I, Brian Ervin, hereby submit this original work as part of the requirements for the degree of Master of Science in Electrical Engineering.

It is entitled:
Neural Spike Detection and Classification Using Massively Parallel Graphics Processing

Student’s name: Brian Ervin

This work and its defense approved by:

Committee chair: Ali Minai, Ph.D.
Committee member: J. Adam Wilson, Ph.D.
Committee member: Fred Beyette, Ph.D.
Neural Spike Detection and Classification Using Massively-Parallel Graphics Processing

A thesis submitted to the

Graduate School of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

in the School of Electronic and Computing Systems
of the College of Engineering and Applied Science
by

Brian Ervin
B.A. in Mathematics, Wittenberg University

Committee Chair: Ali Minai, Ph.D.

August 5, 2013
Abstract

A Brain-Computer Interface (BCI) is a direct communication line that bypasses the neuro-muscular pathway and allows brain signals to directly control a program or neuroprosthetic device in a closed loop system. Advancements in electrode fabrication techniques using biological microelectromechanical systems (BioMEMS) can produce arrays with hundreds or thousands of channels, providing much better control over the system. Unfortunately, traditional real-time computing techniques are outpaced by the flood of input from these electrode arrays. However, the advent of general purpose graphics processing units (GPG-PUs) for inexpensive, massively parallel processing allow for programs capable of handling thousands of channels real-time.

This thesis describes a filter, spike detector, and spike sorter for real-time processing of real and simulated neural recordings, speed and accuracy comparisons between the traditional multi-core CPU algorithms and the many-core GPU algorithm. The algorithm will be implemented and released as open source for use with BCI2000, a free general-purpose BCI system.
# Contents

1 Introduction .................................................. 1  
   1.1 Background .................................................. 1  
   1.2 Problem Description ....................................... 5  
   1.3 Research Objectives ....................................... 7  
   1.4 Thesis Outline ............................................. 7  

2 Background .................................................. 8  
   2.1 Brain-Computer Interface (BCI) ........................... 8  
   2.2 Computing .................................................. 9  
   2.3 CUDA ....................................................... 12  
      2.3.1 History ................................................. 12  
      2.3.2 GPU Architecture .................................... 13  
      2.3.3 CUDA Code Syntax .................................... 15  
   2.4 Neurobiology Background .................................. 15  
      2.4.1 Anatomy and Physiology ............................... 16  
   2.5 Signal Acquisition ........................................ 22  
      2.5.1 EEG .................................................... 22  
      2.5.2 ECoG ................................................... 22  
      2.5.3 Extracellular Action Potentials ...................... 23  
   2.6 Related Work on Spike Processing ......................... 24  

3 Methods ..................................................... 26  
   3.1 CPU vs. GPU Implementation Overview ................... 26  
   3.2 Algorithms and Implementations ........................... 28  
      3.2.1 Data .................................................... 28  
      3.2.2 Overview .............................................. 29  
      3.2.3 Filter .................................................. 29  
      3.2.4 Spike Detector ........................................ 30  
      3.2.5 Classifier .............................................. 32  
   3.3 Hardware .................................................. 34  
   3.4 Software ................................................... 35  
   3.5 Testing .................................................... 35
## Results

4.1 Accuracy .................................................................................. 36
  4.1.1 Recorded Dataset .............................................................. 37
  4.1.2 Results on the Generated Dataset ...................................... 39
  4.1.3 Accuracy Comparisons ...................................................... 40
4.2 Processing Time ....................................................................... 41
  4.2.1 CPU Multiprocessor Results ............................................. 42
  4.2.2 GPU Results ................................................................... 42
  4.2.3 CPU vs GPU Results ...................................................... 43
4.3 Discussion ................................................................................. 45

## Conclusions

5.1 Project Overview ................................................................. 47
5.2 Future Work ........................................................................... 49

## Source Code

A.1 Main Function ....................................................................... 55
A.2 Reduction ............................................................................ 56
A.3 Detector ............................................................................... 57
  A.3.1 CPU Detector ............................................................... 57
  A.3.2 GPU Detector ............................................................... 58
A.4 Sorter .................................................................................. 59
  A.4.1 CPU Sorter ................................................................... 59
  A.4.2 GPU Sorter ................................................................... 61
Chapter 1

Introduction

1.1 Background

Brain-Computer Interfaces (BCIs) provide a direct interface to the brain, bypassing the neuromuscular input and output pathways and allowing users to control a neuroprosthetic system using brain activity. BCIs have been used for neural prosthetics, medical treatment and diagnostics, communication, entertainment, and basic neuroscience.

Neurons (brain cells) represent and communicate information by firing electrochemically generated voltage spikes called action potentials [1], with information carried in both individual spikes and in the instantaneous rate of spiking, which is termed the firing rate. Thus, the most important requirements for a BCI system are access to the spiking activity of neurons in the brain’s motor regions and the accurate interpretation of this activity. Both present serious challenges at the experimental and computational levels.

While many regions of the brain are involved in the inception and control of movement [2], the motor cortex (M1) is generally considered to be the primary output region for control signals to the brainstem and spinal cord. The motor cortex has millions of neurons devoted to motor control, and Georgopoulos et al. demonstrated that activity from a relatively small population of neurons in the motor cortex can accurately predict the direction and velocity of intended reaching movements [3]. This work established the theory of population coding, which has become the basis for one of the most popular approaches to microelectrode-based BCIs. By recording from the motor cortex of rhesus monkeys, it was found that individual
neurons’ firing rates were broadly tuned to preferred direction of arm movement: neuronal firing rates followed a cosine curve as a function of the angle from the preferred direction. Figure 1.1 shows an example of this tuning. While the activity of a single neuron such as this is only an approximate encoder of direction, the tuning-weighted sum of activity from a population of neurons provides a very accurate representation [3, 4, 5]. Subsequent experiments have confirmed and extended this observation, showing that neural codes can be found in several brain regions for complex movements [4, 6, 7], sequential movements [8, 2, 9] and both kinematic variables (e.g., direction, curvature, etc.) and kinetic quantities (e.g., force, velocity, acceleration, etc.).

Figure 1.1: Left: Firing rates of a single neuron during five repetitions of motion in each of eight directions, aligned at the initiation of motion. Right: Sinusoidal directional tuning curve for the same neuron [3]. Used with permission

The conclusion from these results is that accurate observation of spikes from a sufficiently large population of neurons in M1 and/or other motor regions can be used to infer intended action, which in theory could generate control commands for a prosthetic device such as an artificial limb.

There are many methods for measuring electrical activity in the brain. In human research, the electroencephalogram (EEG) is the most popular due to its speed and non-invasiveness. People undergoing surgery and monitoring for medically intractable epilepsy often have an intracranial EEG called electrocorticogram (ECoG), and are often temporary participants in
BCI research. The ECoG signal has better signal amplitude and bandwidth, which are crucial for a BCI system. However, ECoG is far more invasive than EEG, and is therefore limited to patients already undergoing neurosurgery. EEG and ECoG have similar characteristics: each records the summation of the excitatory and inhibitory potentials of millions of neurons, and reflects a high-level view of regional brain activity from which it is impossible to extract the activity of individual neurons.

Another type of brain signal is acquired from penetrating microelectrodes. These electrodes are placed into the cortex, and record activity from one or several nearby neurons. Neuronal spikes are transient bursts of ion currents through the cell membranes. Microelectrodes measure the extracellular ionic gradients resulting from spike generation as well as the local field potentials generated when the spike from a neuron excites the membrane of a cell with which it makes a junction, or synapse. It is possible for a single electrode site to record from multiple neurons, each of which will have a distinct waveform depending on its spatial relationship to the electrode.

High-density microelectrode arrays (MEAs) utilizing biological microelectromechanical systems (BioMEMS) have been developed to record from a large neuronal population within a small region. Though the brain has no pain receptors, it is still very important to minimize damage to brain tissue and to avoid any shifting. Electrode arrays such as the Michigan Probe [10, 11] and the Utah Array [12, 13] are popular sensors with rapid batch fabrication on silicon, making fabrication possible with no more than common semiconductor microfabrication tools.

The Michigan Probe is a multidimensional penetrating electrode array with multiple electrodes on each shank so that different layers of neurons may be recorded. Additionally, several arrays may be attached to a mount and used as a three dimensional array [15, 14]. The Utah Electrode is an array of microneedles that penetrate deep into the cortex. It is a two dimensional array with sensors only at the tips of the needles, but the array fits one hundred electrodes in just over a square centimeter. These innovations in high density
electrode recordings allow for more acute control and a wider range of commands in BCI applications.

In primate experiments, microelectrode-based BCIs typically provide superior signals compared to EEG-based BCIs [16]. An effective spike-based BCI system must be able to consistently and accurately detect spikes on multiple electrodes and to assign them to the correct neurons. A real-time BCI must do this within a very short time, often tens of milliseconds. Most current BCI systems perform spike detection using dedicated hardware, i.e., an external signal processing system connected to a computer via a high-speed interface.
These dedicated systems cost tens of thousands of dollars or more, are inflexible in terms of developing custom algorithms, and lack portability. However, even modern computer processors are unable to process high-bandwidth data streams quickly enough for real-time performance. Therefore, the aim of this work is to develop a robust, real-time spike-based BCI utilizing massively parallel processing capabilities on modern graphics processing units.

1.2 Problem Description

In order to measure neuronal firing rates, the extracellular voltage spikes that occur when a neuron fires must be effectively detected, i.e., separated from background electrical noise, and sorted, i.e., grouped with similar neural waveforms. A common spike detection method uses thresholding, where any value that crosses a set threshold above the noise floor is considered of interest. A single positive or negative voltage threshold is typically used. Alternatively, the data may be preprocessed to find the instantaneous energy of the signal using the nonlinear energy operator [17], in which case an energy threshold is used. The difficulty with any threshold technique is that the signal strength and background noise may vary from electrode to electrode, or even on a single channel over time. Setting a static threshold for all channels may find spikes, but it will likely be inaccurate for many recordings. One simple strategy to improve detection is to set a dynamic threshold based on a preliminary analysis of the dataset.

In order to prepare the detected spikes for sorting, spikes must also be aligned with each other, which makes it important to detect both the presence and time of each spike accurately. The most popular approach is to align the peaks, but other techniques have been proposed as well, such as aligning the point of the steepest slope [18]. The alignment method should be chosen to maximize the separation for the sorter (see below).

Microelectrodes are often in proximity to multiple neurons, and therefore sense action
potentials and local field potentials from more than one source. In order to extract the firing rates for each neuron, the spikes must be sorted into different classes. The spike waveform shape and amplitude varies based on the sensor’s orientation with respect to each neuron, thereby allowing multiple spikes to be extracted from a single data channel. If electrode positions and orientations are stable relative to nearby neurons, spikes from the same neuron are more likely to resemble each other than spikes recorded from other neurons, suggesting that spikes can be assigned to neurons through a feature-based pattern recognition process. Thus, spike sorting algorithms typically consist of two primary steps: feature extraction and classification. Feature extraction finds characteristics of the spikes that are distinguishable between classes, and uses methods such as calculating spike width [18] or principal component analysis (PCA). Spike classification uses these features to distinguish between spike classes, and to group spikes with similar features into clusters. For example, the PCA scores for a group of spikes are used to determine which spike waveforms should be clustered together. Then, these cluster groups are used to classify new spikes based on their PCA score and proximity to existing clusters.

In addition to specifying the correct number of classes, other considerations when sorting spikes include reconciling superposition, miscategorization, electrode shift, and computational complexity. With multiple neurons firing, more than one may fire at about the same time, so that a detected spike waveform may represent more than one spike. In this case, one or more spikes may be ignored, miscategorized, or thrown out. Robustness against such errors is an important attribute of a successful spike processing algorithm.

Electrode shift or insufficient training may result in poorly defined clusters. This could result in miscategorization of spikes, even over the lifetime of the recording. Successful sorters should have the capability to adapt and correct for these potential complications.

Lastly, computational complexity is a major issue, especially with sorters. It is typically the case that higher accuracy comes with higher complexity [18]. Real-time BCI has a future in mobile computing, but high power consumption and computationally intensive sorters will
limit this ability.

1.3 Research Objectives

As indicated by the above discussion, spike detection and sorting are non-trivial processes requiring attention to millisecond-scale data in real-time and in context. This is a problem with the immense amounts of data generated by even modest-sized electrode arrays. In this context, the research presented in this thesis has three aims:

• To develop computationally efficient methods for spike processing of multi-channel microelectrode data.

• To implement the methods as a highly parallel algorithm on a graphics processing unit (GPU)-based system to allow real-time processing.

• To create a viable, scalable framework for future parallel spike detectors and sorters

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides technical background from neuroscience and signal processing, as well as a discussion of important issues for GPU-based spike processing. The algorithm for spike detection and sorting is described in detail in Chapter 3. The results of applying the algorithm are presented and discussed in Chapter 4. Chapter 5 concludes the thesis with a summary of the research and suggestions for future work.
Chapter 2

Background

2.1 Brain-Computer Interface (BCI)

Brain-Computer Interface, also referred to as Brain-Machine Interface (BMI) or Mind-Machine Interface (MMI), forms a direct link between the brain and a computer. BCIs have been successfully used for neural input in systems, such as cochlear implants [19] and optical prosthesis [20], for decades.

In 1964, Walter et al. [21] described the contingent negative variation (CNV), which demonstrated an association between activity in the sensorimotor cortex and an expected movement, and thus is referred to as the “expectancy wave.” Later, this and other event related potentials (ERPs) were found to correlate to position and intensity of the planned movement. Tanji and Evarts’ 1976 work [8] reported on three monkeys trained to push or pull a lever based on the color of a signal. Neuronal discharge was shown to be strongly linked to the direction of planned movement, as shown in Figure 2.1.

In 1982, Georgopoulos et al. showed that the firing rates of motor cortical neurons follow a directional tuning curve, and modulate their firing rates based on the intended movement direction [3]. Thus, the movement direction can be reconstructed by observing the firing rates of a population of neurons (Figure 2.2). This work has been the basis for many BCI
Figure 2.1: Top: Histograms comparing the activity of a pyramidal tract neuron during the “pull” (left) and “push” (right) signals. Bottom: Rasters of twenty five trials, where each dot represents a spike and each row is a separate trial. Notice the increased firing rate in the “pull” and the decreased in the “push.” The changes in neuronal firing rates reached significant levels most commonly between 200 and 500 ms after the stimulus. The instruction stimulus was either a red or green lamp, directing whether to pull (red) or push (green) [8]. Used with permission

systems, including the one presented in this thesis. The predictive power of population coding is the motivation behind sorting spikes to find firing rates of individual neurons.

2.2 Computing

Research and development in real-time BCIs has been held back by the computational limits of PCs and DSP hardware. Larger numbers of electrodes give a more highly resolved picture of cortical activity, which in turn promises improved control of BCI applications, and recent trends have seen much higher numbers of channels for real-time BCI applications [23, 24, 25]. Often there is a tradeoff between the number of channels selected for control and the refresh period of the BCI system. Modern computers with multiple processing cores are capable of performing spike sorting with more than 100 channels; however, this does not allow enough
time to perform additional or concurrent analysis using a sufficient real-time system refresh rate (e.g., 50 ms) [26].

For decades, each generation of central processing units (CPUs) used in computers has provided roughly twice the computational capability of the previous generation as observed by Moore’s Law. Advances in fabrication methods allowed manufacturers to fit twice the number of transistors on an integrated circuit every few years. Under this regime, software developers commonly relied on hardware for speed-up rather than on optimization or innovation of their code, or on limiting the capabilities of their program. Unfortunately, fabrication is reaching its physical limit, and the end of Moore’s Law is expected within the decade [27].
Power dissipation issues and problems meeting the computational demands established by Moore’s Law have prompted the use of multiple cores and multi-threading in CPUs. While this is addressing the current demands, it is an impractical approach in the long-term. Even though instruction-level parallelism is written into the software, processing speed does not scale linearly with the number of processors or threads because of the overhead introduced by the parallelism [28]. Additionally, the per-operation power consumption of CPUs is challenging to mobile computing, which is the logical trajectory for BCI applications.

The shift to parallelism has not gone unnoticed, however. Since the 1980’s, computing on graphics processing units (GPUs) has been used to achieve speed-ups in parallelizable applications. Most recently, two programming interfaces, OpenCL (The Khronos Group)[29] and CUDA (NVIDIA) have simplified general-purpose program execution on both the CPU and GPU. These programming paradigms leverage the hundreds or thousands of cores available on modern GPUs for general computing applications. Many popular applications including Adobe Photoshop and Microsoft Internet Explorer now utilize GPGPU processing on supported hardware, and countless smaller applications are currently being developed and used [28].

With general purpose computing on graphics processing units (GPGPU), power dissipation is not an issue; using multiple cores was the solution to the heat problem in the CPU. Power consumption is much more efficient per operation on the GPU, and will be increasingly so with each new generation, according to NVIDIA’s projections [30]. Additionally, a parallel approach allows for scalability for future applications. Thus, a computationally intensive, massively parallel application, such as spike sorting on hundreds, or even thousands of channels, is an ideal candidate for GPGPU. This thesis will present and evaluate a scalable CUDA-based spike sorting algorithm, which will be evaluated using actual recordings and generated data to determine its limit for real-time execution.
2.3 CUDA

2.3.1 History

For large-scale, numerically intensive applications, GPGPU (general-purpose computing on graphics processing units) has been sparsely utilized as early as the 1980’s [28]. Graphics processing units have evolved from drawing lines, arcs, and bitmaps, to running general-purpose code, having their own application programming interfaces (APIs), even having devoted silicon to aid in parallel programming. Early efforts at hardware acceleration, such as WinG and DirectDraw, were focused on achieving peak performance in 2D graphical applications. As gaming became more popular and 3D graphics became the standard, the OpenGL and Direct3D APIs took over. These quickly developed the capability for general purpose programming, and paved the way for the use of the parallelism of GPUs in non-graphics applications. Both OpenCL and CUDA have created languages which allow for programs to run on both the CPU and the GPU [28].

CPUs and GPUs have been designed for optimizing different types of calculations, so the structure of CUDA and OpenCL programs should reflect these differences. CPUs have been optimized for high speed sequential execution, and therefore are designed with complex control logic for in-order or seemingly in-order execution, and low latency speeds for data access and instruction due to their large cache memory. GPUs, however, have been designed to handle a processing pipeline capable of many parallel floating point operations with little command divergence (e.g., branching). Many graphics operations can be expressed as matrix operations which are inherently parallel in nature. Therefore, GPGPU processing can be leveraged successfully in situations requiring simple, repetetive operations on large datasets for which a single operation can be applied to thousands of elements simultaneously.
2.3.2 GPU Architecture

GPU processing differs from CPU processing in a number of ways. CPU compilers typically abstract away memory transfers from slow device ram to CPU cache and registers, allowing the developer to focus on the algorithm rather than very low-level implementation details. On the GPU, however, the developer must explicitly define which memory to use, and manually transfer data to and from local caches and registers to obtain peak performance. This introduces a level of complexity that can be difficult for a novice GPU developer to comprehend, often resulting in poor GPGPU performance.

A second consideration is memory bandwidth and transfer rate. Data must be transferred from the CPU to the GPU for processing, which introduces an overhead. An important consequence is that the memory bandwidth overhead may be greater than any computational gains, invalidating the purpose of GPU execution. The data transfer rate for a modern GPU is roughly 16 GB/s. This can be masked by combining memory transfer and execution; that is, the device can execute code for one set of instructions while transferring and preparing memory for the next instruction set [28]. Additionally, the memory bandwidth to transfer from GPU device RAM to local caches and registers is roughly ten times faster than that of a comparable CPU. As of this year, NVIDIA’s GeForce GTX 670 and above offer 192.3 GB/s [31], compared to the Intel Xeon E5-2690 (best performing high-end CPU [32]) with a memory bandwidth of 51.2 GB/s [33].

Furthermore, data transfer overhead can be reduced by storing data in device (GPU) memory without transferring it back to the host (CPU) memory, and passing these results to subsequent GPU functions. Only the final result is returned to the CPU.

Additionally, thread reading and writing of the device “global memory” – a large memory bank which is accessible to each thread across the device which also may be read and written to by the host – should be minimized. A call to global memory tends to have long (hundreds of clock cycles) latencies, which might not be masked by number of executing threads in the case of many of them trying to read simultaneously. The resulting back-up may leave whole
grids (groups of thread blocks) idle.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Global</th>
<th>Constant</th>
<th>Shared</th>
<th>Register</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Grid</td>
<td>Grid</td>
<td>Block</td>
<td>Thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Speed</td>
<td>Slow</td>
<td>Fast</td>
<td>Fast</td>
<td>Very Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Size</td>
<td>Large</td>
<td>Small</td>
<td>Small</td>
<td>Very Small</td>
<td>Large</td>
</tr>
<tr>
<td>Permission</td>
<td>R/W</td>
<td>Read Only</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Other memories, shown in Figure 2.3 and described in Table 2.1, have been created for more efficient use; each with their own advantages and disadvantages. Aside from global memory, there is constant memory, shared memory, register memory, and local memory. Shared memory is a low-latency memory with read/write access for all threads within a block. This can be used to share intermediate results within a kernel with other threads, but size is an important consideration. The maximum shared memory per block is only 16-48 KB, depending on the device. Lastly, both register and local memories are per-thread.
These keep track of private, thread-specific variables. Register memory is by far the fastest access memory; at least 3-6x speedup over shared memory and at least 23-45x faster than global memory access [34]. These are extremely valuable, but also very limited; as few as 10 to 30 32-bit registers are available to each thread, depending on the device, if the maximum number of threads are used in a streaming multiprocessor. Local memory, on the other hand, is typically for per-thread arrays, and is much slower than register and shared memory access. It is best to avoid use of local memory if at all possible (e.g., by using registers for small arrays). The amount of local memory available to each thread was increased significantly in CUDA compute capability 2.0 and up, from 16 to 512 KB. Register, local, and shared memories only persist for the lifetime of the kernel [28]. For a full description of NVIDIA GPU device architecture, see the CUDA Toolkit Handbook [28].

2.3.3 CUDA Code Syntax

Code written for CUDA is standard C, with several extensions for GPU computing. For example, within the GPU kernel it is possible to determine the thread number, block number, and grid location. There are also functions for thread synchronization, and keywords to specify memory location for variables. Discussion of the CUDA code syntax is beyond the scope of this work, and the reader is referred to the CUDA C Programming Guide [35].

2.4 Neurobiology Background

Brain-Computer Interface requires an understanding of the anatomy and physiology of the nervous system, of the signals produced, and of the methods used to record those signals. After a brief review of the whole nervous system, this section will concentrate on the anatomy of the cerebral cortex, neuronal transmission, and signal acquisition methods.
2.4.1 Anatomy and Physiology

The nervous system can be divided into the central nervous system (CNS) and the peripheral nervous system (PNS). The PNS can be divided into the somatic PNS and the visceral PNS. The somatic PNS innervates the voluntary muscles. It sends and receives signals from the spinal cord. The visceral PNS (also called the autonomic nervous system) regulates the autonomous functions of the body such as digestion, secretion from various glands, and heart rate. Like the somatic PNS, it has both sensory and motor branches, bringing information such as blood pressure to the CNS and sending commands such as contraction and relaxation to the blood vessels [36].

The CNS also can be divided into two main parts: the brain and the spinal cord. The spinal cord is enclosed in the vertebrae and communicates with the PNS via the dorsal and ventral roots. The dorsal roots bring information into the CNS and the ventral roots send information out to the body [36].

Gross Anatomy of the Brain

The brain can further be divided into three main parts: the cerebrum, the cerebellum, and the brainstem. The brainstem regulates the body’s vitals (breathing, consciousness, and temperature) and aids in communication between the cerebrum and cerebellum. It is the most primitive but also most essential part of the brain.

Among the cerebellum’s many functions is its contribution to movement control. It receives and processes proprioceptive information from the spinal cord and planned movement information from the cerebral cortex via the pons. The cerebellum refines the required muscle contractions needed for the movement to achieve the goal. Unlike the cerebrum, the left cerebellum corresponds to the left side of the body, and the right to the right side.

Finally, the cerebrum is, in humans, the largest and the uppermost part of the brain. It can be divided further into the cerebral cortex and subcortical structures, notably the basal ganglia. The cerebral cortex is the region of the brain that has seen the most rapid
development in recent mammalian evolution, and is a dominant part of the human brain. It is deeply involved in all sensory functions, motor control, sensorimotor integration, memory and all higher functions such as judgment and executive control. The most striking feature of the cerebral cortex is its clear division into hemispheres by the longitudinal cerebral fissure - the so-called “left brain” and “right brain”. It also has several grooves (sulci) and four distinct lobes: frontal, parietal, occipital and temporal. While these can vary widely between individuals, there are some features that are common to all individuals. The central sulcus is a coronal groove which separates the frontal lobe from the parietal lobe. The first gyrus anterior to the central sulcus is called the precentral gyrus, and the first posterior the postcentral gyrus. The lateral or Sylvian fissure is an especially deep groove travelling transversely on both the left and the right of the brain, just below the terminal point of the central sulcus. This fissure separates the temporal lobe from the frontal and parietal lobes. Just below the lateral fissure is the superior temporal gyrus. These features are shown in Figure 2.4.

The precentral and postcentral gyri are especially important in BCI applications involving...
movement. The postcentral gyrus is the primary somatosensory cortex (S1), where the ‘touch’ sensory information is processed. The precentral gyrus has come to be known as the primary motor cortex (M1), and with the premotor area (PMA) and supplementary motor area (SMA), the areas immediately anterior to M1, this region is responsible for planning the movements of the skeletal muscles and is referred to as the motor cortex. In both the somatosensory and motor cortices, somatopic maps (maps that relate specific locations in the brain to correlating parts of the body) were found by Wilder Penfield in the 1950’s by using weak electrical stimulation [36]. The resulting maps are shown in Figure 2.5. Recent research has indicated that the mapping between the brain and the body is somewhat more complex, but the somatotopic maps are still a useful tool for BCI.

The basal ganglia, which is considered a subcortical region of the brain, is also critical for motor control because of its central role in action selection. It receives direct input from all parts of the cerebral cortex and projects back to it – especially to M1 and prefrontal cortex – via the thalamus, forming the so-called thalamo-cortical loops that are believed to be the fundamental building blocks of motor programs [38, 39, 40, 41]. The basal ganglia also provides the primary locus for the learning and regulation of actions through reinforcement via the neurotransmitter dopamine [42, 38].

Neurons

At a cellular level, the activity that is eventually recorded originates from neuronal communication. Neurons typically maintain an electrical voltage – called the membrane potential – between -50 and -100 mV – across their cellular membrane, but this can vary in response to signals received from other neurons. When sufficiently excited, neurons generate spikes (action potentials) that travel over output structures called axons to other neurons. Axons contact other neurons either on the cell body (soma) or on input structures called dendrites. The connection is termed a synapse, the source neuron is called the pre-synaptic neuron and the receiving neuron the post-synaptic neuron. The arrival of an action potential at a synapse
results in the generation of a transient voltage change called a \textit{post-synaptic potential} (PSP) in the post-synaptic neuron. The PSP can be positive (excitatory) or negative (inhibitory) depending on the type of the pre-synaptic neuron. The typical neuron, seen in Figure 2.6, has many dendrites coming into the body of the cell but only one axon leaving. The axon has uniform width throughout, and branches at right angles, if it branches at all [36].

While the dendrites are relatively short (around 2mm or less), a single axon may extend
over a meter within the body. PSPs generated in the dendrites travel indiscriminately both towards and away from the soma and also suffer from leakage as they make their way to the soma. The soma integrates the PSPs reaching it, thus changing the voltage of the neuron at the origin of the axon. Whenever this reaches a certain threshold, the neuron sends an action potential down the length of the axon to its synapses, where it releases chemicals called neurotransmitters. These chemicals are picked up by the post-synaptic neurons and result in PSPs in these neurons. Many axons are insulated by a sheathing called myelin, with periodic breaks called the “nodes of Ranvier.” The myelin sheath protects the signal from leaking, and the nodes essentially recharge the action potential by allowing for ion exchange across the membrane [36]. This allows axons to carry action potentials over long distances (up to several feet) with little attenuation.
Synaptic Transmission

The end of the axon that terminates at a synapse is called the *presynaptic axon terminal*, and the small space between the axon terminal and the postsynaptic dentrite is called the *synaptic cleft*. The basic anatomy of the synapse is shown in Figure 2.7. When an action potential reaches the axon terminal, synaptic *vesicles* (sacks containing neurotransmitters) release a neurotransmitter into the synaptic cleft, where it can bind to *receptors* on the postsynaptic neuron. This chemical binding creates the post synaptic potential (PSP), which is then propagated through the dendrite to the soma. Post synaptic potentials can be either excitatory (EPSP) or inhibitory (IPSP), depending on the neurotransmitter’s action on the post-synaptic receptors. *Excitatory post-synaptic potentials* (EPSPs) increase the membrane potential of the post-synaptic cell, bringing it closer to the threshold required for an action potential. In contrast, *inhibitory post-synaptic potentials* (IPSPs) decrease the membrane potential, thus inhibiting action potentials [36].
2.5 Signal Acquisition

Several methods have been used for medical imaging of the brain or regions of the brain, but only a few are capable of real-time long-term recording required by BCI. The three standouts are electroencephalogram (EEG), electrocorticogram (ECoG), and neuronal recording of extracellular action potentials.

2.5.1 EEG

EEG is a non-invasive strategy for neural recording which is simple, fast, and inexpensive, making it the most popular recording method and the most common approach to BCI systems. Electrodes are placed at specific locations on the scalp, using conductive gel, paste, or saline solution, typically in the standard 10-20 system as described in [43].

Before reaching the EEG electrodes, the electrical signals from the brain must penetrate from the cortex, through the meninges, the skull, and the scalp. Because of the distance from the source and the weak signal, the spatial resolution of EEG is very poor. A single electrode on the scalp actually detects the summed activity of thousands or even millions of neurons. As described previously, an action potential is a rapid change in membrane potential, usually lasting only a millisecond and traveling rapidly over long distances down an axon. These do not affect the field potential enough to be detected on the scalp. The post-synaptic potentials, however, are more widespread, as one axon may interact with many other neurons, rapidly multiplying a synchronized signal throughout a region. Thus even though a PSP is significantly smaller in amplitude than an action potential, it is the temporal summation of EPSPs and IPSPs that is detected by an EEG electrode [22].

2.5.2 ECoG

Electrocorticogram (ECoG) also records large scale neural activity, but its electrodes are placed underneath the skull, either above the dura mater (epidural) or just below it (sub-
The electrodes are contained within a transparent silastic sheet and are arranged in either a strip or grid configuration, spaced 0.5 to 1 cm from each other. Because of the invasiveness of ECoG electrodes, their use is much less common, typically limited to patients with medically intractable epilepsy. In these cases, the electrodes are implanted and record seizure activity with the goal of localizing the seizure focus, and recordings last nonstop for 3 days to 2 or more weeks [22].

Despite their invasiveness, ECoG electrodes are still an excellent choice for BCI applications. Many of the sources of artifact in EEG (such as EMG, ECG, and other muscular movements) are decreased because the electrodes are beneath the skull and scalp. Because ECoG electrodes are much closer to the neural activity, the signal is much stronger and the spatial resolution is much greater than in EEG, resulting in a larger signal to noise ratio and less electrode to electrode overlap. As with EEG, ECoG records only from PSPs, but the location of the electrodes allows for higher frequency signals, providing more information from localized sources. This means high-resolution recordings of gamma (30-70 Hz) or even high-gamma (>80 Hz) neural oscillations are possible. Recent work [44] has shown that high-gamma power band may have a high correlation to movement or the intent to move, which would make ECoG an even more powerful tool for BCI applications [22].

2.5.3 Extracellular Action Potentials

The Utah array and Michigan probe, referred to in Section 1.1, perform neuronal recording, the highest resolution and most invasive strategy for acquiring neural signals. These record the extracellular potential changes of individual cells, and have been used in vitro and in vivo for the study of neural networks, electrical propagation, and action potentials [36]. The first neural recording techniques involved micropipets, one inside the cell and one in the extracellular space, recording the fluctuations in the potential difference across the membrane. Various clamping techniques, such as voltage clamping and patch clamping, allow for studies of the ion channels. Such intracellular techniques do not lend themselves to BCI applications,
since they cannot be performed long-term, and give a very small picture of cortical activity [22]. Microelectrode arrays, however, give a very high resolution picture across dozens or hundreds of neurons in a local population, and can be implanted for long-term continuous recording [45, 11].

2.6 Related Work on Spike Processing

As discussed in chapter 1, the research in this thesis focuses on spike detection and spike sorting for signals recorded with microelectrode arrays. Many different approaches have been used for spike detection and spike sorting [18, 46]. Detection algorithms vary widely, but most use either a threshold or some form of template matching [47]. Threshold detection methods include a simple threshold, absolute-value threshold, an energy-domain threshold, and applying a threshold to the discrete wavelet transform. This project uses a parallel version of the absolute value threshold for spike detection.

Almost all spike sorters use the same strategy: extraction of features followed by clustering. However, the methods for feature extraction and clustering are highly varied. Early forms of feature analysis compared spike width or peak-to-peak differences to find clusters. Principal component analysis has become very popular, but in order to function in real-time, the principal components of the initial group of detected spikes are calculated off-line before the classification stage [18]. Other approaches, such as discrete wavelet transform, have been used to extract features [48]. There have been a wide range of clustering strategies used, ranging from something as simple as k-means to much more sophisticated and computationally intensive algorithms.

With increased acceptance of GPGPU technologies, several groups have started using parallel processing in neuroscience research. In work by Chen et al. [49], GPUs were used to measure synchronization between channels in EEG recordings. Their goal was to make existing EEG applications, such as localization of epileptic focus, faster in order to make
decision-making more timely and accurate. Their study allows complex analysis to run in less than an hour, as compared to several days. Rossant et al. [50] demonstrated automatically generated parallel algorithms from user-defined Python code for fitting arbitrary spiking neuron models to electrophysiological data. This is freely available to researchers who want empirically validated models. Cao et al. [51] used GPGPU for frequent episode mining from MEAs. Traditionally, mining neuronal circuits in spike train datasets required clusters of workstations or supercomputing capability, but the work of Cao et al. [51] has made this capability inexpensive and accessible.

In recent work by Wilson and colleagues [26, 52], a single threshold and k-means clustering approach has been applied to massively parallel spike detection and sorting. While k-means is a popular method of classification, the number of clusters has to be known a priori. In spike sorting applications, the number of classes corresponds to the number of neurons from which each electrode is recording. When dealing with hundreds, even thousands of electrodes, the number of neurons on each channel is impossible to guess accurately. The classification algorithm presented in this thesis auto-detects the number of classes and uses a very simple point-by-point correlation analysis, which can most closely be categorized as clustering without feature extraction. This method was chosen because of its simplicity, but also because of the limited operation capabilities of the GPU, where algebraic operations are very fast but transcendental operations are considerably slower. Thus, the method presented in this thesis is more customized to GPU implementation than previous approaches.
Chapter 3

Methods

3.1 CPU vs. GPU Implementation Overview

Converting sequential programs into parallel for execution on the GPU is nontrivial. In order to achieve optimal speedup, there are a number of factors to consider. First, most programs are not entirely parallelizable; there are aspects that are specifically sequential. Typical candidates for parallelization are large loops or segmented operations where order may not matter. Some operations are fairly straightforward, such as matrix addition, where each parallel thread may only do a few additions and there are no memory conflicts. Other operations are surprisingly more difficult than their sequential counterparts, e.g. summing an array. In a traditional sequential paradigm, summing an array only requires one “sum” variable and simply iterates through the array, adding each value. The best strategy for summing an array in parallel is called reduction, and is described in Algorithm 1 and Figure 3.1. Parallel implementations must take into consideration the possibility of memory access queueing, where more than one thread requests access to the same location in memory at the same time, which greatly slows down a program. Also, the current versions of CUDA require threads to act in “warps.” A warp is the unit of thread scheduling in the multiprocessor on the GPU. Warps are groups of 32 threads that operate together. So, for
Data: Array
Result: Sum of Array
find the largest power of two $P$ less than the size of the array;
while thread index $< \text{the power of two } P$ do
  copy value in the array at thread index $i_t$ to thread index $i_t$ in reduction array;
  increase thread index $i_t$ by number of threads $N$;
end
while thread index $< \text{size of the array minus the power of two } P$ do
  add the value in the array at the power of two plus the thread index $P + i_t$ to the
  value at the thread index $i_t$ in the reduction array;
  increase thread index $i_t$ by number of threads $N$;
end
// the number of sums $S$ is equal to the power of two $P$ found earlier
while number of sums $S > 1$ do
  while thread index $i_t < \text{half the number of sums } \frac{1}{2} S$ do
    add value in array at thread index $i_t$ and the value at thread index plus
    half the number of sums $i_t + \frac{1}{2} S$;
    increase thread index $i_t$ by number of threads $N$;
  end
  halve the number of sums $S$;
end

Algorithm 1: Parallel Reduction Overview

example, if one thread entered into an if loop and the other 31 did not, the other 31 in the
warp would wait for the one in the loop.

Because the threads execute in parallel, read/write operations must be used carefully. If
multiple threads are to add to a single point in memory, for example, the results would be
unpredictable and inconsistent. To illustrate this, two threads, $t0$ and $t1$, are both going to
iterate the same location in memory, $M$, which currently has the value 0. First, $t0$ gains read
access to the memory, and sees that the current value of $M$ is 0. When $t0$ releases the read
access, $t1$ also reads that the current value is 0. Meanwhile, $t0$ has added 1 to the value,
and now is looking for write access to $M$. As $t0$ writes the new, iterated value (so that the
value at $M$ is 1), $t1$ is iterating the value it read at $M$. After $t0$ releases write access to $M$, $t1$
gains access and writes the new, iterated value, 1. So after both $t0$ and $t1$ have iterated the
value at $M$, the new value at $M$ is 1. Parallel programs must be careful to avoid these race
conditions. Additionally, as mentioned in Subsection 2.3.2, the speed and size of different

27
3.2 Algorithms and Implementations

3.2.1 Data

Two datasets were used for testing: the first was an artificially generated dataset consisting of local field potentials and one or more spike waveforms. The sampling rate was set to 25000 Hz, amplitude was set to 1.2 $\mu$V, firing rate was set between 10 and 80 Hz, with a noise floor of 0.1 $\mu$V. This represents a normalized dataset, with a corresponding spike amplitude of 70 $\mu$V and noise floor of 6 $\mu$V. Three spike templates were read from a file and added to the dataset according to the firing rates, giving a filtered dataset with known timestamps and classifications.

The other dataset was a real, unfiltered neural recording; performance and accuracy

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>3</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: An example of parallel reduction. Four parallel threads are shown doing the first addition step ($t0 - t3$), then two in the second addition, and finally one thread finds the sum of the whole array. A much more detailed explanation, including execution times and optimization strategies, can be found in [53].

CUDA memories, as well as the cost of memory transfer to and from the GPU all must be considered.
were assessed by visual inspection of hand-scored spikes. Each dataset was tested for the algorithm’s accuracy, and the real dataset was used for channel-variant CPU vs GPU performance tests.

### 3.2.2 Overview

The algorithm has three main components: a filter, a spike detector, and a spike sorter. First, low-frequency (<300 Hz) signal components are removed with a high-pass filter. Next, an adaptive threshold function detects spikes. Finally, spikes are classified into existing spike groups, or new groups are created if an acceptable group is not found. Using the massively parallel GPU version, more than 1000 channels may be sorted simultaneously in real time (<50 ms), or a smaller number (e.g., 64-256 channels) may be processed very quickly, allowing additional analyses to be performed.

### 3.2.3 Filter

Typically, neural recordings have high-amplitude, low-frequency components that prevent thresholding. These must be removed through filtering before thresholding can be applied. A finite-impulse response (FIR) filter (effects shown in Figure 3.2) is available on both the CPU and GPU. The algorithm utilizes frequency domain convolution using the FFTW library on the CPU, and the CUFFT library on the GPU. A 128-point high-pass FIR filter with a 300 Hz cutoff band was generated, and the frequency domain filter coefficients calculated before signal processing. During processing, a block of data was converted to the frequency domain using the FFT library, multiplied with the filter coefficients, and converted back to the time domain with an inverse FFT. For GPU-processed data, the results are left on the GPU to reduce data transfer overhead.
3.2.4 Spike Detector

Spikes were detected using an adaptive threshold, such that spikes that crossed the threshold were selected. Both positive and negative thresholds were used to find spike peaks, since the waveform and direction can vary due to the neuron’s location relative to the electrode. The threshold can adapt continuously to new data using the standard deviation of previous data (e.g., 1 sec). This threshold can be continuously updated, set intermittently, or set manually. The standard deviation was derived on the GPU using parallel reduction (see Section 3.1 for details.) The standard deviation is scaled and saved to an array of the eight most recent scaled standard deviations. The current buffer’s threshold is determined to be the average of these eight values. This allows dynamic adaptation to changes in the noise floor while remaining robust to transient noise bursts.

Next, the filtered data is evaluated against the threshold to find the spike timestamps. On the CPU, the data buffer for each channel is iterated through to find peaks (i.e., values that are greater – or less, for the lower threshold – than both neighboring values and exceed the threshold). If it is at a peak, it then must check if it is the first peak both above the

Figure 3.2: Top: Raw extracellular data. Bottom: The same section of data after FIR filtering with low-frequency local field potentials removed. Note that the spikes are all still detectable, but thresholding is now possible.
upper threshold and below the lower threshold. If all of these tests are passed, the timestamp is recorded.

![Example of Parallel Threshold Detection](image)

![Spike Peak Detected by Thread 0](image)

Figure 3.3: Top: Parallel spike detection is done by each thread only evaluating a few datapoints to see if they cross the threshold. Bottom: When a thread finds a point crossing the threshold, it checks the surrounding datapoints to determine whether it is a local extremum.

On the GPU, the thresholds for each channel are copied to the threads’ registers, and the channel’s data is copied to a shared memory array. The number of spikes and the timestamps are saved in shared memory and eventually copied back to global memory. Each thread works simultaneously, checking whether the data point at their own index crosses the threshold, as shown in Figure 3.3. The whole data set is examined using the same incrementing strategy employed in calculating the average: adding the number of threads to the index of each thread. If a thread finds a point that crosses the threshold (bottom of Figure 3.3), it then checks whether it is at a local extremum, simply checking the two data on either side of
it. As in the CPU version, there are a few tests to verify whether the timestamp should be recorded: first, the points preceding the peak are checked for peaks in the opposite direction. Next, the data is searched for larger peaks in the same direction, both before and after the peak. If these tests are passed, the timestamp of the peak is recorded. The whole data set is stepped through in the usual fashion, by increasing the threads' indices by the number of threads in the thread block.

Detected spikes are aligned based on the peak of the first crossing. The purpose of aligning the first peak was to minimize the point-by-point correlation of different waveforms, whereas in a single threshold approach, for example, all waveforms would have an aligning positive peak or an aligning negative peak, which would make them less distinguishable using a correlation-based sorter.

### 3.2.5 Classifier

![Found Spike]

Figure 3.4: A found spike is compared to the existing spike classes. The class with the largest correlation is selected for the found spike.

The sorter calculates the correlation coefficient between a new spike and each spike tem-
plate stored for that channel. The correlation coefficient (Equation 3.1) can be calculated using a parallel reduction, making it suitable for GPU computation. The correlation is calculated between each new spike and all of the spike templates stored for that channel, shown in Figure 3.4. The template with the greatest correlation exceeding a threshold of 0.7 is selected as the match, and the new spike is added to the template waveform. If no template has a correlation exceeding 0.7, the new spike is used to create a new template. A decision tree of the classifier is depicted in Figure 3.5.

\[ R_{X,Y} = \frac{N \sum X_i Y_i - \sum X_i \sum Y_i}{\sqrt{N \sum (X_i^2) - (\sum X_i)^2} \sqrt{N \sum (Y_i^2) - (\sum Y_i)^2}} \] (3.1)

In Equation 3.1, \( N \) is the number of points in a spike waveform, \( X \) is the set of points of the spike being evaluated, \( Y \) is the set of points of a template waveform, and all sums are from 1 to \( N \). The equation for correlation coefficient can be simplified to that in Equation 3.2, where \( \sigma_X \) is the standard deviation of the points in the found spike and \( \bar{X} \) is the average of the points in the found spike. Both the CPU and GPU implementations use Equation 3.2 for simplicity and speed.

\[ R_{X,Y} = \frac{1}{N - 1} \sum_{i=1}^{n} \frac{X_i - \bar{X}}{\sigma_X} \sum_{i=1}^{n} \frac{Y_i - \bar{Y}}{\sigma_Y} \] (3.2)

Regardless of whether the algorithm is run on the CPU or the GPU, the timestamps are stored in a vector, in order to keep track of the firings of each class on each channel over time. Additionally, another function keeps track of the time since each template has had a spike added to it. If a template has been idle for a long enough time, it is deleted with all of its associated data. If, for example, a waveform with overlapping spikes was found early on and given its own template, after a time, it would be deleted, allowing for the “real” classes to be found. This also has the potential to reduce the workload of the sorter by requiring fewer correlation calculations.
3.3 Hardware

Testing was done on a custom-built desktop PC with dual AMD 2.4 GHz processors each with 12 cores, enabling up to 24 threads to run simultaneously. 32 GB of RAM were installed. This computer was equipped with an NVIDIA 480 GTX video card (see Table 3.1 for additional GPU information).

<table>
<thead>
<tr>
<th>GPU</th>
<th>Cores</th>
<th>Processor Clock</th>
<th>RAM</th>
<th>RAM Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 480</td>
<td>448</td>
<td>1215 MHz</td>
<td>4 GB</td>
<td>1674 MHz</td>
</tr>
</tbody>
</table>
3.4 Software

Each processing step (filter, detector, and classifier) was written as individual C++ classes. These classes can be used in a C++ program, and we intend to develop a spike sorting module for the general purpose BCI program, BCI2000. Additionally, the GPU code can be used in other environments such as Matlab and Python. The code is cross-platform compatible, and has been tested on Windows 7 64-bit, Ubuntu Linux 13.04, and OS X 10.8.

3.5 Testing

Four timers were used to time the filter, the spike detector, the spike sorter, and the total overall time. The median, tenth percentile, ninetieth percentile, and maximum processing times were recorded.

The dataset was a real, unfiltered LFP recording, which was compared against visual inspection for performance comparisons. Each dataset was tested for the algorithm’s accuracy, and the real dataset was used for channel-variant CPU vs GPU performance tests. Timing results were taken from a 10 second block of data using 50 ms blocks, for a total of 200 measurements.

The next chapter shows the results obtained for both datasets along with a discussion of these results.
Chapter 4

Results

The goals of the project presented here are:

- To develop methods for spike processing in multi-channel microelectrode data.
- To implement the methods as a highly parallel algorithm on a GPU-based system to allow real-time processing.
- To create a viable, scalable framework for future parallel spike detectors and sorters.

In order to achieve these goals, the detector and classifier methods must be accurate, the whole system must be able to function at high capacity below a 50 ms real-time threshold on the GPU, and the system must be suitable to scale with future hardware. The results in this chapter demonstrate the accuracy of the detector and classifier based on visual inspection of a real recording, and the objective accuracy based on a generated dataset. The processing times of the CPU and GPU implementations are presented and compared, illustrating the real-time capabilities of both.

4.1 Accuracy

The first step in determining whether the program accomplishes its goals is to test its accuracy. If the program processes thousands of channels, but does not detect and sort spikes
correctly, it is not practically useful. With that in mind, the detection and classification performance was evaluated in two ways: against a dataset obtained by recording LFPs from the brain, and against a generated dataset where the ground truth was known objectively.

When comparing against the recorded dataset, the ideal response is not known, either for the detection of spikes or for their classification. The electrode simply recorded local field potentials over time, and it is the goal of neural spike detectors and sorters to pull out the useful information. Nevertheless, the new algorithms need to be validated for such data. There are two ways to obtain the reference response for comparison: using other sorting programs to establish a “ground truth” (that is, what is defined to be the true timestamps and classes), or through visual inspection by a human expert. The second approach was used in this research. The first five seconds (125,000 samples) of the dataset were inspected, and the spike timestamps and classes were recorded. The results for this test are shown in Section 4.1.1.

The generated dataset is much easier to evaluate for accuracy: because the spikes are added by the generator within the program, their classes and timestamps may be recorded and used to compare results. Thus, testing against this data provides an objective assessment of the algorithm, albeit on an artificial – and possibly somewhat unnatural – dataset. The accuracy on the generated dataset is discussed in Section 4.1.2.

### 4.1.1 Recorded Dataset

To evaluate the recorded data, spike timestamps were detected and recorded by visually examining five seconds of filtered data. The program was then run on the same section of data using the dynamic threshold approach, and results were compared. As can be expected, there were both false negatives (spikes detected by visual inspection that were missed by the program) and false positives (spikes detected by the program that were missed by visual inspection). The detector’s accuracy is determined by the selectivity (also called sensitivity), which is the measure of how well all true spikes are detected, and by specificity, which
measures how well false spikes are rejected. In the five second dataset, 267 spikes were detected by hand. The accuracy results for the detector are summarized in Table 4.1. Of the 267 spikes identified by visual inspection, the algorithm missed 12 and detected 16 false spikes.

<table>
<thead>
<tr>
<th># Spikes</th>
<th># Detected</th>
<th>Specificity</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>267</td>
<td>271</td>
<td>94%</td>
<td>97%</td>
</tr>
</tbody>
</table>

The accuracy of the classifier was evaluated next. Rather than compounding the errors from the detector, the classifier used all the spikes found by the detector. Because there are no definite spike classes, all 271 spikes were overlaid on the same plot (left of Figure 4.1), and the visual inspection was based on the three apparent classes. The found spikes were sorted by hand, and the results were compared to the program’s classifications.

![Detected Spikes in Recorded Dataset](image1.png)

Figure 4.1: Left: Overlay of spikes from first 50 buffers (5 seconds) of actual recording. Note the three apparent classes of spikes. Right: The detected classes found by the algorithm.

The classifier was able to correctly classify 86% of the 271 spikes, with an average of 89% accuracy per buffer. “Per buffer” accuracy is an important consideration for BCI applications running on top of a spike sorter. For a given time period (in this case, each 100 ms), the information the application receives will be 89% accurate on average. A lower overall accuracy compared to the per buffer accuracy indicates an inverse correlation between
the number of spikes in the buffer and the sorter’s ability to correctly classify the spikes. The sorter’s positive predictive value (PPV), the ratio of true positives to total number of spikes for the recorded dataset is shown in Table 4.2.

<table>
<thead>
<tr>
<th>Number of Spikes</th>
<th>True Positives</th>
<th>PPV</th>
</tr>
</thead>
<tbody>
<tr>
<td>271</td>
<td>232</td>
<td>86%</td>
</tr>
</tbody>
</table>

4.1.2 Results on the Generated Dataset

The generated dataset was created by reading in three spike waveforms from a file, shown in Figure 4.2, and adding them to a noisy dataset. Because the spikes are added by the program, an actual record of what spike waveform fired when may be compared to the detector and classifier to give a baseline for their accuracies.

![Figure 4.2: Left: Spikes from read-in file to be used to generate simulated neural data. Right: The detected classes found by the algorithm. Notice the shifting so that the first peaks are aligned.]

When processing the generated data, 7 false positives and 14 false negatives were detected
out of the 199 true spikes in the data. The selectivity and sensitivity, reported in Table 4.3, confirms the results from the recorded dataset.

<table>
<thead>
<tr>
<th># Spikes</th>
<th># Detected</th>
<th>Specificity</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>199</td>
<td>185</td>
<td>93%</td>
<td>96%</td>
</tr>
</tbody>
</table>

The algorithm correctly classified 97% of the 185 spikes found by the detector, with an average accuracy per buffer of 92%. The sorter had 90% sensitivity and 94% selectivity. The positive predictive value is shown in Table 4.4.

<table>
<thead>
<tr>
<th>Number of Spikes</th>
<th>True Positives</th>
<th>PPV</th>
</tr>
</thead>
<tbody>
<tr>
<td>185</td>
<td>180</td>
<td>97%</td>
</tr>
</tbody>
</table>

4.1.3 Accuracy Comparisons

In an effort to qualitatively evaluate the accuracies of the classification and detection algorithms presented in this project, comparisons ought to be made to other existing algorithms. Gibson et al. [54] reported the median accuracies of three spike detection methods: absolute value (Abs), nonlinear energy operator (NEO), and stationary wavelet transform product (SWTP). The accuracy was measured in $P_D$, probability of detection, defined in Equation 4.1.

$$P_D = 1 - \frac{FalseNegatives}{TruePositives} \quad (4.1)$$

The accuracy of each was: Abs - 91%, NEO - 93%, SWTP - 77%. For comparison, the detection algorithm presented here had a $P_D$ of 92% on the generated dataset and 95% on the recording.
Gibson’s group [54] also reported on the median accuracies of eight classification methods: Principal Component Analysis (PCA), Discrete Wavelet Transform (DWT), Discrete Derivatives (DD), Integral Transform (IT), Lilliefors’s Test (LT), Hartigan’s Dip Test (HDT), Maximum Difference Test (MDT), and Uniform Sampling (US). The average accuracies are shown in Table 4.5.

Table 4.5: Median Accuracies of Various Sorting Algorithms [54]

<table>
<thead>
<tr>
<th>Method</th>
<th>Median Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCA</td>
<td>80%</td>
</tr>
<tr>
<td>DWT</td>
<td>77%</td>
</tr>
<tr>
<td>DD</td>
<td>80%</td>
</tr>
<tr>
<td>IT</td>
<td>63%</td>
</tr>
<tr>
<td>LT</td>
<td>80%</td>
</tr>
<tr>
<td>HDT</td>
<td>80%</td>
</tr>
<tr>
<td>MDT</td>
<td>87%</td>
</tr>
<tr>
<td>US</td>
<td>76%</td>
</tr>
</tbody>
</table>

The probability of detection for the sorter presented here was 89% on the generated dataset and 83% on the recorded dataset.

A lot of work has been done to improve accuracy of a spike sorters and detectors [17, 46, 47, 48, 50]. Ideally, equivalent work will be done for parallel implementations. It is not the goal of this work to create the best or even most optimized algorithms, but instead aims to build a framework for parallelization of other spike detection and sorting algorithms while presenting a detector and a classifier with reasonable performance and accuracy.

4.2 Processing Time

While accuracy of detection and classification is a necessary condition for the algorithm to be useful, its real benefit lies in the ability to process multiple input channels in real time. More specifically, the processing times of the CPU and GPU versions have been determined to find the real-time processing capacity of both. This section will clearly depict the benefits of GPU computing for massively parallelizable functions like spike sorting. Additionally, because GPU thread creation is negligible and GPGPU technologies will improve with time, the GPU processing speed should continue to scale into future applications. Multithreaded
CPUs, however, face large overheads for thread creation and memory bus bottlenecks, making them much less scalable.

### 4.2.1 CPU Multiprocessor Results

Using the CPU, the total processing time for the filter, detector, and classifier stayed under the 50 ms threshold for real-time operation until the test reached 128 channels. At that point, the median value was 57.9 ms, and the upper and lower tenth percentiles were 75.1 and 45.6 respectively. Any channel count above 128 was unreasonable for real-time feedback. The processing times of each component is shown in Table 4.6.

<table>
<thead>
<tr>
<th># Chs</th>
<th>Filter</th>
<th>Detector</th>
<th>Classifier</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4 [0.3 0.4]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.1 [0.0 0.2]</td>
<td>0.5 [0.4 0.6]</td>
</tr>
<tr>
<td>2</td>
<td>0.7 [0.6 0.7]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.1 [0.0 0.4]</td>
<td>0.9 [0.8 1.2]</td>
</tr>
<tr>
<td>4</td>
<td>1.3 [1.2 1.4]</td>
<td>0.2 [0.2 0.2]</td>
<td>0.2 [0.1 0.7]</td>
<td>1.7 [1.5 2.6]</td>
</tr>
<tr>
<td>8</td>
<td>2.5 [2.4 2.7]</td>
<td>0.4 [0.3 0.4]</td>
<td>0.4 [0.1 1.4]</td>
<td>3.3 [3.0 4.9]</td>
</tr>
<tr>
<td>16</td>
<td>4.9 [4.9 5.0]</td>
<td>0.7 [0.6 0.7]</td>
<td>0.7 [0.3 2.8]</td>
<td>6.5 [5.9 8.6]</td>
</tr>
<tr>
<td>32</td>
<td>11.3 [11.1 13.0]</td>
<td>1.3 [1.3 1.5]</td>
<td>1.4 [0.6 5.5]</td>
<td>14.5 [13.1 20.0]</td>
</tr>
<tr>
<td>64</td>
<td>18.5 [17.5 31.8]</td>
<td>2.7 [2.5 11.9]</td>
<td>2.9 [1.2 10.6]</td>
<td>30.6 [21.8 48.3]</td>
</tr>
<tr>
<td>128</td>
<td>40.0 [35.9 52.7]</td>
<td>5.6 [5.1 11.6]</td>
<td>5.7 [2.5 19.7]</td>
<td>57.9 [45.6 75.1]</td>
</tr>
<tr>
<td>256</td>
<td>83.8 [78.2 98.8]</td>
<td>10.8 [9.8 17.1]</td>
<td>11.1 [5.0 38.7]</td>
<td>11.9 [98.0 143.7]</td>
</tr>
<tr>
<td>512</td>
<td>149.6 [144.7 161.6]</td>
<td>18.9 [18.5 24.4]</td>
<td>20.5 [9.7 75.5]</td>
<td>196.4 [179.1 254.2]</td>
</tr>
<tr>
<td>1024</td>
<td>289.3 [280.8 304.0]</td>
<td>38.5 [35.1 46.3]</td>
<td>40.5 [18.2 150.7]</td>
<td>381.2 [345.5 488.4]</td>
</tr>
</tbody>
</table>

Processing times given in median [0.1 0.9] percentiles.

### 4.2.2 GPU Results

For the GPU, the combined processing time stayed under the 50 ms limit all the way up past 1024 channels. At that point, the median value was 18.3 ms, and the upper and lower tenth percentiles were 24.7 and 17.1 respectively. These times are strikingly similar to the CPU results at 32 channels, demonstrating the GPU’s capability to process up to 32 times the number of channels as traditional many-core processors. The breakdown of processing
### Table 4.7: GPU Processing Times

<table>
<thead>
<tr>
<th># Chs</th>
<th>Filter</th>
<th>Detector</th>
<th>Classifier</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1 [0.1 0.1]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.5]</td>
<td>0.4 [0.3 0.7]</td>
</tr>
<tr>
<td>2</td>
<td>0.2 [0.2 0.2]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.5]</td>
<td>0.4 [0.3 0.8]</td>
</tr>
<tr>
<td>4</td>
<td>0.2 [0.2 0.2]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.5]</td>
<td>0.4 [0.3 0.8]</td>
</tr>
<tr>
<td>8</td>
<td>0.2 [0.2 0.2]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.5]</td>
<td>0.4 [0.4 0.8]</td>
</tr>
<tr>
<td>16</td>
<td>0.3 [0.3 0.3]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.6]</td>
<td>0.5 [0.5 0.9]</td>
</tr>
<tr>
<td>32</td>
<td>0.4 [0.4 0.4]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.6]</td>
<td>0.8 [0.7 1.2]</td>
</tr>
<tr>
<td>64</td>
<td>0.7 [0.7 0.7]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.2 [0.1 0.6]</td>
<td>1.2 [1.1 1.6]</td>
</tr>
<tr>
<td>128</td>
<td>1.6 [0.4 1.7]</td>
<td>0.1 [0.1 0.1]</td>
<td>0.4 [0.2 1.2]</td>
<td>2.5 [2.0 3.8]</td>
</tr>
<tr>
<td>256</td>
<td>3.1 [2.8 18.8]</td>
<td>0.2 [0.2 0.2]</td>
<td>0.6 [0.4 1.8]</td>
<td>4.8 [4.1 20.6]</td>
</tr>
<tr>
<td>512</td>
<td>5.7 [5.5 12.6]</td>
<td>0.4 [0.4 0.4]</td>
<td>1.0 [0.6 3.1]</td>
<td>8.7 [7.9 15.5]</td>
</tr>
<tr>
<td>1024</td>
<td>12.3 [12.1 16.2]</td>
<td>0.7 [0.7 0.7]</td>
<td>1.9 [1.2 5.8]</td>
<td>18.3 [17.1 24.7]</td>
</tr>
</tbody>
</table>

Processing times given in median [0.1 0.9] percentiles.

Times for each component is shown in Table 4.7.

#### 4.2.3 CPU vs GPU Results

The processing times of both GPU and CPU execution are shown in Figure 4.3. Both approaches are plotted on the same axes, with the shaded area depicting the range from the tenth percentile to the ninetieth percentile. The real-time threshold is shown on each graph as a dashed line. Clearly, the filter is what really held back the system in both cases, although the CPU sorter is extremely divergent in its processing times – probably due to the varying number of spikes and classes.

Because the filter is the limiting factor in performance, other, more complex detection and sorting methods could be implemented in parallel without sacrificing performance. In fact, this is one of the most hopeful outcomes of this project: it demonstrates a scalable, viable real-time spike detection and classification program that provides a framework for more advanced parallel algorithms. Additionally, most analog-to-digital converters used for spike recording perform high-pass filtering in hardware, allowing this step to be eliminated entirely.
Figure 4.3: Top Left: Processing times of the CPU and GPU versions. In order to support real-time BCI, the processing time must be under 50 ms [26]. Top Right, Bottom Left, and Bottom Right: Average processing times of the filter, detector, and classifier, respectively. For all graphs, the shaded area shows the range in processing time from the tenth percentile to the ninetieth percentile.

GPU processing clearly provides a significant speedup over the CPU algorithm. The total speedup holds at approximately 22-fold for 32 channels and above. The filter follows suit, having leveled off at about 26× speedup over the same range. Interestingly, the speedups of GPU detection and classification algorithms increase with the number of channels, indicating they will gain even better performance over the sequential versions for even larger-scale recordings. These speedups are shown in Table 4.8.
### Table 4.8: Ratio of CPU:GPU Median Processing Times

<table>
<thead>
<tr>
<th># Chs</th>
<th>Filter</th>
<th>Detector</th>
<th>Classifier</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.6</td>
<td>1.2</td>
<td>0.3</td>
<td>1.3</td>
</tr>
<tr>
<td>2</td>
<td>4.5</td>
<td>1.8</td>
<td>0.6</td>
<td>2.3</td>
</tr>
<tr>
<td>4</td>
<td>8.0</td>
<td>3.3</td>
<td>1.2</td>
<td>4.3</td>
</tr>
<tr>
<td>8</td>
<td>13.0</td>
<td>6.2</td>
<td>2.3</td>
<td>7.5</td>
</tr>
<tr>
<td>16</td>
<td>19.0</td>
<td>10.0</td>
<td>4.4</td>
<td>12.0</td>
</tr>
<tr>
<td>32</td>
<td>26.8</td>
<td>16.0</td>
<td>8.2</td>
<td>18.8</td>
</tr>
<tr>
<td>64</td>
<td>26.5</td>
<td>29.8</td>
<td>14.7</td>
<td>26.4</td>
</tr>
<tr>
<td>128</td>
<td>25.8</td>
<td>40.3</td>
<td>15.1</td>
<td>23.1</td>
</tr>
<tr>
<td>256</td>
<td>27.4</td>
<td>51.4</td>
<td>18.8</td>
<td>23.5</td>
</tr>
<tr>
<td>512</td>
<td>26.2</td>
<td>52.6</td>
<td>20.1</td>
<td>22.7</td>
</tr>
<tr>
<td>1024</td>
<td>23.6</td>
<td>55.7</td>
<td>21.2</td>
<td>20.8</td>
</tr>
</tbody>
</table>

### 4.3 Discussion

As can be seen in Figure 4.3, there is a very significant speedup between the CPU and GPU versions. In addition to maintaining around a $22\times$ speedup as the number of channels increase, the GPU version also has a smaller spread from the tenth to ninetieth percentile. This means the response time of the parallel version is more consistent than that of the sequential version. From the same figure, it is of note that the CPU sorter has the highest deviation in processing times. This is probably due to the variety in the number of spikes to be sorted buffer to buffer. While there is a significant deviation in the GPU sorter (1.0-5.5 ms at 1024 channels), there is actually greater span in the GPU filter (12.5-26.5 ms at 1024 channels). This is especially interesting, because the CPU filter shows very little deviation relative to its speed (289.2-323.8 ms at 1024 channels). It should also be mentioned that the graphs are not to scale; the dotted line on each of them is at 50 ms. Examining the graphs with respect to the real-time threshold reveals that the filter is the slowest for both CPU and GPU, and the threshold-based detector is by far the fastest operation.

The CPU and GPU implementations had different performance optimization goals. On the GPU, the development priorities were minimizing data transfer and optimizing memory access, whereas on the CPU maximizing shared memory between threads and reducing the thread creation overhead were the primary goals. As stated in Section 3.3, the CPU used for
these tests is capable of running up to 24 threads in parallel, but if more than one requests the same location in memory, the application encounters bottlenecks and becomes essentially sequential, but with the added overhead from creating the threads. This may account for some of the poor performance of the CPU, even with dual 12-core processors.

Table 4.1 and Table 4.3 show the selectivity and sensitivity of the detector on each dataset. The consistency in performance between the two datasets is striking; the detector was 97% selective on the recorded data and 96% selective on the generated data. Similarly, the sensitivity of on the recorded data and generated data was 94% and 93%, respectively. When comparing to Gibson’s results [54], who used probability of detection as the measure of accuracy, the detector presented here showed excellent performance. The three algorithms evaluated by Gibson et al. (absolute value, nonlinear energy operator, and stationary wavelet transform product) had median $P_D$ of 91, 93, and 77 percent, while the detector used here performed at 92% for the generated data and 95% on the recorded data.

The positive predictive value of the sorter on each dataset was shown in Tables 4.2 and 4.4. The median probabilities of detection for the classifiers reported by Gibson et al. ranged between 63% and 87%. The $P_D$ for the sorter presented here was 89% on the generated dataset and 83% on the recorded dataset.

These results show that the strategies presented in this paper are viable options for spike detection and classification. The accuracies found in tests place both detector and classifier well within the range of existing methods.

The capacity to scale was another important factor in this project’s design. While current BCI research is done using up to a few hundred channels, future work may require several thousand. Although this project showed the ability to handle 1024 channels in well under the real-time threshold, it is reasonable to think that future needs could expand beyond 1024 electrodes. Because this system was written for parallel processing on a GPU, it is inherently scalable. As GPUs improve and are focused more on general purpose computing, the capabilities of parallel programs will improve with them.
Chapter 5

Conclusions

5.1 Project Overview

The future of brain-computer interface technologies is facing the question of how to best handle the increasing amounts of available neural information. In order to be widely accepted and available, BCI applications need to be able to give real-time feedback for more data, and they need to be inexpensive and mobile.

Real-time feedback is necessary for a closed-loop system so that the brain can learn and link its actions to the output of the BCI system [16]. Because the quality of signal from the brain is poor, the amount of information gathered is important. More information allows for better control and a wider control domain, so increasing the number of channels that can be processed in time has been a goal in recent BCI technologies research ([23, 55, 56]).

Expense is a deterrent for research labs, hospitals, and small companies and individuals in adopting BCI technologies. With high-end dedicated systems costing tens of thousands of dollars, the number of entities that can even take the first step into BCI is extremely limited. Mobility is also an important consideration. The current and future availability of mobile computing suggests an expectation that all technologies should have the ability to be mobile. Dedicated systems and high-end desktop computers are not able to make the
transition to mobile applications. However, the human brain is extremely mobile and as BCI technologies step out of the “proof of concept” and development stages, they should be mobile as well. Therefore, the goals of the research presented here are to 1) develop methods for spike processing of multi-channel microelectrode data, 2) implement the methods as a highly parallel algorithm on a graphics processing unit (GPU)-based system to allow real-time processing, and 3) create a viable, scalable framework for future parallel spike detectors and sorters.

In this work we developed a parallelized, adaptive sorting algorithm based on template matching. A thresholding detector using dynamic running-average standard deviation-based thresholds was written specifically to help maximize the sorter’s accuracy. A filter was added to deal with real neural recordings. All three of these were written in both sequential and parallel versions for traditional multicore CPU execution and massively parallel execution on a CUDA-enabled NVIDIA GPU, respectively.

For testing, a generated dataset was used to model neural recordings for accuracy tests, and an actual LFP recording was used to evaluate real-world performance, which was compared against a human detector and classifier on the filtered data. Timers in the code were used to record median, tenth and ninetieth percentile processing times for each of the three components of the spike sorting program.

This thesis has presented a scalable, parallel spike sorting program for use with BCI applications. In accuracy tests, both the detection and classification algorithms performed reasonably. Speed tests reveal the capability for thousands of channels to be processed under the 50 ms real-time threshold. Packaging this program open source with BCI2000 will provide immediate capability for BCI research labs worldwide to process any practical number of channels for very little, if any expense. It will also provide the framework to parallelize more accurate detectors and sorters, with a lot of room for more computational intensity. This will lead to the advancement of BCI technologies and their acceptance into a wider audience.
5.2 Future Work

The code developed in this research will be finalized and submitted to BCI2000 to be packaged with their software, and thus be made available in open source form. Within the code, detection strategies should be made available for both GPU and CPU. Handling spike overlap is a significant issue that remains to be addressed in this regard. Also, significant work was done in getting a quantile-based dynamic threshold working, but it is still in development.

It is the intention of this project to provide a viable option for massively parallel spike sorting as well as a framework for the development of other parallel algorithms. There are a lot of strategies for improving accuracy of both detectors and classifiers, or making better use of the information collected (such as improving spatial resolution). Parallelization of these algorithms to get the absolute most out of available neural data is one way to help move the field of BCI forward. The capacity to process thousands of channels in real time makes this especially important. Tasks such as this, which can be handled separately from the detection and classification algorithms, can be run on a different GPU.

Overall, this project represents a “proof of concept”, and can yield even greater efficiency and accuracy with the addition of more sophisticated signal processing, adaptation and classification methods.
Bibliography


Appendix A

Source Code

The following are highlights from the code describing the detector and sorter implementations on the CPU and GPU.

A.1 Main Function

Below is a very stripped-down version of the main function.

```c
int main(int argc, char **argv)
{
    // Set up Filter Class
    FilterGroup mFilter(nChs, blockSize, filterSize, fs, 300, 0, true);

    // Set up Detector Class
    ThresholdGroup threshold(nChs, blockSize, spikeLength);
    if(useCUDA) threshold.SetNumCUDAthreads(nCUDAthreads);
    threshold.SetMaxNumSpikes(maxNumSpikes);

    // Set up Classifier Class
    CorrClassifier CorrClass(nChs,
                            maxComponents,
                            spikeLength,
                            CorrThresh,
                            maxNumSpikes,
                            inactiveLimit);
    if(useCUDA) CorrClass.SetNumCUDAthreads(nCUDAthreads);
    CorrClass.setSpikeLength(spikeLength);

    // Begin Data Processing Loop
    for (int b = 0; b < dataLength−blockSize && blockNum < 200; b+=blockSize, blockNum++){
        // Filter Raw Data
        mFilter.Filter(rawData, fRes);

        // Calculate Threshold
        threshold.CalculateThreshWithSTD(fRes, 4);

        // Repeat for running average threshold
        if (blockNum < 8) continue;
    }
}
```
A.2 Reduction

A lot of attention was paid to parallel reduction throughout this paper. Implementing standard deviation in parallel is fairly trivial, except for the inclusion of reduction. Below is the source code used in this project for parallel reduction.

```cpp
inline void Reduce(float *sumArray, int nThreads, int tid)
{
    if (nThreads >= 512)
    {
        if (tid < 256)
        {
            sumArray[tid] += sumArray[tid + 256];
        }
        __syncthreads();
    }
    if (nThreads >= 256)
    {
        if (tid < 128)
        {
            sumArray[tid] += sumArray[tid + 128];
        }
        __syncthreads();
    }
    if (nThreads >= 128)
    {
        if (tid < 64)
        {
            sumArray[tid] += sumArray[tid + 64];
        }
        __syncthreads();
    }
    if (tid < 32)
    {
        volatile float *smem = sumArray;
        if (nThreads >= 64)
        {
            smem[tid] += smem[tid + 32];
        }
        if (nThreads >= 32)
        {
            smem[tid] += smem[tid + 16];
        }
        if (nThreads >= 16)
        {
            smem[tid] += smem[tid + 8];
        }
        if (nThreads >= 8)
        {
            smem[tid] += smem[tid + 4];
        }
        if (nThreads >= 4)
        {
            smem[tid] += smem[tid + 2];
        }
    }
}
```

else{
    //Detect Spikes
    threshold.Process(fRes, timeStamps, numSpikes);

    //Classify Found Spikes
    CorrClass.Classify(fRes, timeStamps, numSpikes, spikeClass);
}

```
A.3 Detector

This section includes the source code for the core of the detection algorithms for the CPU and GPU. The similarities between the two can be clearly seen, and the subtle differences reflect the additional considerations for GPU programming.

A.3.1 CPU Detector

The following source code describes the detection algorithm on the CPU. The variables \texttt{mPre} and \texttt{mPost} are the number of samples on either side of the peak, \texttt{inChData} is the filtered data, and \texttt{peaks} records the timestamps.

```c
// Iterate through the filtered data
for (unsigned int s = mPre; s < inData.numSamples-mPost; s++){
    // Check for local max
    if (inChData[s] >= thrHigh &&
        inChData[s] > inChData[s-1] &&
        inChData[s] > inChData[s+1]){  
        int i;
        // Test if the local max is a peak
        for (i=2;i<span;i++){
            // Test for surrounding peaks
            if (inData.data[s-i]>inData.data[s])
                break;
            if (inData.data[s+i]>inData.data[s])
                break;
            if (inData.data[s-i]<thrLow)
                break;
        }
        // If it passes the tests, record the timestamp
        if (i==span){
            peaks.push_back(s);
        }
    }
    // Check for local min
    else if (inChData[s] <= thrLow &&
             inChData[s] < inChData[s-1] &&
             inChData[s] < inChData[s+1]){  
        int i;
        // Test if the local min is a peak
```
for (i=2;i<\text{span};i++){
    \textbf{if}(\text{inData}.\text{data}[s-i]<\text{inData}.\text{data}[s])
    \textbf{break};
    \textbf{if}(\text{inData}.\text{data}[s+i]<\text{inData}.\text{data}[s])
    \textbf{break};
    \textbf{if}(\text{inData}.\text{data}[s-i]>\text{thrHigh})
    \textbf{break};
}

// If it passes the tests, record the timestamp
if(i==\text{span}){
    \text{peaks}.\text{push}\_\text{back}(s);
}
}

\subsection*{A.3.2 GPU Detector}

The source code below shows the thresholding strategy on the GPU. The variable \text{tid} is the thread ID, \text{pre} and \text{post} are the number of samples in the waveform before and after the peak, and \text{buffer} is a shared memory array.

\begin{verbatim}
pos=\text{tid}+\text{pre};

// iterate through the filtered data
\textbf{while}(\text{pos} < \text{nSamples}−\text{post}){

    // Check for threshold crossing
    \textbf{if} (\text{buffer}[\text{pos}] > \text{upperthreshold}){
        // Check for local max
        \textbf{if} ( (\text{buffer}[\text{pos}] > \text{buffer}[\text{pos}−1]) \&\&
                     (\text{buffer}[\text{pos}] > \text{buffer}[\text{pos}+1])){
            \text{int} \ i;

            // test if the local max is a peak
            \textbf{for} (i=2;i<\text{span};i++){
                \textbf{if}(\text{buffer}[\text{pos}−i]>\text{buffer}[\text{pos}])
                \textbf{break};
                \textbf{if}(\text{buffer}[\text{pos}+i]>\text{buffer}[\text{pos}])
                \textbf{break};
                \textbf{if}(\text{buffer}[\text{pos}−i]<\text{lowerthreshold})
                \textbf{break};
            }

            // If it passes the tests, record the timestamp
            if(i==\text{span}){
                \text{ts}[\text{nSpikes}++]=\text{pos};
            }
        }
    }

    // Check for threshold crossing
    \textbf{else if} (\text{buffer}[\text{pos}]<\text{lowerthreshold}){

        // Check for local min
        \textbf{if} ( (\text{buffer}[\text{pos}]<\text{buffer}[\text{pos}−1]) \&\&
                     (\text{buffer}[\text{pos}]<\text{buffer}[\text{pos}+1])){
            \text{int} \ i;

            // test if the local min is a peak

    }
\end{verbatim}

58
for (i=2;i<\text{\texttt{span}};i++){
    \text{\texttt{if}}( \text{\texttt{buffer}}[\text{\texttt{pos}}-\text{\texttt{i}}]<\text{\texttt{buffer}}[\text{\texttt{pos}}])
        \text{\texttt{break}};
    \text{\texttt{if}}( \text{\texttt{buffer}}[\text{\texttt{pos}}+\text{\texttt{i}}]<\text{\texttt{buffer}}[\text{\texttt{pos}}])
        \text{\texttt{break}};
    \text{\texttt{if}}( \text{\texttt{buffer}}[\text{\texttt{pos}}-\text{\texttt{i}}]>\text{\texttt{upperthreshold}})
        \text{\texttt{break}};
}

\text{\texttt{// If it passes the tests, record the timestamp}}
\text{\texttt{if}}(i==\text{\texttt{span}}){
    \text{\texttt{ts}}[\text{\texttt{nSpikes}}++]=\text{\texttt{pos}};
}

\text{\texttt{// Iterate by the number of threads}}
\text{\texttt{pos+=}}\text{\texttt{nThreads}};
\_\_\_\_\text{\texttt{syncthreads}}();

\textbf{A.4 Sorter}

The classification algorithm employs several important functions. Fortunately, the implementations on the CPU and the GPU are reflective of one another, so the differences between them are clear.

\textbf{A.4.1 CPU Sorter}

The first function shown controls the CPU sorter and calls the latter functions. It is important to remember that this function only processes one channel at a time.

\begin{verbatim}
void \texttt{CorrClassifier::ClassifyChannel(DataVectorArray& data, DataVectorArray& timestamps, DataVectorArray& nSpikes, int ch, DataVectorArray& spikeClassify)}
{
    float N = nSpikes[ch]->data[0];
    const DataVector *dataCh = data[ch];
    const DataVector *tsCh = timestamps[ch];
    float *curSpike;

    // Iterate through the spikes, classifying each one sequentially
    for (int curSpikeNum = 0; curSpikeNum < N; curSpikeNum++){
        float maxR;
        int maxRidx;
        int ts = (int)tsCh->data[curSpikeNum];

        //Copy the current spike’s waveform to an array
        curSpike = &dataCh->data[ts - mSettings.prePeakLength];

        //Find the maximum correlation with the existing templates
        CalculateMaxR(curSpike, ch, maxR, maxRidx);
        spikeClassify[ch]->data[curSpikeNum] = -1;
    }
}
\end{verbatim}
Next, the `CalculateMaxR` function calculates the correlation coefficient using Equation 3.2.

```c
void CorrClassifier::CalculateMaxR(float *curSpike, int ch, float &maxR, int &maxRidx) {
    float spikeMean, spikeStd;
    // Calculate the standard deviation of the spike
    spikeStd = StdDev(curSpike, mSettings.spikeLength, spikeMean);
    float *curTemplate;
    float numTemplates = mNumTemplates[ch]->data[0];
    maxR = 0;
    maxRidx = -1;
    // Find R for each existing template
    for (int i = 0; i < numTemplates; i++){
        float templateMean, templateStd;
        curTemplate = &mTemplates[ch]->data[i*mSettings.spikeLength];
        // Calculate the standard deviation of the template
        templateStd = StdDev(curTemplate, mSettings.spikeLength, templateMean);
        float R = 0.0f;
        // Calculate the correlation
        for (int s = 0; s < mSettings.spikeLength; s++){
            R += ((curSpike[s]-spikeMean)/spikeStd)*((curTemplate[s]-templateMean)/templateStd);}
        R /= (float)(mSettings.spikeLength-1);
        // Keep track of the maximum correlation
        if (R > maxR){
            maxR = R;
            maxRidx = i;
        }
    }
}
```

Lastly, once the class has been determined, the template is updated to keep an average
waveform of every spike in the class.

```cpp
void CorrClassifier::UpdateTemplate(float *curSpike, int ch, int maxRidx)
{
  // Copy template to an array
  float *curTemplate = &mTemplates[ch]->data[maxRidx*mSettings.spikeLength];
  // Find the number of spikes in the class
  float N = mSpikesPerTemplate[ch]->data[maxRidx];
  // Add the spike to the template waveform point-by-point, and find the new average
  for (int i = 0; i < mSettings.spikeLength; i++){
    curTemplate[i] = (curTemplate[i]*N + curSpike[i])/(N+1.0f);
  }
  // Increase the number of spikes in the class
  mSpikesPerTemplate[ch]->data[maxRidx]++;
}
```

### A.4.2 GPU Sorter

Lastly, the code for parallel sorting will be shown. As stated previously, the GPU sorter is very similar looking to the CPU sorter, but the underlying operations are very different. Note that, while the spikes and templates are processed sequentially, there is a lot of parallelism in the thread operations, and each channel is being processed in parallel by different blocks of threads. As was done with the CPU code, the kernel will be shown first, followed by its functions.

```cpp
__global__ void CorrelationSort_kernel(DataVector *signal,
DataVector *timestamps,
DataVector *numSpikes,
DataVector *numTemplates,
DataVector *spikesPerTemplate,
DataVector *spikeClassify,
DataVector *corrArray,
DataVector *corrThresholds,
DataVector *templates,
CorrClassifierSettings settings)
{
  // Thread-specific variables
  int tid = threadIdx.x;
  int nThreads = blockDim.x;
  int ch = blockIdx.x;

  // Declare shared variables
  __shared__ int nSpikes;
  __shared__ float maxCorr;
  __shared__ int maxCorrIdx;
  __shared__ TemplateArray templateArray;

  float rthreshold = corrThresholds[ch].data[0];
  // Initialize shared memory pointers
```
float *sTemplates = (float*)sharedMemory;
float *curSpike = (float*)&sTemplates[settings.spikeLength*settings.maxNumTemplates];
float *rArray = (float*)&curSpike[settings.spikeLength];
float *scratch = (float*)&curSpike[settings.maxNumTemplates];
int *sSpikesPerTemplate = (int*)&scratch[nThreads];

// Small portion of sequential operation – defining shared variables
if (tid == 0)
    nSpikes = (int)numSpikes[ch].data[0];
    templateArray.numTemplates = (int)numTemplates[ch].data[0];

// Copy global data to shared memory
int pos=tid;
while(pos < settings.maxNumTemplates)
{
    sSpikesPerTemplate[pos]= (int)spikesPerTemplate[ch].data[pos];
    pos+=nThreads;
}

// Give initial values to arrays
InitializeFloatArrayWithValue(spikeClassify[ch].data, -1.0f, 
spikeClassify[ch].numSamples);
for (pos=tid; pos<settings.maxNumTemplates*settings.spikeLength; pos+=nThreads)
    sTemplates[pos] = 0.0f;
CopyFloatArray(sTemplates, 
    templates[ch].data,
    settings.maxNumTemplates*settings.spikeLength);
for (pos =tid; pos < settings.maxNumTemplates; pos += nThreads)
    templateArray.templatePtr[pos] = sTemplates + settings.spikeLength*pos;
    __syncthreads();

// Loop through all of the spikes and calculate the correlations
for(int spikeNum = 0; spikeNum < nSpikes; spikeNum++)
{
    // Give initial values to arrays for each new spike
    InitializeFloatArrayWithValue(rArray, 
        CUDART_NAN_F, 
        settings.maxNumTemplates);
    __syncthreads();
    int ts = timestamps[ch].data[spikeNum];
    CopyFloatArray(curSpike, 
            &signal[ch].data[ts-settings.prePeakLength],
            settings.spikeLength);

    // Calculate the correlation of the spike with each template
    // Store the result in corrArray
    CalculateCorrelation(curSpike, 
        &settings, 
        &templateArray, 
        scratch, 
        rArray, 
        &maxCorr, 
        &maxCorrIdx);
    __syncthreads();

    // Check for an acceptable correlation
    if (fabsf(maxCorr) >= rthreshold)
    {
        if (tid == 0)
            spikeClassify[ch].data[spikeNum] = maxCorrIdx;
    }
    __syncthreads();

    // Update the chosen class
    if (settings.updateTemplates)
        UpdateTemplate(curSpike, 
...
Above, the \texttt{CalculateCorrelation} function is called to find the correlations of the spike and each template. The GPU version uses the same calculation (Equation 3.2) as the CPU version. Within \texttt{CalculateCorrelation}, the \texttt{Mean} and \texttt{StdDev} functions use parallel
reduction (as does the `Reduce` function), but these will not be highlighted here.

```cpp
__device__ void CalculateCorrelation(float *curSpike,
         CorrClassifierSettings *settings,
         TemplateArray *templateArray,
         float *scratch,
         float *rArray,
         float *maxCorr,
         int *maxCorrIdx)
{
    int tid = threadIdx.x;
    int nThreads = blockDim.x;

    __shared__ float spikeAvg;
    __shared__ float spikeStd;
    __shared__ float templateAvg;
    __shared__ float templateStd;

    // Calculate the mean of the current spike
    Mean(curSpike, settings->spikeLength, scratch);

    // Small bit of sequential processing
    if (tid == 0){
        spikeAvg = scratch[0];
        *maxCorr = 0;
        *maxCorrIdx = -1;
    }
    __syncthreads();

    // Calculate the stddev of the current spike
    StdDev(curSpike, settings->spikeLength, spikeAvg, scratch);
    if (tid == 0)
        spikeStd = scratch[0];
    __syncthreads();

    // Iterate through one template at a time
    for (int t = 0; t < templateArray->numTemplates; t++){
        // Calculate the mean of the current template
        Mean(templateArray->templatePtr[t], settings->spikeLength, scratch);
        if (tid == 0)
            templateAvg = scratch[0];
        __syncthreads();

        // Calculate the stddev of the current template
        StdDev(templateArray->templatePtr[t], settings->spikeLength, templateAvg, scratch);
        if (tid == 0)
            templateStd = scratch[0];
        __syncthreads();

        // Use reduction to add the values in shared memory
        float mySum = 0.0f;
        for (int pos = tid; pos < settings->spikeLength; pos += nThreads)
            mySum+=((curSpike[pos]-spikeAvg)/spikeStd)*
            ((templateArray->templatePtr[t][pos]-templateAvg)/templateStd);
        scratch[tid] = mySum;
        Reduce(scratch, nThreads, tid);
    }

    // Record the correlation
    if (tid == 0){
        rArray[t] = scratch[0]/((float)settings->spikeLength-1.0f);
        if (rArray[t] > (*maxCorr)){
            *maxCorr = rArray[t];
            *maxCorrIdx = t;
        }
    }
}
```
The last function to be shown in this appendix is the `UpdateTemplate` function for the GPU. Notice that this function and `CalculateCorrelation` are labelled as device functions, whereas `CorrelationSort_kernel` is a global function. This is just a way for CUDA to tag from where functions may be called. Global functions may be called by the host or the device, while device functions can only be called from the device. Below, the `UpdateTemplate` function operates in a very similar fashion to its CPU counterpart.

```c
__device__ void UpdateTemplate(float *curSpike, CorrClassifierSettings *settings, int maxCorrIdx, int *sSpikesPerTemplate, TemplateArray *templateArray)
{
    int tid = threadIdx.x;
    int nThreads = blockDim.x;

    // Add the spike to the template waveform point-by-point, and find the new average
    for (int pos = tid; pos < settings->spikeLength; pos += nThreads){
        templateArray->templatePtr[maxCorrIdx][pos] =
            (templateArray->templatePtr[maxCorrIdx][pos]*sSpikesPerTemplate[maxCorrIdx] +
            curSpike[pos])/(sSpikesPerTemplate[maxCorrIdx]+1.0);
    }
    __syncthreads();

    // Increase the number of spikes in the class
    if (tid == 0)
        sSpikesPerTemplate[maxCorrIdx]++;
}
```