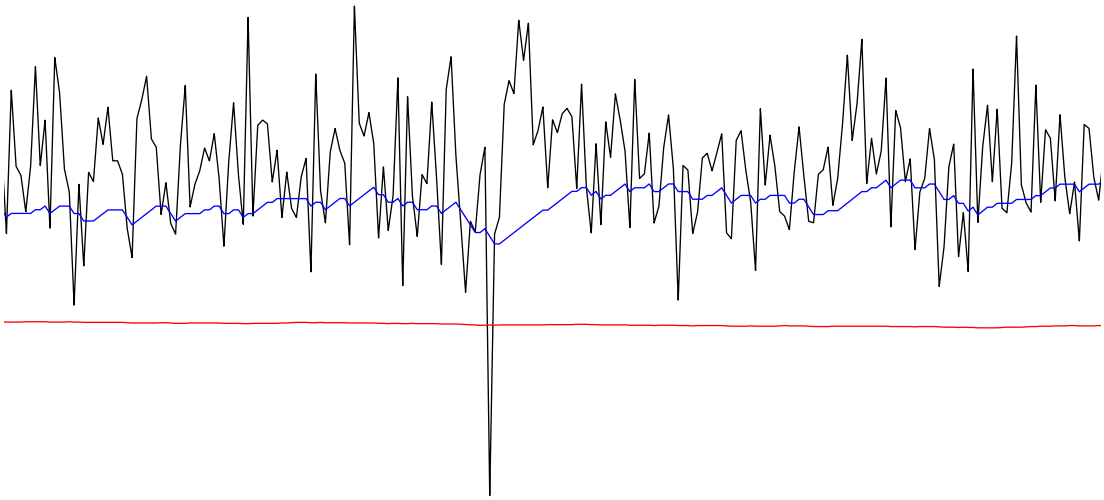

Figure 3.1: Signal of a channel (displayed in black) with its baseline and threshold for detecting spikes. The threshold (in red) is computed as θv, where v is the estimation of the variability in the baseline (shown in blue), updated at every time-step.

After a signal exceeds the threshold (i.e. the depolarisation is greater than θv), three criteria are used to discard false positives (spikes must fulfil all of them):

- The signal repolarises within the following τ_{event} frames after the spike.
- The spike voltage is the local minimum within the following τ_{event} frames.
- The sum of the baselines subtracted signals of the following τ_{event} frames is larger in amplitude than $\theta_{ev} v$.

3.2 Interpolation-based spike detection algorithm

In order to exploit the signal spread over multiple channels during detection and hence reduce the amount of false positives due to channel-specific noise, Muthmann et al. proposed the use of interpolated signals amongst nearby channels as a preprocessing step. Then, a similar baseline-threshold detection as in the aforementioned algorithm could be used to detect the spikes.

In detail, the algorithm first computes a moving average of all signals in order to reduce the noise. Next, baselines are computed for each channel and the minimum of every two consecutive frames is stored. The reason behind this is that in high sampling frequency recordings, spikes can be detected in contiguous frames for neighbouring channels. With these baselines, the next step is computing the interpolations in two different ways. A five-channel and a four-channel interpolation:

- The five-channel interpolation assigns a different weight to the signal of a central channel and the 3 channels with largest amplitude of the four adjacent ones (see Figure 3.2) and stores the weighted average in the position of the central channel. The factors used to interpolate the signals v are $v\frac{w_{cs}}{3+w_{cs}}$ and $v\frac{1}{3+w_{cs}}$ for the central and peripheral channels respectively, where w_{cs} determines the ratio of contribution to the interpolated signal between centre and peripheral channels.
- The four-channel interpolation takes the three channels with largest amplitude in each square of 4 channels (see Figure 3.2) and stores the average in a virtual position placed at the centre of the subset.

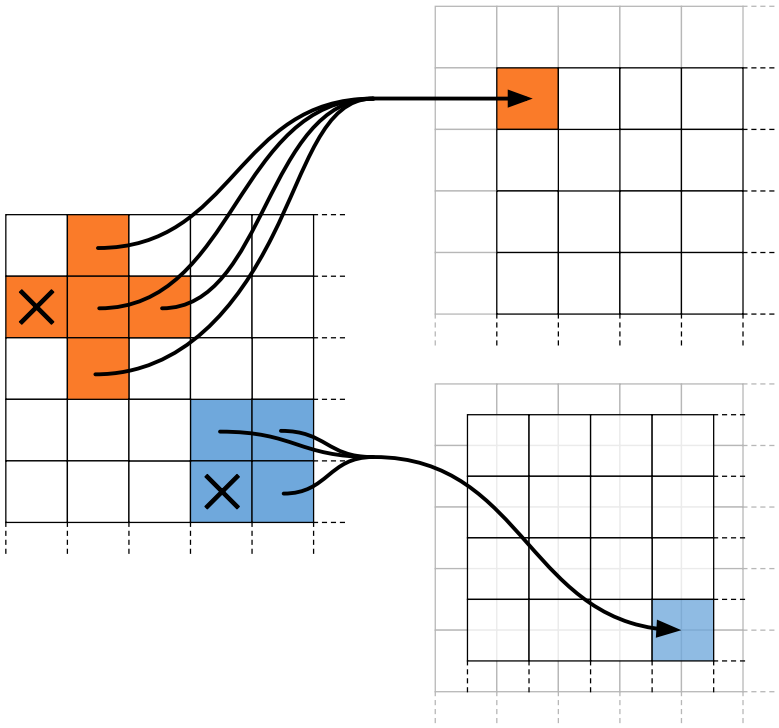

Figure 3.2: 5 and 4 channel interpolations. The 5-channel interpolated signals are stored in the location of the central channels and the 4-channel interpolated signals are stored in the centre of the 4 origin channel square. X indicates the channel with the weakest signal, which is not taken into account in the resulting interpolation.

The detection is then performed on the interpolated signals following the same procedure as in the online algorithm with a threshold-based spike detection. Furthermore, the same criteria for discarding false positives is applied. Regarding the output of the algorithm, the shapes of the spikes are printed on a file in order provide an input for an spike sorting algorithm (Hilgen et al., 2016). This shape is defined as the voltage for the channel with the largest depolarisation and the surrounding channels for any spike at times t, including all respective frames within the range $(t - \tau_{pre}, t + \tau_{post})$.

Chapter 4

Implementation of the algorithms

The programs developed during the course of the project and the decisions that shaped their implementation are explained in this section.

4.1 Methodology

4.1.1 Considerations on the programming language

Implementing fast and optimised code is usually associated with the use of low-level languages such as *C* or *C++*, where the programmer can control more explicitly the computations and manage the memory associated with the algorithms. However, these languages suffer from lower readability in comparison to others (e.g. python), complicate modification of the algorithm parameters on the go, and limited usability by people not aware of the internals of the software. Since the code developed in this project is aimed at being used by anyone interested in spike detection, it has been deemed necessary to use a high level interface in *python*. Nonetheless, the core has been implemented in *C++*, starting with the standard version *C++11* (naturally, *C++14* and posteriors are also supported), for two reasons:

- *C++11* includes multithreading support in the standard library.
- *SYCL* requires (at least) *C++11* (explained in subsection 2.4.3).

There are several libraries aimed at linking *C++* and *python* code. For example, wrappers such as *SWIG*, *Boost.python* or the newer *pybind* enable interoperability between

languages by exposing *C++* functions to *python*. Probably, the most common way of writing python modules is using *Cython* (Behnel et al., 2011), a translator for pseudo python code with *C* features, which is able to be called by pure *python* code and, in turn, calls *C++* functions (see Figure 4.1).

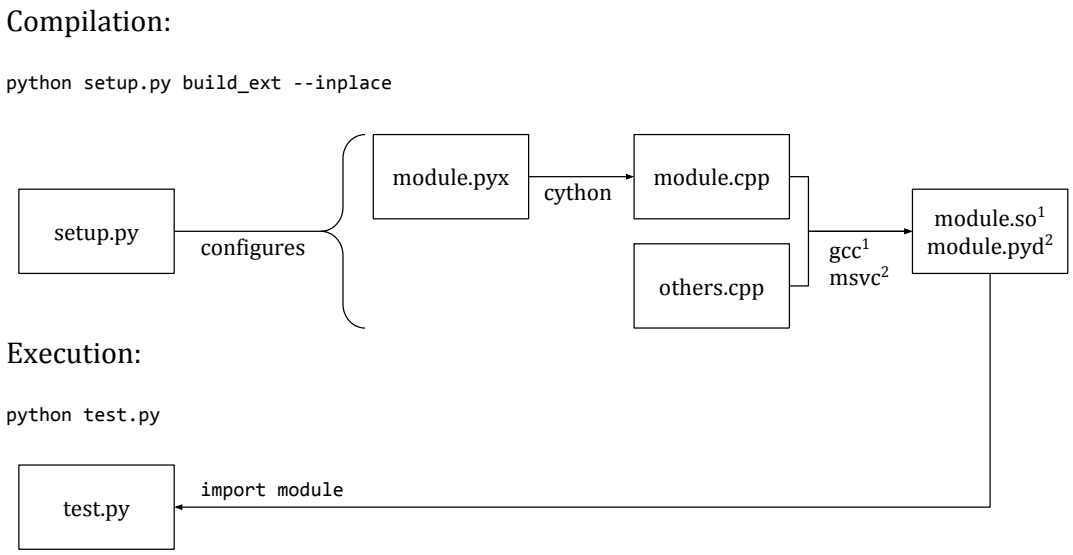

Figure 4.1: Diagram of the compilation and execution of *python* modules using *cython*. The file *setup.py* tells *cython* which files and compilers must use. This ultimately generates the compiled module files ([1] for linux systems and [2] for windows) which can be called after being imported in python as any generic module.

The fact that the *python* code is translated to *C++* completely eliminates any overhead that might occur when comparing with other wrappers, and yet maintains the readability and usability at a medium level of abstraction.

4.1.2 Data input format

The format in which the data coming from the chip is acquired is not fixed to any standard and might change depending on the vendor or even version of the chips' firmware. In the original implementation (Muthmann et al., 2015), the data was handled using proprietary libraries by 3Brain. However, during the course of this project, their latest version of the software used the data model *HDF5* together with a set of transformations over the original data. In particular, the signals are presented within a 2 dimensional matrix, stored by time-step in its first dimension, and by channel in

the second one. To provide support for any future format and allow users to easily extend the way the data is handled in during the read, all the input functions have been writen in a file called *readUtils.py*. In the current version, *HDF5* files are read using the *python* module *h5py* and modified[1] to match the old format. Therefore, the input performance has not been addressed in the project beyond the fact of reading chunks of data to maximise the throughput and reduce the overhead (see chapter 6).

Due to structural limitations of the MEAs, the amplifier of a signal may go out of linear regime and reach saturation for a range of frames creating what it is known as an outlier. Consequently, all implementations deal with this kind of invalid signals and are able to isolate them from the variables used to estimate the baselines and other on-going computations of the algorithms.

4.1.3 Work plan of the implementations

In order to understand the algorithms developed in this project, a first step has been analysing the current prototype implementations in *C#* and their possible bottlenecks. The majority of these bottlenecks (see section 4.3 and section 4.4) are related to the way the algorithms structure the data that is processed and hence the data dependencies underlying the existing control flow of the original implementations. By isolating them, measuring their impact and researching and implementing possible workarounds, many of these issues have been resolved or minimised.

Due to different hardware/software requirements, multiple versions have been developed for each algorithm. These implementations have been developed using the following sequence of pertinent tasks:

1. Porting the original online spike detection algorithm to *C++* with a python interface. This involved changing the syntax of the *C#* implementation and adapting it to work with python and the new data format.

2. Adapting the online algorithm *C++* port to run on an arbitrary number of threads threads using only *C++11* features.

3. Extending the parallel *C++* port of the online algorithm to run with *SYCL* kernels

[1] Unsigned signals are subtracted to 4095 to invert their absolute amplitude and then concatenated in a one dimensional array. The input files contain frames with 16-bit unsigned integer precision voltages which are converted to match the *C++* standard type *unsigned short* (guaranteed to fit 16 bits).

on both GPUs and CPUs.

4. Porting the original interpolation-based spike detection algorithm to *C++*, again, with a python interface and compatible with the new data format.

5. Implementing an equivalent interpolation-based algorithm from the ground up taking into account a parallel execution flow using only C++11 features.

6. Extending the new interpolation-based algorithm with *SYCL* kernels on a suitable part of the algorithm.

4.2 Porting the algorithms to *C++*

Adapting *C#* code to *C++* has involved a series of simple syntax conversions related with array declarations and access, control statements (e.g. foreach), file input and ouput statements, type declarations and built-in functions (e.g. sort and max). It is worth noting that simple changes such as distributing the input data in a contiguous one dimensional array (with stride access) instead of sparse bi-dimensional structures can yield a significant improvement on performance if the algorithms can benefit from accessing cached data (locality of reference, spatially and temporally). This is discussed with regard to the global average computation of the signals and other computations in chapter 6.

One feature that has been kept the same way as in the original implementation is the fact of processing the data in temporal chunks. Even though this contradicts the online nature of both algorithms, it can be understood as using a buffer for the input data, processed only when it reaches a minimum size. The use of a multi-step data chunk allows for a faster processing of the data since it reduces the amount of time dedicated to start all the different procedures that compose the algorithm. Otherwise, all those procedures would be called at every time-step of the algorithm, which at high sampling rates (e.g. 7022 Hz) would be prohibitively slow.

In addition, many parameters of the algorithms as well as some execution parameters (e.g. data chunk sizes) have been externalised and placed at a *python* level. This allows for a fine control of the algorithm without having to recompile the whole module (.pyx and .cpp files) and it does not have any impact on performance (all parameters are passed only once).

4.3 Parallel implementation of the online spike detection algorithm

4.3.1 Standard C++ implementation

The online algorithm is mainly composed of three steps, which are performed on every data chunk that is processed.

First, the data is read. Then, the global average signal is computed for each time-step. This is, the average voltage for all channels at each time-step (excluding all outliers) is stored to be used during detection. And finally, spike detection is performed for all channels independently.

Assuming every data-chunk includes the voltage of a number of channels (i.e. *NChannels*) for several time-steps (named *tInc* because it represents the amount of time-steps increased every time the algorithm is called), the averaging and detecting sections have been parallelised using the following distribution of work on a system with a maximum number of threads *nthreads*.

Each thread computes the global average for the a number of time-steps distributed in a loop executed by all of threads.

```
int chunkSize = std::ceil( (float) tInc/ (float) nthreads );
for (int t = threadID*chunkSize;
         t < tInc and t < (threadID+1)*chunkSize; t++) {
    // Compute global average across all channels at time t
}
```

Here, *threadID* is a consecutive identifier different in all threads (0 to *nthreads* - 1).

For the spike detection, each channel can be processed independently. Thus, the former loop is adapted to allow each thread to process a number of channels instead of a number of time-steps.

```
// Number of channels associated to a thread
int chunkSize = std::ceil( (float) NChannels/ (float) nthreads);
for (int t = 0; t < tInc; t++) {
    // Loop accross all channels associated to this thread
    for (int i = threadID*chunkSize;
             i < NChannels and i < (threadID+1)*chunkSize; i++) {
        // Spike detection at time t for channel i
```

```
    }
}
```

Note that the time loop has been left at the outermost level to make all threads advance through time simultaneously (it is not explicitly forced though). Since the data is organised by time-step, it is expected a better use of any intermediate memory cache thanks to this execution flow.

In both cases, threads are called using the standard *C++11* notation and joined right after finishing their work.

```
for (int threadID = 0; threadID < nthreads; threadID++) {
    threads[threadID] = std::thread( [=] {
        func(threadID, param1, param2, ...);
    });
}
for (int threadID = 0; threadID < nthreads; threadID++) {
    threads[threadID].join();
}
```

In order to avoid errors when simultaneously writing to file, the output of the program is managed with a mutual exclusion lock shared amongst threads. This means that every time a thread finds a spike and needs to print, it locks the output by calling `output_mtx.lock()` and after writing the amplitude, time-step and channel identifier the lock is released with `output_mtx.unlock()`[2].

Code:
`onlineDetection/SpkDonline.cpp`
↪ `MeanVoltage`, `MeanVoltageThread`, `Iterate` and `IterateThread`

4.3.2 Heterogeneous C++ implementation

The part of the code that takes more time to execute is naturally the detection of spikes. Therefore, since the overhead related to copying the data to the device (potentially an external GPU) would already be bigger than the sequential execution time in a CPU for

[2] `output_mtx` has the type `std::mutex`. All the functions related with threads and locks require including `<thread>` and `<mutex>` in the header file.

the averaging part, only the spike detection section has been parallelised using *SYCL* (discussed in chapter 6).

As has been explained for the standard *C++* implementation, the spike detection part of the algorithm can be processed independently on each channel. Accordingly, every thread (workitem in *OpenCL* terms) will now process a single channel independently for the sake of simplicity. Following the steps explained in subsection 2.4.3, a *SYCL* device is selected and a queue is created in order to assign the work. Then, multiple buffers are used to refer to the variables in memory with different `accessor` permissions (read/write) depending on the kinds of accesses performed during the spike detection. Finally, the kernel is called for the whole range of channels.

Internally, in order to avoid repetitive accesses to the global memory of the device, the variables associated with the baseline, variability and evolution of the spike detection for a channel in general, are copied to the private memory of each thread. Then, the spike detection is performed in a loop across time. Before ending the kernel, all the working variables are stored again in global memory to allow a putative next call of the algorithm to continue from where it ended.

One of the major caveats of using *SYCL* and, more precisely, *OpenCL*, is that kernels cannot directly write to files (or read them). To solve this, all spikes are now stored in an array which can be updated by multiple threads at the same time. Specifically, each kernel writes to different positions in memory. A drawback of writing in a array is that memory for all spikes will need to be pre-allocated. Nevertheless, since the number of spikes is relatively small comparing with the memory necessary to store the input, a maximum number can be safely estimated beforehand to allow all threads to write at different specific regions without overflowing. To maximise the throughput of the implementation, the printing of the output is delayed until the next iteration or, in its absence, the end of the program. The effects on the performance are displayed in Figure 6.1.

Code:
`onlineDetection/SpkDonline.cpp`
↪ `IterateSYCL`

4.4 Interpolation-based spike detection algorithm

Porting the interpolation-based spike detection algorithm to *C++* has been relatively straightforward. However, the original *C#* algorithm computed everything (including the baseline estimates, interpolations, and the detection of spikes) within the same loop. The fact that there are precedence between the aforementioned variables (in the stated order), and that all these tasks could be distributed in different functions specifically optimised for each sub-task, has motivated the creation of an equivalent new algorithm due to the infeasibility of adapting the existing one. The new implementation has been programmed in a modular and more object oriented style, incorporating modern *C++* features wherever possible as a proof of concept of a parallel version. Additionally, the proposed implementation incorporates a new data structure aimed at reducing the number of redundant spikes within a spatial region and a temporal interval.

Port code:
`interpDetectionPort/SpkDslowFilter.cpp`

4.4.1 New equivalent parallel implementation

In the new implementation, after reading each data chunk, the signals are processed in 4 major steps following the original algorithm (section 3.2):

Computation of the global voltage per frame:

Similar to the online spike detection algorithm. A group of equally sized time-steps is processed independently by different threads disregarding the outliers.

Code:
`newInterpDetection/InterpSpkDetection.cpp`
`↪ computeMean`

Estimation of the global average subtracted baselines:

In this case, two major differences can be pointed out when comparing with the original algorithm. First, a moving average of a parametrised length is taken as the input signal instead of the raw one. This average is calculated at every time-step incrementally:

```
vMovingAvg[ch] += ((int) vm[ch + t*NChannels] - (int) v_old)
                    *scale / movingWindowLength;
```

Where `v_old` contains the voltage at the time-step $t - movingWindowLength$, `ch` is the index for the channel, and `scale` is a factor multiplying all signals to provide more resolution during the interpolation and spike detection phases (the original data with `unsigned int` type is scaled and stored as `int` to provide more resolution without having to rely on floating point types).

The second main difference is that the minimum of every two consecutive frames is stored. Nonetheless, the work is distrbuted across threads using the same chunk-based method:

```
// Number of channels associated to a thread
int chunkSize = std::ceil( (float) NChannels/ (float) nthreads);
for (int t = start; t < tInc; t++) {
    // Loop accross all channels associated to this thread
    for (int ch = threadID*chunkSize;
              ch < NChannels and ch <  (threadID+1)*chunkSize; ch++) {
        // Estimation of the global average substracted baselines
    }

}
```

Code:
`newInterpDetection/InterpSpkDetection.cpp`
↪ `preprocessData`

Interpolation of the signals (4 and 5 channels):

The computation of the interpolations can be performed independently for all time-steps and channels. Still, threads have been given a chunk of time-steps with all their corresponding frames (all channels) to interpolate as their part of work. The reason is that every interpolation takes multiple values, and different interpolations within a same frame use signals from the same channels (when computing surrounding interpolations). This way, threads can have faster access to the variables thanks to cached memory because they might already accessed that direction.

```
int chunkSize = std::ceil( (float) (tInc - start)/ (float) nthreads);
// Loop accross all channels associated to this thread
for (int t = start + threadID*chunkSize;
        t < tInc and t < (threadID+1)*chunkSize; t++) {
    // Compute 4/5-channel interpolations on all channels at time t
}
```

In detail, the 4-channel interpolation code takes the average of the three largest negative voltages of a four neighbour channels. When an outlier is found in one of the channels, the execution flow is interrupted and an mark (i.e. *outlierMark*) is returned instead.

```
int values[] = {V[ch + t*NChannels],
                V[ch + 1 + t*NChannels],
                V[ch + chCols + t*NChannels],
                V[ch + chCols + 1 + t*NChannels]};
int interp = 0;
int maxValue = 0;
bool outlier = false;
for (auto v : values) { // Add all the values and find the minimum
    if (v == outlierMark) {
        outlier = true;
        break;
    }
    interp += v;
    maxValue = max(maxValue, v);
}
if (outlier) return outlierMark;
else return (interp - maxValue)/3;
```

Similarly, the 5-channel interpolation takes the weighted average of the three largest surrounding channels surrounding another, together with the value of this central channel with a larger weight using the ratio defined by w_{cs}. Again, when any of the values is identified as an outlier, the interpolation is stopped and the outlier mark is returned.

```
int values[] = {V[ch - chCols + t*NChannels],
                V[ch + 1 + t*NChannels],
                V[ch + chCols + t*NChannels],
                V[ch - 1 + t*NChannels]};
int interp = (V[ch + t*NChannels]*w_cs)/(3 + w_cs);
int maxValue = values[0]/(3 + w_cs);
bool outlier = (V[ch + t*NChannels] == outlierMark);
if (outlier) return outlierMark;
for (auto v : values) { // Add all the values and find the minimum
    if (v == outlierMark) {
        outlier = true;
        break;
    }
    int weightedValue = v/(3 + w_cs);
    interp += weightedValue;
    maxValue = max(maxValue, weightedValue);
```

```
}                                                                    17
if (outlier) return outlierMark;                                     18
else return interp - maxValue;                                       19
```

In both codes, the smaller[3] signal amplitude is subtracted at the end to keep the computations as light as possible.

Code:
`newInterpDetection/InterpSpkDetection.cpp`
↪ `computeFourChInterp` and `computeFiveChInterp`

Spike detection:

The spike detection part is fairly simple compared with the online algorithm. Since the baselines have already been computed and interpolated, the signals are processed to find negative peaks (exceeding a threshold) across all channels. In order to take the maximum depolarisation of a series of values surpassing the threshold, the largest signal and its time step is stored during several frames until a valid repolarisation (end of the spike) is found.

However, taking the largest depolarisation of a certain region has involved creating a custom data structure described below. The data structure aims at solving the problem of detecting redundant spikes (i.e. spikes originated by a single neuron which are reported on multiple adjacent channels and wrongly detected more than once).

The channel space is divided into different squared chunks, where the width (d in Figure 4.2) of a chunk is mandatorily larger than the minimum distance used to discard spikes because of being too close. When a spike is found, its largest amplitude is stored in the corresponding chunk together with its source channel id. In addition, a queue data structure is used to store all recent spikes using the first in first out method. The underlying motivation of this structure is that, whenever a new spike is found, it will only be compared with the spikes found within its chunk and the surrounding ones, thus, comparing with the set of spikes that contain all possible relevant spikes. If a new spike is larger than any other within a set distance, the other ones are deactivated (an internal Boolean value is set to false). Otherwise, if a new spike is smaller than any other nearby spike, it is stored being already deactivated. Temporally, after every time step is processed, the global queue is checked and all spikes old enough so that it is

[3]Since we are dealing with negative voltages, the smallest signal amplitude is taken as the max (smallest depolarisation).

impossible for them to be overshadowed by a new one are eliminated from both queue and corresponding chunk. However, before any spike is eliminated, if its Boolean relevance value is still set to *true*, it is printed to file. This way, only the largest spikes in a temporal and spatial region are stored.

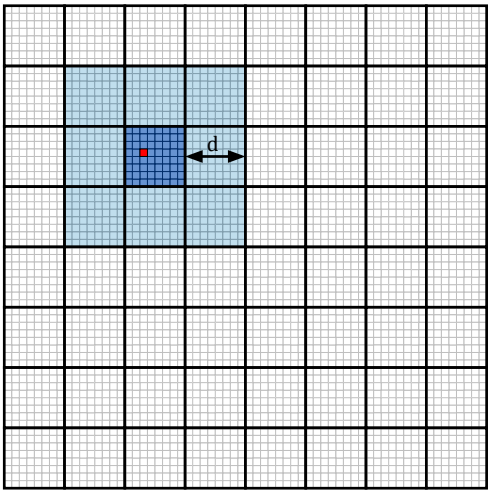

Figure 4.2: Subdivision of the 4096 channel space into 64 chunks of 8x8 channels each. Whenever a spike (in red) is found, its amplitude is compared with only those found in the same and surrounding chunks.

This ensures a linear number of comparisons between spikes (if the number of chunks is increased as the number of channels does) instead of a quadratic one. Furthermore, since the global queue will contain in its front position the oldest (or as old as others) spike, eliminating all the spikes older than a certain time-step becomes trivial.

Code:

`newInterpDetection/InterpSpkDetection.cpp`
↪ `findSpikes`

4.4.2 Heterogeneous C++ implementation

In order to show that the former code can also be parallelised using an heterogeneous implementation, the interpolations have been programmed to work with *SYCL* as well. The interpolations are specially suitable for being computed in a GPU because involve performing many simple computations that share spatially distributed portions of the

data, and are the section that takes most of the time when computed sequentially (see chapter 5).

This time, instead of calling a thread (workitem in *OpenCL* terms) per channel, 256 items per work group will process 16 channels each (4x4) in a single time-step. That is, every workgroup processes 4096 channels (the entire span of the chip) for a time-step. In addition, the local memory[4] of a workgroup is shared and preloaded using all workitems before computing the interpolations to take advantage of the repeated access to memory amongst nearby channels.

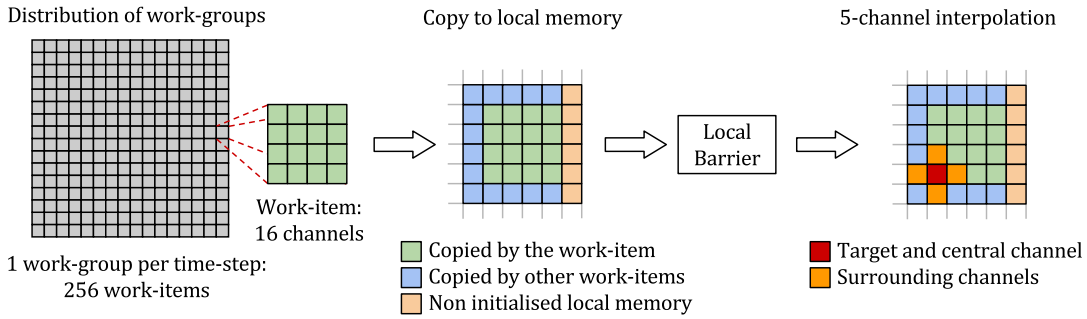

Figure 4.3: Distribution and execution flow of workgroups for the 5-channel interpolation using local memory (shared among workitems within a workgroup). Note that even though interpolations are computed for the fist/last column and row with invalid numbers, they will not be used during the spike detection phase.

In order to provide an homogeneous execution across workitems in a workgroup, the 5-channel interpolation assumes that the signal from the surrounding channels is always available (see Figure 4.3). Since the data is contained in a one dimensional array, this would imply that in order to compute the interpolation of the channels in the first or last row/column of channels (which is computed as any other one for the sake of homogeneity), non existing locations (the surrounding channels of boundary channels) would be accessed. To solve this, a local memory accessor is pre-allocated using 66x66 channels and used with an offset of +1 on each dimension instead of 64x64 channels.

```
auto workSpaceRange = nd_range<3>{ range<3>(tInc, 16, 16),
                                   range<3>(1, 16,16) };
```

Here, the first dimension is the time-step and the other two are the vertical and horizontal position of the chunk. The first `range` determines the global number of workitems,

[4] A whole frame fits in the local memory of the device (FirePro W5100, 32 KB of local memory for each compute unit) used to test *SYCL*.

and the second one determines the range of workitems for a workgroup. Therefore, there will be *tInc* workgroups.

Distinct workitems copy different subsets of channels preparing first a set of global offsets (with regard to the global memory) and local offsets (with regard to the local memory of the workgroup). In order to identify each workitem and workgroup, multidimensional indices have been used:

```
// The first dimension indicates the time
int t = it.get_group(0);
// Block position
int bx = 4*it.get_local(1);
int by = 4*it.get_local(2);
// Prepare offsets (global and local)
int chOffsets[16];
int chOffsetsL[16];
for (int i = 0; i < 16; i++) {
    chOffsets [i] = bx + by*chCols + i%4*chCols + i/4;
    chOffsetsL[i] = 1 + bx + by*chColsL + i%4*chColsL + i/4;
}
// Copy memory (blocks of 16)
for (int i = 0; i < 16; ++i)
    localV[chOffsetsL[i]] = VPtr[chOffsets[i] + t*NChannels];
```

The synchronisation across workitems within a workgroup is achieved using a a local barrier (shown below), and then, interpolations are computed for all channels in the same manner as in the parallel *C++* implementation explained before.

```
it.barrier(access::fence_space::local_space);
```

Code:

`newInterpDetection/InterpSpkDetection.cpp`
↪ `computeInterpSYCL`

Chapter 5

Performance results

This section presents the execution times for the algorithms developed in the project under different configurations to show how well does the implemented parallelisation perform.

All tests have been performed in a server with the following specifications:

- CPU: Intel® Xeon® CPU E5-2630 v2 with a base frequency of 2.60 GHz (3.1 GHz max single core) and a (L3) memory cache of 15 MB. This processor has 6 cores with the capability of executing 12 threads at the same time by means of hyper-treading (a form of simultaneous multithreading, Tullsen et al., 1995).
- GPU: AMD FirePro W5100 with a total of 4 GB of memory, and 32 KB of memory per compute unit.
- Memory: 32 GB of DDR3 RAM.
- Mechanical hard-drive (note that the file reading times have been omitted in the parallelisation analysis).

Many of the tests involved using a different number of cores in the CPU to test the scalability of the developed software. In the case of SYCL kernels, those have been executed in the GPU. It is worth mentioning that the number of CPU cores in the SYCL executions has been of 2 in the interest of representing the performance of a system with an mid-range CPU using a GPU to accelerate some computations. That is, display the performance of a cost-effective system as discussed in chapter 6. Nevertheless, there is no actual reason to not use all cores and the GPU at once in this system to maximise its performance.

Figure 5.1 and Figure 5.2 show how the new *C++* implementation for the online spike detection algorithm performs when executed with different levels of parallelisation. In all cases, real-time performance is achieved (the minimum amount of frames to achieve it is displayed with a dashed line).

Both figures show a significantly large speed-up comparing the sequential execution and the 2-core version of the algorithm. As the number of cores increases, the speed-up becomes relatively less prominent (i.e. sub-linear scaling) due to the additional overhead of using more cores within the same CPU. Therefore, the benefits of using 12 cores instead of 6 are marginal.

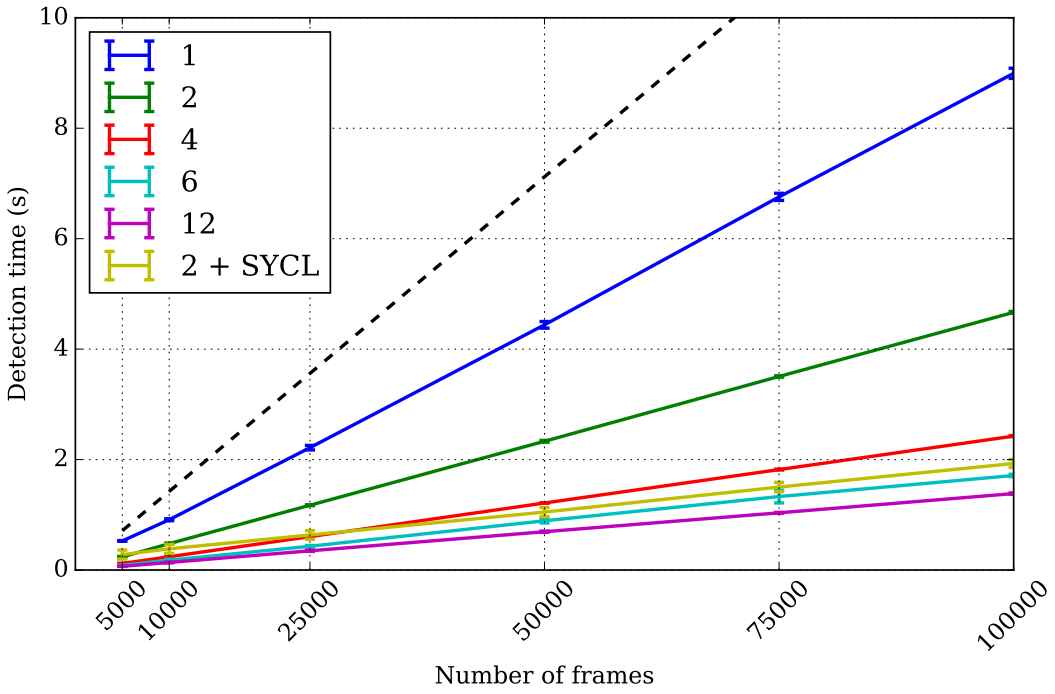

Figure 5.1: Execution time on different data chunk sizes for different levels of parallelism (1-12 CPU cores and 2 CPU cores + SYCL on a GPU for the spike detection part). Each point represents the average time of 6 executions on different data. The marginal variability between executions is displayed with error bars measuring the standard deviation. The dashed line represents the boundary of real-time processing for the MAE sampling frequency of the recording (7022 Hz).

In theory, there is nothing in the implementation preventing the same linear scaling for bigger data chunks from occurring provided the system has enough memory. In our case, sizes higher than $15 \cdot 10^4$ can yield execution errors or significantly larger

execution times due to the lack of memory. In addition, for very small data chunks, the SYCL implementation displays a relatively higher execution time when compared to the other executions.

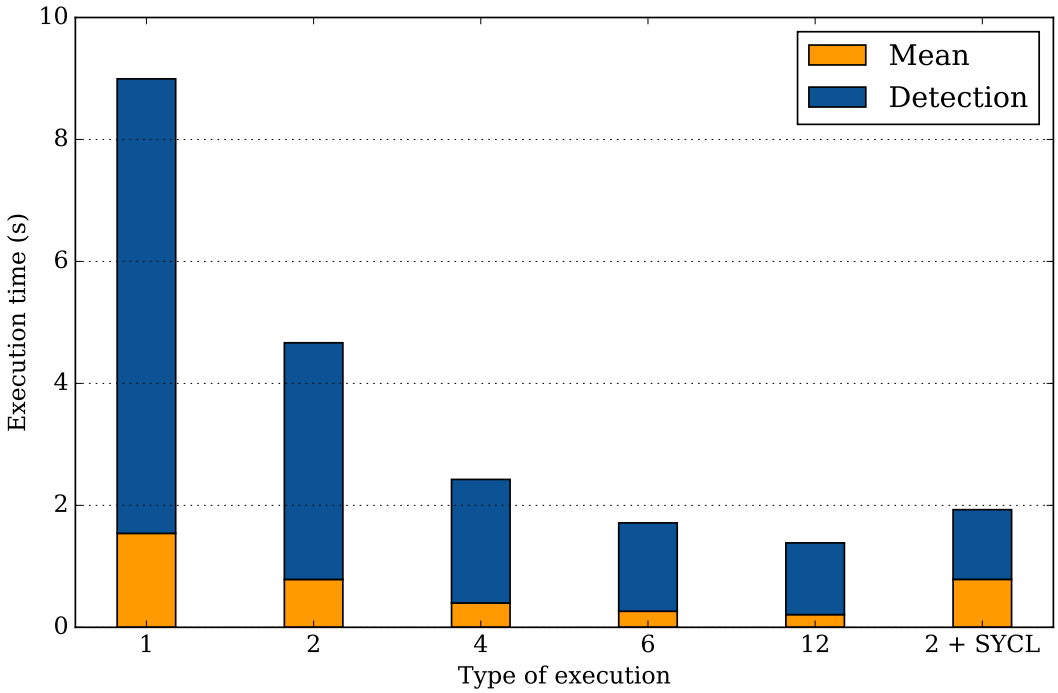

Figure 5.2: Distribution of the execution time with a data chunk of 100000 frames for different levels of parallelism (1-12 CPU cores and 2 CPU cores + SYCL on GPU for the spike detection part). Each set of times represents the average time of 6 executions on different data.

Figure 5.3 and Figure 5.3 show that by means of a parallel execution and a reorganisation of the computations of the interpolation-based spike detection algorithm, the execution time can be reduced to work in real-time with a recording of 4096 channels at 7022 Hz.

Since the complexity of the computations performed in the implementation is significantly higher compared to the online algorithm, even with local memory optimised kernels, the performance shown when using 2 cores + SYCL (GPU computing the interpolations) does not reach the same level of performance as for 4 cores. However, it provides a sufficient boost to achieve real-time performance (drawn as a dashed line).

Overall, the different executions show a quasi-linear scaling with the size of the data chunks (number of frames per iteration). The implementation with SYCL kernels dis-

plays a (marginally) sub-linear scaling which allows to achieve real-time performance for data chunks larger than approximately 10^4 with only two CPU cores.

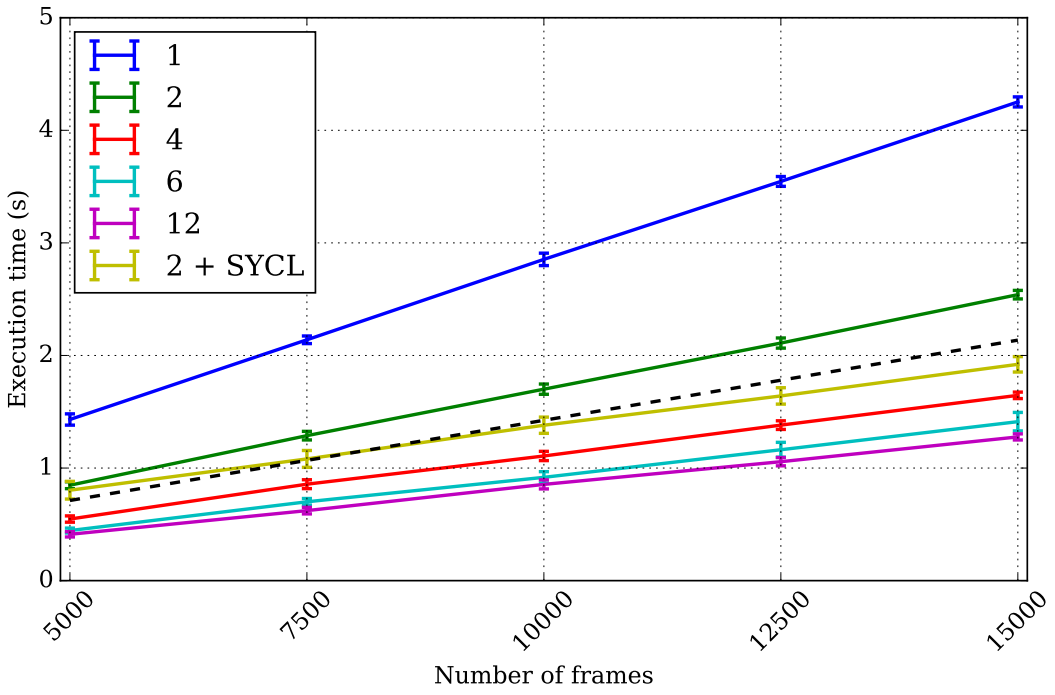

Figure 5.3: Execution time on different data chunk sizes for different levels of parallelism (1-12 CPU cores and 2 CPU cores + SYCL on a GPU used to compute the interpolations). Each point represents the average time of 10 executions on different data. The variability between executions is displayed with error bars showing the standard deviation. The dashed line represents the boundary of real-time processing for the MAE sampling frequency of the recording (7022 Hz).

Figure 5.4 shows the execution time for the different parts of the algorithm. In all cases, the spike detection phase is performed sequentially. Preliminary tests to parallelise such section have yielded somewhat longer execution times and, therefore, have not been included in the final implementation of the algorithm. Other sections show a much shorter execution time, which is specially significant in the estimation of the baselines and the computation of the interpolations (which is the longest part in the sequential execution of the algorithm).

Note that the only difference between the execution with 2 cores and the execution with 2 cores with SYCL kernels is that the interpolations are computed in the GPU.

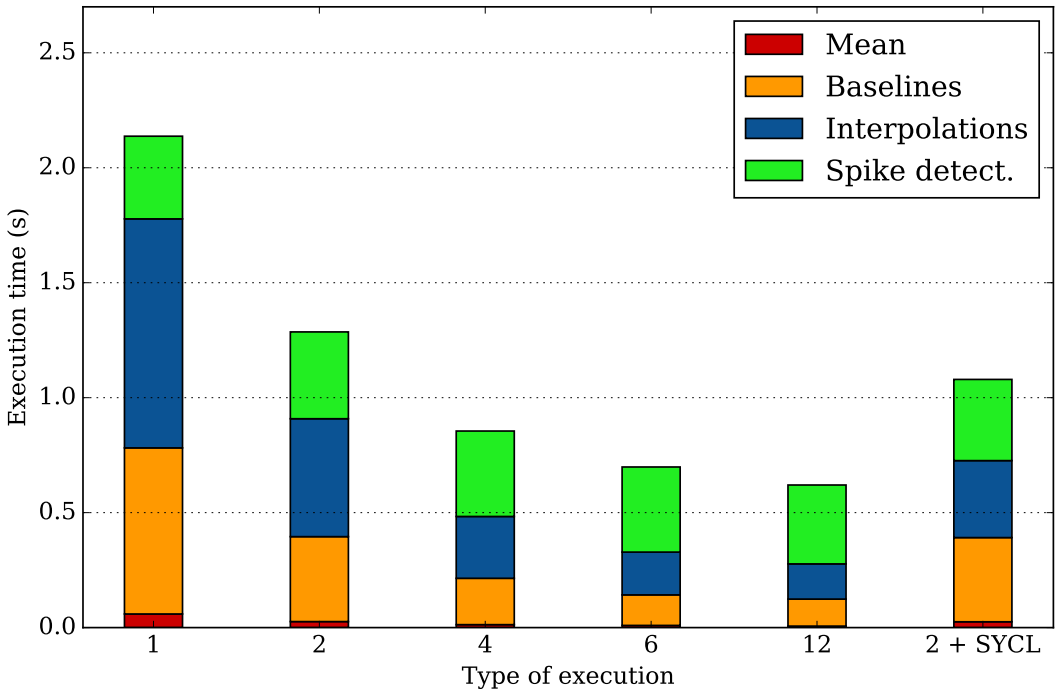

Figure 5.4: Execution time on a data chunk size of 15000 frames for different levels of parallelism (1-12 CPU cores and 2 cores with SYCL on GPU used to compute the interpolations). Each point represents the average time of 10 executions on different data.

Chapter 6

Discussion

6.1 Performance speed-up

The parallelisation of the algorithms first introduced in Muthmann et al. (2015), has significantly shortened their execution time (see Table 6.1). For the online algorithm, which before ran for about ten times the length of the recordings, can now be processed not only in real time but in up to 6 times faster than that. For instance, for a 130 seconds recording, the fastest implementation takes around 20 seconds to finish.

Table 6.1: Example execution times of different versions and levels parallelism for the methods implemented in the project and the original *C#* implementations (Muthmann et al., 2015) on the same 130 seconds recording of 4096 channels with a sampling frequency of 7022 Hz (a total of 912032 frames per channel).

Algorithm	**Version**	**Parallelism**	**Par. speed-up**	**Exec. time**
Online	Original *C#*	-	-	750.07 s
Online	New *C++*	-	1	88.63 s
Online	New *C++*	2 cores + SYCL	3.46	25.63 s
Online	New *C++*	12 cores	4.33	20.46 s
Interpolated	Original *C#*	-	-	1569.79 s
Interpolated	*C++* port	-	-	533.03 s
Interpolated	New *C++*	-	1	254.72 s
Interpolated	New *C++*	2 cores + SYCL	2.21	115.47 s
Interpolated	New *C++*	12 cores	3.39	75.18 s

The translation of the interpolation-based algorithm has accelerated the execution and shorted its execution time to be approximately a third of the original execution time. In addition, the new implementation of the interpolation-based algorithm has achieved real-time performance thanks to the use of multiple threads in two possible configurations: a high CPU load on a high-end CPU, or a less aggressive CPU load combined with a GPU to accelerate the computation of interpolations.

6.1.1 Online spike detection algorithm

The .HDF5 format, different from the one the original implementation used, provides the data indexed by time-step. To exploit this, the new implementation uses a transposed access to the data ($channel + time \cdot NChannels$ instead of $channel \cdot NTimeSteps + time$), which significantly eases the computation of the global potential at each time-step because all the signals of a time-step are contiguously stored in memory. In addition, the fact that all threads perform the spike detection algorithm advancing through time at the same time, allows for a better management of the memory hierarchy present in the test computer.

One of the reasons for the sub-linear scaling when incrementing the number of cores is that all cores ultimately share the same L3 cache, and its usage across is limited by the amount of memory size which has. Furthermore, as happens in many modern processors, when multiple cores are used at their full potential, the CPU (its firmware) automatically reduces the frequency from turbo (only used when few cores are extensively used to perform single threaded tasks) to its base frequency to avoid any damage to the CPU die due to high temperatures (i.e. reduce the electric energy consumption).

A major benefit of using a GPU is that it liberates the CPU from work load, allowing it to perform other computations. This has been taken advantage of in the implementation, where spikes are printed in the subsequent iteration of the algorithm. That is, spikes are stored in memory by the GPU (which cannot directly write to file), and during the next iteration of algorithm, the CPU prints a copy of the results and waits for the ongoing GPU spike detection to finish. Compared to a sequential print of the results right after finishing the computations, it has a higher throughput (see Figure 6.1). In addition, it is worth noting that the use of a GPU thanks to SYCL together with 2 CPU cores provided a significant boost to the performance when compared to the simple 2-core execution time.

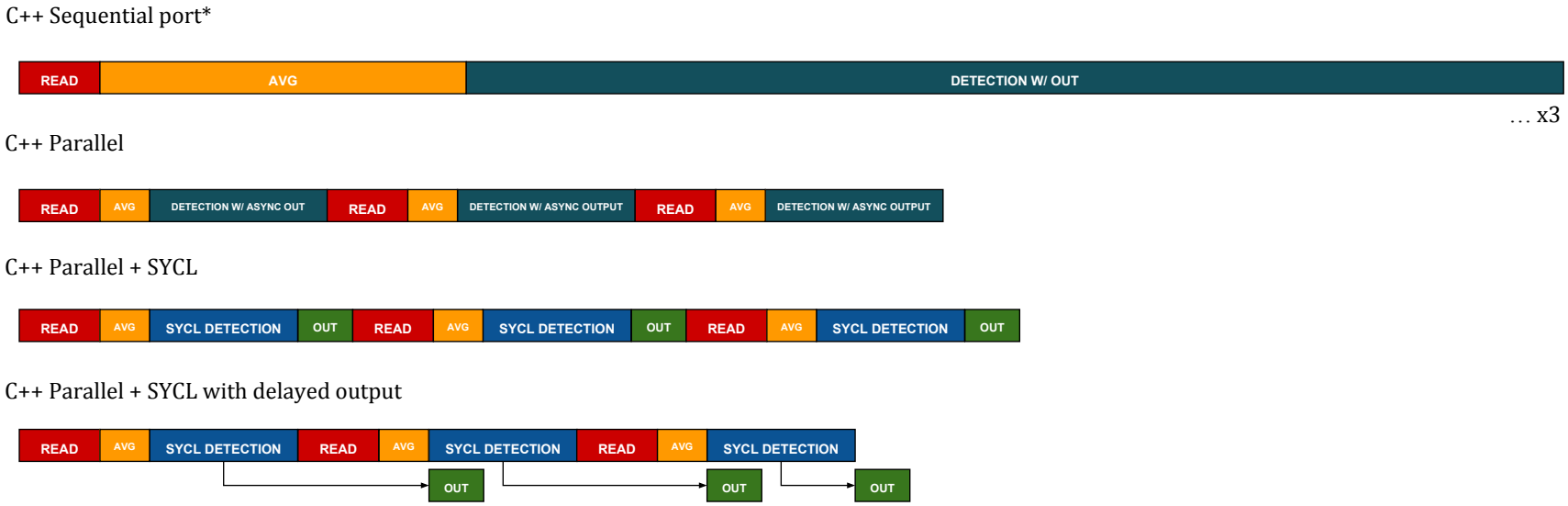

Figure 6.1: Diagram showing the execution times for different versions of the algorithm (sequential, 4-core, 4-core with SYCL for the detection part with and without a delayed output of the spikes). The different colours indicate the different parts of the algorithm: the file input (READ) appears in red and the computation of the global average signal (AVG) across signals in yellow. The output of the spikes is performed differently between the standard *C++* implementation and the implementation with SYCL kernels: in the first case, the output is done asynchronously during the the detection phase (in dark blue) whilst in the second one the output is performed right after the spike detection (in blue) is completed or with a delay (in green). Note that the sizes are to scale, but that both SYCL executions would actually be a bit shorter than what is displayed for the sake of clarity (in consistency with Figure 5.2).

Taking into account that the cost of the utilised GPU is lower than the CPU[1], and that the performance with only two cores joined with the GPU provides a similar level of performance to the one achieved when using all cores, it seems reasonable to say that a cost-effective solution would be using a more affordable CPU together with an medium-range GPU. All these considerations are based on the performance of the current implementation of SYCL provided by Codeplay®, which is provisional and hence its performance has not yet been completely optimised. Therefore, the benefits of using GPUs to accelerate some computations could be larger than the ones presented before.

6.1.2 Interpolation-based spike detection algorithm

Overall, the new implementation performs much more efficiently the algorithm when compared to the original *C#* implementation. Using optimised functions for each subtask allows the developer to conveniently focus on each calculation more easily, identify redundant computations, and distribute the work to make a better use the data distributed in memory. Moreover, using a more object oriented approach can facilitate the task of optimising the code for the compiler. Nevertheless, the developed implementation has been implemented, so far, as a proof of concept to show an example of parallelisation and integration of SYCL kernels on spike detection algorithms. Thus, the algorithm is not identical to the original implementation, albeit it should be feasible to achieve the same results with simple extensions and a tuning of the parameters thanks to its modular architecture.

In all cases, the benefits of hyper-threading are pretty limited. Jumping from 6 (real) cores to 12 (virtual) cores does not present a significant speed-up even though being slightly faster. As stated previously, different kinds of overhead diminish its performance speed-up to the level where using the maximum number of virtual cores might not necessarily be the best configuration. In addition, when using a number of cores slightly above the number of real cores (e.g. 8), the performance may actually be somewhat worse compared to using only the total number of physical cores in the CPU. Finally, the SYCL implementation displays a marginally sub-linear scaling with the number of frames per iteration because all tests suffer from a approximately sta-

[1]The recommended retail price of the Intel® Xeon® CPU E5-2630 v2 started at 612.00 $ at the time of release. For the AMD FirePro W5100, the retail price at its release was of approximately 400 $.

ble overhead coming from the initialisation and copying of variables to the device (i.e. GPU), which is relatively exacerbated when the number of frames per iteration is small.

6.2 Future work

Following the same method shown for the delayed output of the online algorithm, it could be interesting to see how this concept could be exploited in the delayed computation of tasks in the interpolation-based algorithm when used in combination with the GPU. That is, maximise the throughput by delaying the spike detection part to be performed while the GPU computes the interpolations of a consecutive iteration.

In addition, this project could be used as a starting point to be generalised to other chips and platforms in general in order to create a python library of spike detection algorithms implemented in *C++*.

Conclusively, as soon as implementations of SYCL become available to the public and their device support expands to include NVIDIA GPUs and Intel/AMD integrated graphics, it would be interesting to see how well the developed implementations perform on such new architectures. The case of integrated graphics is of a special interest because it could provide a speed-up when comparing with using only CPU cores, but without having to rely on a separate device (i.e. GPU). The benefits include using the same memory instead of copying the data to and from the VRAM in a GPU and still, use two computing devices at once. This would require performing an extensive profiling of the resources utilised during the executions of the algorithms in terms of memory usage, disk input/output and CPU/GPU load. In addition, the distribution of the work and parallelisation of certain tasks shown in this project together with the aforementioned profiling could be used to aid in the design of hardware aimed at specifically performing the task of detecting spikes in high density electrophysiological data.

Bibliography

Azami, H., Escudero, J., Darzi, A., and Sanei, S. (2015). Extracellular spike detection from multiple electrode array using novel intelligent filter and ensemble fuzzy decision making. *Journal of neuroscience methods*, 239:129–138.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

Blanche, T. J., Spacek, M. A., Hetke, J. F., and Swindale, N. V. (2005). Polytrodes: high-density silicon electrode arrays for large-scale multiunit recording. *Journal of neurophysiology*, 93(5):2987–3000.

Dayan, P. and Abbott, L. F. (2001). *Theoretical neuroscience*, volume 10. Cambridge, MA: MIT Press.

De Oliveira, P. G., Queiroz, C., and Da Silva, F. L. (1983). Spike detection based on a pattern recognition approach using a microcomputer. *Electroencephalography and clinical neurophysiology*, 56(1):97–103.

Gray, C. M., Maldonado, P. E., Wilson, M., and McNaughton, B. (1995). Tetrodes markedly improve the reliability and yield of multiple single-unit isolation from multi-unit recordings in cat striate cortex. *Journal of neuroscience methods*, 63(1):43–54.

Harris, K. D., Henze, D. A., Csicsvari, J., Hirase, H., and Buzsáki, G. (2000). Accuracy of tetrode spike separation as determined by simultaneous intracellular and extracellular measurements. *Journal of neurophysiology*, 84(1):401–414.

Hilgen, G., Sorbaro, M., Pirmoradian, S., Muthmann, J.-O., Kepiro, I., Ullo, S., Ramirez, C. J., Maccione, A., Berdondini, L., Murino, V., et al. (2016). Unsupervised spike sorting for large scale, high density multielectrode arrays. *bioRxiv*, page 048645.

Lewicki, M. S. (1998). A review of methods for spike sorting: the detection and classification of neural action potentials. *Network: Computation in Neural Systems*, 9(4):53–78.

Maccione, A., Gandolfo, M., Massobrio, P., Novellino, A., Martinoia, S., and Chiappalone, M. (2009). A novel algorithm for precise identification of spikes in extra-

cellularly recorded neuronal signals. *Journal of neuroscience methods*, 177(1):241–249.

Maccione, A., Gandolfo, M., Zordan, S., Amin, H., Di Marco, S., Nieus, T., Angotzi, G. N., and Berdondini, L. (2015). Microelectronics, bioinformatics and neurocomputation for massive neuronal recordings in brain circuits with large scale multielectrode array probes. *Brain research bulletin*, 119:118–126.

Muthmann, J.-O., Amin, H., Sernagor, E., Maccione, A., Panas, D., Berdondini, L., Bhalla, U. S., and Hennig, M. H. (2015). Spike detection for large neural populations using high density multielectrode arrays. *Frontiers in neuroinformatics*, 9.

Nenadic, Z. and Burdick, J. W. (2005). Spike detection using the continuous wavelet transform. *IEEE Transactions on Biomedical Engineering*, 52(1):74–87.

Novellino, A., Chiappalone, M., Maccione, A., and Martinoia, S. (2009). Neural signal manager: a collection of classical and innovative tools for multi-channel spike train analysis. *International Journal of Adaptive Control and Signal Processing*, 23(11):999–1013.

Purves, D., Augustine, G. J., Fitzpatrick, D., Katz, L., LaMantia, A.-S., McNamara, J., and Williams, S. (2001). Neuroscience. *Sunderland, Massachusetts, USA*.

Quiroga, R. Q., Nadasdy, Z., and Ben-Shaul, Y. (2004). Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. *Neural computation*, 16(8):1661–1687.

Rey, H. G., Pedreira, C., and Quiroga, R. Q. (2015). Past, present and future of spike sorting techniques. *Brain research bulletin*, 119:106–117.

Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23-2, pages 392–403. ACM.

Ventura, V. and Gerkin, R. C. (2012). Accurately estimating neuronal correlation requires a new spike-sorting paradigm. *Proceedings of the National Academy of Sciences*, 109(19):7230–7235.