A Real-time Spike Sorting System using Parallel OSort Clustering

Daniel Valencia and Amir Alimohammad
Department of Electrical and Computer Engineering
San Diego State University, San Diego, U.S.A.

***Abstract*—This article presents an efficient design and implementation of a real-time spike sorting system using unsupervised clustering. We utilize the online sorting (OSort) algorithm and model it first in both floating-point and fixed-point numerical representations to accurately assess the feasibility of our hardware architecture and also reliably analyze the sorting accuracy. For efficient hardware realization of OSort, we propose a modified parallel OSort algorithm. By reducing the number of required memory accesses, the number of computations performed for the management and upkeep of cluster averages and cluster merging is substantially reduced. By limiting the number of supported clusters per channel, the classification/clustering latency is significantly reduced compared to the previously published work, making the designed OSort system applicable for in-vivo spike sorting. The proposed OSort hardware architecture utilizes a novel memory configuration scheme to parallelize the OSort algorithm, which allows us to avoid using relatively large memory queues for storing detected spike waveforms and process them concurrently to the spike cluster management. The characteristics and implementation results of the designed OSort-based spike sorting system on a Xilinx Artix-7 field-programmable gate array (FPGA) are presented. The ASIC implementation of the designed system is estimated to occupy 2.57 mm^2 in a standard 32-nm CMOS process. Post-layout power estimation shows that the ASIC will dissipate 2.78 mW, while operating at 24 kHz.**

I. Introduction

Single-unit recording allows neuro-scientists to study how the neural activity of neurons is correlated with one another. Due to the electrical activity of nearby neurons and ambient noise, these accumulated electrical activities will be detected by the recording electrodes. In principle, each neuron tends to produce spikes, also known as action potentials (APs), of a particular shape. The process of classifying the neural activity of different neurons is referred to as spike sorting [1]. Spike sorting consists of four different processing steps: (i) spike detection, (ii) spike alignment, (iii) feature extraction, and (iv) clustering [2]. The neural signal from the sensing electrode is first filtered (300 Hz – 3 kHz). Neural spike waveforms are then detected from the accumulated neural activities of neighboring neurons and noise. Some of the most commonly employed detection methods are the absolute value thresholding [3], the non-linear energy operator (NEO) [4], or the discrete wavelet transform (DWT) algorithm [5]. These detected spikes are then aligned such that a prominent spike feature will be at the same sample number to simplify the feature extraction or clustering processes. The feature extraction process reduces the dimensionality of each neural waveform from P_1 to P_2, where P_1 denotes the number of samples used to represent the spike waveform and P_2 denotes the number of features extracted. When more discriminative features are found, the spike waveforms can be distinguished more easily. Some commonly employed feature extraction techniques include principal components analysis (PCA) [6] and the DWT [7]. Finally, clustering is the process in which spikes are sorted into clusters of distinct spikes.

Conventionally, analog signals from neurons surrounding electrodes were digitized and transmitted to an external computing unit for subsequent off-line software processing. Possible neural recording configurations include a physical connection from electrodes to a workstation or wireless transmission of neural recordings. While offline processing of recorded neural data allows researchers to utilize computationally-intensive, but relatively accurate signal processing algorithms, real-time processing is of vital importance for in-vivo operation of brain-implantable devices. To minimize the processing latency, it is crucial to avoid computationally daunting and power hungry algorithms, such as PCA or DWT, which also incur relatively large computational delay. Additionally, algorithms that circumvent the feature extraction and clustering processes, such as template-matching [8], may not be applicable as they require prior spike waveform information, which may or may not be available. Compressed sensing (CS) has been recently used to reduce the wireless data rate requirement for transmitting sampled neural signals [9] [10]. For example, the compression rates achieved by the work in [9] ranges from 8 to 16 times, while the work in [10] achieves a compression rate of 10 times. Although compressed sensing presents a viable bandwidth reduction strategy, spike sorting offers a more considerable data rate reduction, as only relevant spiking activity is transmitted [11]. Assuming a 24 kHz sampling rate with 16 bits per sample, each neural recording site generates a raw data rate of 384 kbps. This data rate becomes prohibitively large with modern recording configurations, where hundreds of electrodes record neural signals. Assuming the typical spiking rate of a neuron is 40 spikes per second [1], and utilizing only a few bits (between 3 and 5) for representing the type of spike detected, the output data-rate would range between 120 and 150 bps. Moreover, CS algorithms require signal reconstruction to be performed after compressed data has been received, which may incur additional processing latency. The main goal of real-time spike sorting implementations is to reduce the output sorting data rate while the latency between spike detection and spike classification is minimized. This is because inducing synaptic modification of neurons requires applying stimulation within critical timing windows. For example, stimulation within 20 ms before or after pre-synaptic activation produces different kinds of synaptic modification [12]. Thus, the classification/sorting latency is of vital concern

in closed-loop, BCI applications. Moreover, a reduction in resource utilization is desired toward the development of low-power and area-constrained implantable devices. In this article, we present modifications to the OSort algorithm to ensure that the classification latency is reduced, which is crucial for real-time spike sorting and closed-loop experiments. We also present an efficient hardware architecture for the real-time realization of OSort-based spike sorting. For efficient hardware implementation of a real-time spike sorting system, we first developed our proposed spike sorting system in both floating-point and fixed-point numerical representations in MATLAB using our custom-developed library of numerical operations in MEX/C to carefully assess the feasibility of our hardware architecture and also reliably analyze the sorting accuracy. Then, we modify the original OSort algorithm to make it more computationally efficient for real-time operation on detected spikes. Then, the proposed spike sorting system is developed using Verilog hardware description language (HDL). The publicly available WaveClus [7] data-sets are used to verify and quantify the sorting accuracy of the designed spike sorting system.

The rest of this article is organized as follows. Section II briefly reviews the conventional steps of spike sorting. Various algorithms are discussed for performing different steps of the spike sorting chain. Section III presents the OSort algorithm and details our modifications to the algorithm for real-time operation. Section IV discusses the hardware architecture for efficient realization of the detection, alignment, and the original and proposed parallel OSort algorithms. Section V presents the system configuration and simulation results of the designed parallel OSort algorithm performed by the implemented spike sorting system. The characteristics and implementation results of the parallel OSort clustering system are presented in Section VI. Our designed OSort-based spike sorting system is compared to several state-of-the-art spike sorting realizations on both field-programmable gate arrays (FPGAs) and ASIC. Finally, Section VII makes some concluding remarks.

II. Spike Sorting System

Fig. 1 shows the conventional process of spike sorting, consisting of spike detection, spike alignment, feature extraction, and clustering. The output of the spike sorting system is a spike train indicating to which cluster each spike belongs to. The spike train also gives timing information as to which and when neuron's fire.

A. Spike Detection

Spike detection is used to identify the presence of a neural spike, also known as an action potential, from ambient noise. Detection is performed in two steps: (i) Pre-emphasis of the neural signal and (ii) thresholding of the pre-emphasized signal. Pre-emphasis is defined as an operation performed on the neural signal to prepare it for the thresholding step. Various pre-emphasizing methods have been utilized for spike detection, such as employing absolute value [3], NEO [4], and DWT [5]. These methods compute the absolute value of the input signal, the energy of the signal, or the wavelet coefficients of the signal, respectively. In the second step of detection, if the pre-emphasized signal crosses the threshold, this is an indication that a spike has been detected. Because the thresholding step utilized by the absolute value method is not adaptive, it is not chosen for our spike detection scheme. While both NEO and DWT-based detection algorithms utilize adaptive thresholding via the probability of detection and probability of false alarm, the DWT is significantly more computationally-intensive. Therefore, we implement a NEO-based spike detection scheme. The energy operator of the instantaneous value of the input at time t, $x(t)$, can be written as:

$$\psi\big(x(t)\big) = \left(\frac{\mathrm{d}x(t)}{\mathrm{d}t}\right)^2 - x(t)\left(\frac{\mathrm{d}^2x(t)}{\mathrm{d}t^2}\right), \tag{1}$$

It can be shown that the output of NEO is proportional to the product of the amplitude and frequency of the input signal. For a discrete-time sequence $x[n]$, NEO can be written as:

$$\psi\big(x[n]\big) = x^2[n] - x[n+1]x[n-1], \tag{2}$$

where $x[n]$ is a sample of the waveform at discrete time n. The result is large only when the signal is both large in power (i.e., $x^2[n]$) and in frequency (i.e., $x[n]$ is large while both $x[n+1]$ and $x[n-1]$ are small). Since a spike, by definition, is characterized by localized high frequencies and an increase in instantaneous energy, this method is preferred over methods that only consider an increase in signal energy or amplitude.

For thresholding, the threshold can be automatically set to a scaled version of the mean of the recorded signal as:

$$Thr = \frac{C}{N}\sum_{n=1}^{N}\psi\big(x[n]\big), \tag{3}$$

where N denotes the number of samples in the signal and the scaling factor C can be chosen initially by experiment as a constant (e.g., 8). The value of C will then be increased if $P_{FA} < 0.3$ and $P_D > 0.7$ where P_{FA} and P_D denote the probability of false alarm, and probability of detection, respectively. Our simulation results verify that using the NEO algorithm, at least 98% of simulated neural spikes, developed as part of Wave_Clus software [7], which is a well-known and widely available unsupervised spike sorting program developed at the University of Leicester, can be detected.

B. Spike Alignment

Because all spikes are compared by some common features after the feature extraction step, it is important that the feature extraction process produces similar features for similar spike waveforms. To aid the feature extraction process, spike alignment is performed, such that detected spikes are aligned to a common metric. Some of the commonly employed alignment metrics are maximum amplitude, maximum absolute amplitude, and maximum slope [2]. In each of these alignment schemes, the detected spikes are aligned such that the spike waveforms have the maximum amplitude, the maximum absolute amplitude (positive or negative), and the maximum slope are all at the same aligned spike waveform sample number,

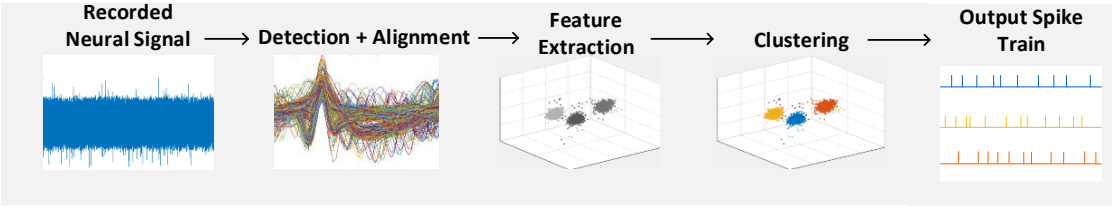

Fig. 1: The conventional off-line spike sorting process.

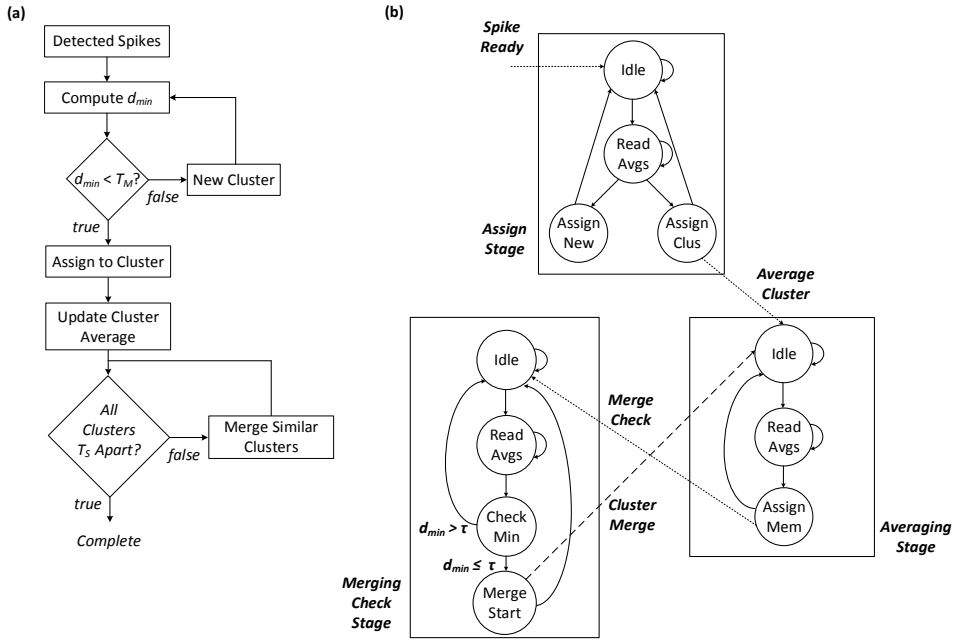

Fig. 2: (a) Flowchart of the original OSort algorithm; and (b) the finite state machine for the real-time realization of OSort.

respectively. Depending on the pre-emphasis method utilized, other alignment techniques can be employed as well, such as alignment to maximum energy for use with NEO-based detectors.

C. Feature Extraction

Once spikes are detected and aligned, feature extraction is applied to extract a set of spike features that yield the best discrimination between spikes of different shapes. In general, the more discriminative features are used, the better the ability to distinguish spike waveforms. For example, if λ features are extracted and stored for each detected spike, the spike waveforms can therefore be represented in a λ-dimensional space. Fig. 1 shows the three features in a 3-dimensional space. The key challenge is to select the minimal set of features that yields the best discrimination.

One of the most commonly used feature extraction and dimensionality reduction methods is PCA [6]. PCA gives an ordered set of orthogonal vectors that captures the directions of the largest variations in the data. A waveform is represented as a linear combination of its principal components. The dimensionality reduction is achieved by going from an M-dimensional space to a K-dimensional space, with $K \ll M$ (typically, K equals to 2 or 3). Using only the first K components can account for most of the energy in the data and their scores can be used as an input to the clustering algorithm. During the off-line parameter estimation, PCA is performed on all detected spikes to create a subspace of maximum variance, which can be used to extract common features among spikes. The DWT has also found an application in feature extraction [7]. The output coefficients of the DWT, which describe both the time and frequency domains, can be used as features for clustering. In addition to PCA and DWT, the discrete derivatives (DD) technique has also been used for feature extraction [13]. It computes the slope at each sample point of the spike and can be considered as a simplified DWT. The integral transform (IT) [14] has also been used for feature extraction, where the discriminating features are the areas under the positive and negative portions of the AP. The appeal of the IT is its inherent reduction to only two features per spike.

D. Clustering

Clustering is used to ultimately classify neural spikes into different groups and recreate the spike train. Traditional spike sorting schemes would often involve forming the clusters by hand on pre-recorded data [2]. On-line techniques have the ability to classify spikes in real-time [15], however, prior

knowledge of the spike waveforms is required. Sorting based on the predetermined shapes of spikes may not be sufficiently accurate since spikes that do not appear often during the learning phase, would either be misclassified or simply not be classified during the online sorting process. Rutishauser et al. [16] developed the OSort algorithm to overcome the barrier of sparsely firing neurons during the learning phase of traditional spike sorting methods. For the template-matching approach, the generation of templates is also a research topic of interest. In [17], the separability of clusters is enhanced by making relatively small changes to the mean template waveforms, thus improving the accuracy of sorting by imposing additional pre-processing without affecting the computational complexity of the algorithm.

While feature extraction based on PCA, DWT, and DD require dimensionality reduction to facilitate the clustering process, OSort can perform clustering without the need for feature extraction. In OSort, the spike waveform itself can be considered as the feature of interest. That is, the incoming spike is stored in its entirety and can be considered as being represented in an R-dimensional space, where R denotes the number of samples in a spike waveform. The actual number of samples depends on the sampling frequency. Employing the commonly used 24-kHz sampling frequency, the WaveClus waveforms consist of 64-samples, which correlates to 2.7 ms of time. Because OSort stores each spike waveform in its entirety, spikes are directly compared to one another with the Euclidean distance metric. Because the Euclidean distance measures the linear distance between the spike waveform samples, it must be ensured that each waveform is aligned to a common feature. For simple implementation, spikes are aligned to the point of maximum amplitude such that all detected spikes share the same alignment point. Since the alignment point contains the maximum amplitude, the waveforms are distinguished by their shapes. If alignment is not performed, a lateral shift of either waveforms would cause the distance metric to return a significantly greater and thus incorrect value.

III. The Modified Parallel OSort Algorithm

The flowchart of the original OSort algorithm is shown in Fig. 2(a). The bandpass filtered neural signal is first applied to the spike detection module. Once a spike is detected, the distances between the new spike and all of the cluster averages are computed. If $d < T_C$, the spike is assigned to the cluster with minimum Euclidean distance, where d denotes the minimum distance between the detected spike and the stored cluster averages, and T_C denotes the cluster assignment threshold. Otherwise, a new cluster is created. When the new spike is assigned to a cluster, the mean waveform of the cluster is updated. Finally, the clusters are compared to one another to verify that indeed their distances are greater than or equal to a merging threshold T_M. If clusters are too close in distance, they will be merged together and the cluster distances are updated. The cluster assignment threshold T_C and the cluster merging threshold T_M are approximated as $T_C = T_M = T$, where T denotes the standard deviation of the filtered neural input signal [16]. While it is commonplace to normalize distance metrics with the number of data-points, it is computationally more efficient to multiply the threshold T [16]. As presented in [16], the optimal threshold for OSort sorting T_C and cluster merging T_M are computed separately. It is shown that the distances between an incoming spike and the spike follows a chi-squared χ^2 distribution. Therefore, the clustering threshold T_C can be calculated as the distance for which all points in a waveform belong to a cluster with probability $1 - \alpha$, where a is usually given as 5%. The cluster merging threshold T_M is computed as the number of standard deviations that clusters need to be apart before they are considered the same cluster.

From the hardware perspective, there are two important considerations when designing digital circuits for implementing the OSort algorithm. The first is the potentially infinite memory requirement. Because the algorithm has the ability to create new clusters, an efficient approach must be utilized to manage the creation of new clusters given a finite memory storage. The second challenge is designing the system for real-time operation. The original OSort flowchart in Fig. 2(a) shows that cluster averaging and cluster merging occur sequentially after a spike is assigned to a cluster. In the traditional offline processing, this can be readily done as all data recordings are accessible. In real-time operation, however, the OSort algorithm should not stop accepting incoming detected spikes while computing new cluster averages as well as checking for cluster merging. One solution is to utilize a queue for storing all newly detected spikes, however, implementing a queue would consume a relatively large amount of silicon area. Alternatively, we suggest to perform cluster merging and averaging concurrently to comparing incoming spikes to the existing clusters.

Fig. 2(b) shows the finite state machine (FSM) of the proposed parallelized OSort algorithm. The algorithm is divided into three stages: the *Assign Stage*, the *Averaging Stage*, and the *Merging Check Stage*. The *Assign Stage* is responsible for reading cluster average spike waveforms from the spike memory, comparing the minimum distance to the clustering threshold, and then assigning the new spike to an existing cluster or to a new one. The *Averaging Stage* reads the spike waveforms assigned to cluster memory for either cluster averaging or cluster merging. Finally, the *Merging Check Stage* compares the minimum distance between the newly averaged cluster spike waveform and all other existing cluster average spike waveforms. We assume that the incoming spikes to the sorting system have already been filtered. The data is processed by the detection and alignment modules, which produce a spike of p samples in length and is aligned to some particular feature, such as maximum energy or maximum amplitude. The distances between the incoming spike and all existing cluster average spikes are computed. The minimum distance $d_{\min}$ is compared to the threshold $\tau = p\sigma_r^2$, where σ_r denotes the standard deviation of the filtered input signal [16]. If $d_{\min} \leq \tau$, then the incoming spike will be assigned to the closest cluster. Otherwise, a new cluster will be created. If the M–th spike is assigned to an existing cluster, the cluster will be averaged. M can be chosen to be any particular number, but for hardware

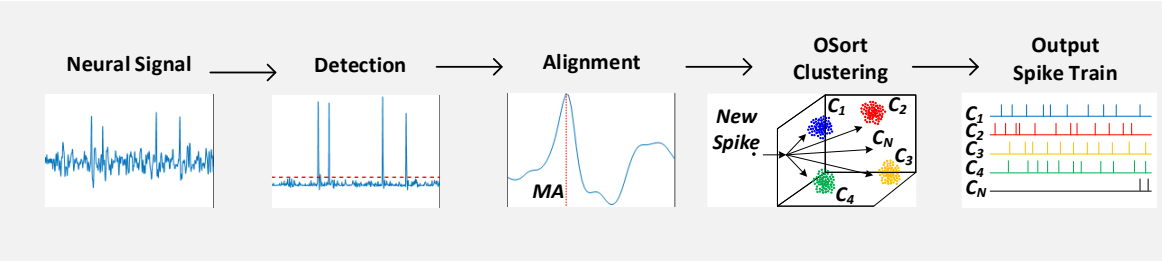

Fig. 3: Real-time spike sorting system.

implementation, it is selected as a power of 2, to replace division with an arithmetic right shift operation.

A key step in the OSort clustering process is computing the average of the cluster when a new spike is assigned to a cluster. Consider a spike cluster as a spherical object in a 3-dimensional xyz-space. The cluster average can then be considered as the average of all spikes within that sphere. When a new spike is added to a given cluster, the further away it is from other spikes in that cluster, the bigger an impact it will have on the cluster average. This would then cause a considerable change on the average dimensions of the sphere and hence, changing the average itself. A cluster may grow to the point in which it intersects a different cluster. Therefore, every time the cluster average is updated, the distances between the newly computed average and other existing cluster averages are calculated to ensure that the clusters themselves remain different enough to one another. If the minimum distance between the newly averaged cluster and any other cluster is less than τ, the average of those two particular clusters will be computed and assigned to one of those two clusters, which implements the cluster merging operation. To implement cluster averaging and cluster merging efficiently, two modifications are applied to the original OSort algorithm: (i) a cluster is averaged only if it was assigned its M-th spike. This allows us to avoid using complex computational resources, such as division units; and (ii) when a cluster average is updated, instead of exhaustively comparing all cluster averages to each other, only the newly updated cluster average is compared to all other clusters to check for potential merging.

To quantify the achieved relative performance increase resulting from the changes applied to the original OSort algorithm, the number of memory accesses (read and write) are measured. For a fair comparison, it is assumed that the original OSort algorithm has an upper bound on the number of clusters that can be supported, as well as the number of waveforms stored in each cluster. For every newly detected and aligned spike, both the original and modified OSort algorithms would require C clock cycles to read the average waveforms for the C clusters. The distance computation between two R-sample waveforms requires $2 \times R$ operations (one addition and one multiplication per sample of the two waveforms). One memory access is required for assigning the spike to a cluster. Because the original OSort algorithm performs the averaging of a cluster for every spike assignment, the number of operations required for averaging M waveforms is thus $(M-1)\times(R+1)$. Our modified OSort algorithm, however, performs the cluster averaging only on the M-th spike assignment, which requires only $R\times(M-1)+1$ operations. Moreover, the original OSort algorithm requires $C\times(C-1)$ memory accesses for assessing potential cluster mergers. In our proposed modification, the potential cluster mergers are only assessed for clusters that have just recently changed (i.e., their averages have been changed and merging may be required), which requires only $(M-1)$ memory accesses. Also, note that each merging assessment memory access requires $2\times R$ numerical operations to compute the distances between the spike waveforms. It is clear that our proposed modified OSort algorithm requires significantly fewer numerical operations and memory accesses compared to the original OSort algorithm.

IV. Architecture of the Real-time Spike Sorting System Hardware

The processing steps of our designed and implemented real-time spike sorting system is shown in Fig. 3. Spikes are first detected from ambient noise picked up by the recording electrodes. After detecting a spike, the new spike is aligned to the point of maximum amplitude *MA*. After alignment, the spike is passed to the OSort-based clustering module, which will assign the new spike to an existing cluster or a new cluster. Then, the clustering unit will output the cluster ID of the cluster to which the spike was assigned and hence, generating the output spike train.

A. Spike Detection Unit

Fig. 4 shows the architecture of the designed and implemented NEO-based spike detection unit. The input signal is shifted into the *NEO Shift Reg*, which is used to delay the input signal accordingly, to compute the energy of the signal at any given sample $x[n]$. The energy output $\psi[n]$ is then compared to the energy threshold τ_d. If the energy output matches or exceeds the given threshold value, the comparator will assert a "1" at the output, which indicates to the alignment unit that a spike is present in its respective buffer. The threshold value τ_d can be specified at the system level to a desired value . The NEO scaling factor C in (3) is estimated via the P_{FA} and P_D, which are computed during the offline parameter estimation phase. This allows the end-user to fine-tune the threshold value used for spike detection.

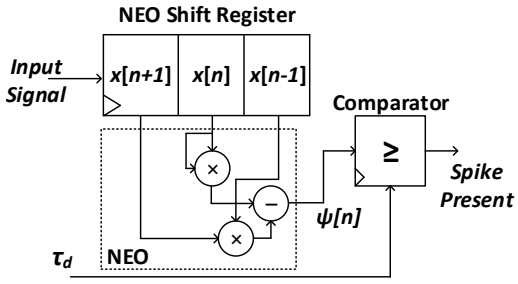

Fig. 4: Architecture of the NEO-based spike detection unit.

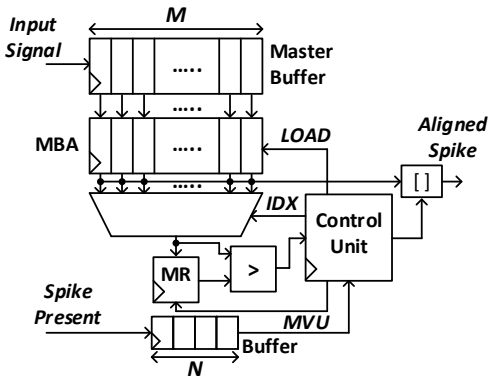

Fig. 5: Architecture of the spike alignment unit.

B. Spike Alignment Unit

To calculate the Euclidean distances among spikes accurately, spikes must be aligned prior to comparison. Aligning to the maximum amplitude is commonly used, as it is both computationally-simple and it also allows the clustering's Euclidean distance computation unit to readily differentiate spikes from one another. Because the stored cluster averages are also aligned to maximum amplitude, this requires the detected spikes to be aligned to the same metric to increase the consistency of the Euclidean distance unit.

As shown in Fig. 5, the spike alignment unit consists of two main buffers: the *Master Buffer* shift register, which receives new values of the input signal every clock cycle, and the *MBA* register, which is a parallel-in parallel-out register that copies the values stored in the *Master Buffer* when the *Spike Present* input signal is asserted by the spike detection unit. The *Spike Present* signal is delayed using an N-bit shift register to allow the detected spike to buffer more input data for a more accurate alignment. The *Control Unit* will assert the *LOAD* signal high to copy the contents of *Master Buffer* to *MBA*. The *Control Unit* then begins reading values from the *MBA* by controlling the select line *IDX* accordingly. The first value read from *MBA* is stored in the maximum value register *MR* by asserting the register enable signal *MVU* high. During all subsequent reads, the value read from the *MBA* will be compared to the value currently stored in *MR*, with *MR* being updated accordingly. The *Control Unit* also stores the index of the maximum value. The alignment point, which can be specified by the user as a parameter, is used such that the maximum amplitude of each spike lies at the chosen alignment point index. This helps prevent erroneous comparisons in the Euclidean distance unit that would occur from temporal shifts in the waveforms, which can induce a significant distance metric error.

Disregarding the *Spike Present* buffering latency, that is, the time of actual spike occurrence delayed by the buffers, the alignment unit takes 17 clock cycles to find the maximum value within the master buffer. The latency is directly related to the number of samples in a spike waveform, the chosen alignment point, and the size M of the master buffer and the *MBA*. The alignment unit begins reading values from the alignment point index to *max address*, which is computed as $M - (NSS - AP)$, where M, NSS, and AP denote the size of the *Master Buffer*, the number of samples in a spike waveform, and the chosen alignment point, respectively. For 64-sample spike waveforms, a buffer size of $M = 80$, and an alignment point of 23, the maximum value search spans 17 values starting from the alignment point. Assuming a 24 kHz sampling rate, the search spans 0.7 ms of signal time. Then the maximum index is used to part select the parallel output of *MBA*, shown by the [ ] block in Fig. 5, such that the maximum value is placed at the alignment point. The *Control Unit* also asserts an output flag to notify the OSort module that an aligned spike is ready for clustering.

One challenge during spike detection and alignment is due to the overlapping of spikes, i.e., when more than one spike resides in the detection window. While the refractory period of a neuron will prevent same-neuron overlapping, different neurons can fire at around the same time. Thus, the input to the system is the summation of signals at the electrode. In the worst case, our system will assign the overlapped spikes to its own cluster.

C. Parallel OSort Hardware Architecture

Fig. 6 shows the top-level block diagram of the designed OSort module. The primary inputs to the module are a *Spike Ready* input flag, which is asserted by the preceding spike alignment module to indicate that a spike is ready to be processed, and *Spike In* which is received in parallel from the alignment module. Other inputs to the module are the cluster distance threshold *ds* and the cluster merging threshold *dm*. The *DS* register holds the input spike for processing and is enabled via the *Spike Ready* input flag. The *DST*, *MIN*, and *COMP* units are responsible for computing distances, finding the minimum values, and comparing distances, respectively. The finite state machine *Assign FSM* reads cluster averages from the cluster memory via the *Assign raddr* port. It is also responsible for assigning the new spike to either an existing or to a new cluster. The *Assign FSM* handles when to enable the *MIN* unit as well as monitoring the output of the *COMP* unit for cluster assignment. For functional verification, the *Assign FSM* also keeps track of how many spikes have been assigned to each cluster.

The distance computation unit *DST* consists of p difference-squaring units and a pipelined adder tree of $p - 1$ two-input adders in $\log_2(p)$ stages to calculate the squared Euclidean distance between an input spike and an average spike from a particular cluster. After the *Assign FSM* has read the cluster

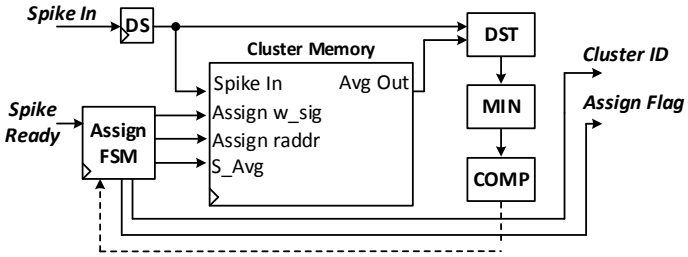

Fig. 6: Top-level block diagram of the designed OSort module.

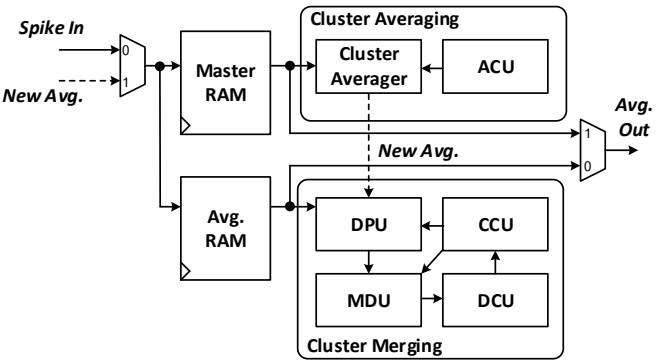

Fig. 7: Top-level block diagram of the cluster memory module.

averages, the *MIN* unit is enabled and will find the minimum distance and its associated cluster average. The *COMP* unit will then assert a flag if the minimum distance is less than the stored threshold d. If the flag is not asserted, then the spike is assigned to the cluster memory as a new cluster. If the flag is asserted, the *Assign FSM* will assign the new spike to the cluster that produced the minimum distance. The *Assign FSM* sets the cluster memory's write enable and write address via the *Assign w_sig* port. Additionally, if the M–th spike was assigned to a cluster, the *Assign FSM* triggers the cluster memory to perform the cluster averaging via the *S_Avg* port. Finally, regardless of whether the spike is assigned to an existing cluster or a new cluster, the *Assign FSM* generates the cluster identifier to generate the spike train output.

The block diagram of the cluster memory is shown in Fig. 7. The cluster memory not only stores clusters and their assigned spikes, it also has the ability to perform cluster averaging and merging. The *Assign FSM* reads cluster averages for comparing the incoming spikes to the averages. The cluster memory module consists of two main memory units: the *Master RAM* and the *Average RAM*. The cluster memory module stores the clusters and cluster average waveforms, which are maintained by performing cluster averaging and cluster merging. The *Master RAM* is responsible for storing the clusters; that is, storing all cluster averages and their assigned spikes. The *Average RAM* is responsible for storing only the cluster average waveforms. The cluster averages can come from either the *Master RAM* or the *Average RAM*, wherein lies a novel memory configuration scheme. By separating the averaging process from the *Master RAM*, the system is readily available to process incoming spikes into the *Master RAM* regardless of the current operational status of the averaging unit. This is due to the fact that the *Master RAM* also contains the average spike information, thus allowing the cluster management portion of the OSort algorithm to be parallelized alongside the incoming spike processing/assignment. A multiplexer is used to control which of the two memory units the *Assign FSM* reads from for incoming spike comparison.

To perform the distance comparison between clusters, the cluster memory module has a dedicated *Distance Pipeline Unit DPU*, a *Minimum Distance Unit MDU*, and a *Distance Comparator Unit DCU*. These units are all controlled by the *Clustering Control Unit CCU*. For computing the cluster averages, the *Cluster Averager* consists of p signed accumulators with a variable shifter at the output of each accumulator. The variable shifter is used to implement either cluster averaging (computing the new average for 2^n deep clusters) by right shifting n times, or to implement cluster merging (computing the average of two clusters) by right shifting once. The *Averaging Control Unit ACU* controls the averaging of spikes, which includes reading the *Master RAM* and starting the check for potential cluster merging (*Cluster Merging Stage*).

The cluster storage for the *Master RAM* is organized as shown in Fig. 8. Each spike cluster is alloted M rows of memory, where each row stores a spike waveform. The first row of a cluster is designated as the cluster average. When a spike is first assigned to an "empty" cluster, the assigned spike becomes the representative spike for that particular cluster, i.e., the cluster average. When a cluster is later averaged, the average spike waveform is again written to the first row of that particular cluster. The clusters are separated in this way to facilitate memory I/O via two addressing indices: the *clusterID* and the *cellID*. The *clusterID* is a $\log_2(K)$–bit identifier, where K denotes the number of total clusters supported by the hardware, and is used to index the clusters themselves. The *cellID* is a $\log_2(M)$–bit identifier, used in conjunction with the *clusterID*, to access a particular spike waveform stored in a particular cluster. Thus, the memory's read/write address is a concatenation of the two indices $\left[\textit{clusterID.cellID}\right]$.

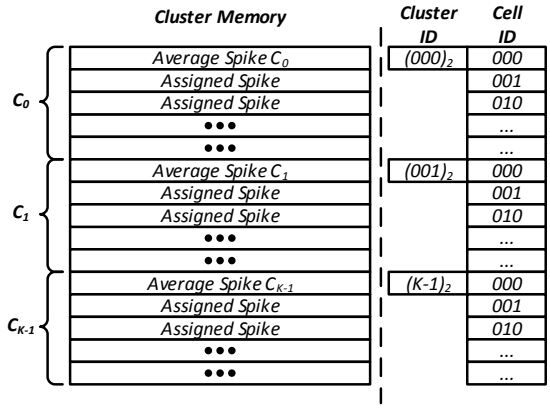

Fig. 8: Master RAM memory organization.

The *ACU*, *CCU*, and *Assign FSM* units work in unison to parallelize the OSort algorithm. The first operating stage is the *Assign Stage*. The *Assign FSM* reads average waveforms from the cluster memory and then assigns incoming spikes into

different clusters via the *clusterID* and/or *cellID* cluster indices. The *Assign FSM* keeps track of the next empty *cellID* for each cluster. When a spike is assigned to a particular cluster, the *cellID* for that cluster is incremented, such that the next spike assigned to that cluster has a valid memory address. If the increment causes an overflow (i.e. the increment returns the *cellID* to zero), this indicates that the cluster is now full, and the *Assign FSM* starts the averaging stage. The *Assign FSM* is now available to process another incoming spike.

During the *Averaging Stage*, the *ACU* reads the *clusterID* of the overflow from the *Assign FSM* and begins reading the spike waveforms stored in that cluster into the averaging unit. Once the new average is computed and written into the appropriate *clusterID* addresses in both the master and average RAMs, the *ACU* instructs the *CCU* to compare the newly computed average to other existing cluster averages for potential cluster merging. This initiates the third operating stage, which is the *Merging Check Stage*. The *CCU* reads cluster average waveforms and compares the distances between the newly averaged waveform and existing cluster averages. Once the minimum distance is computed, the *CCU* checks the value of the comparator's flag. If the flag is asserted, this indicates that two clusters must be merged. As the *CCU* does not have control over the averager, the *CCU* instructs the *ACU* which two clusters require merging. The design uses a *merge down* scheme which follows that the higher index is always merged into the lower index. As such, OSort will converge to more commonly occurring clusters located in the lower indices. The *Assign FSM* keeps track of the number of cluster merges and during distance computation in the *Assign Stage*, the *Assign FSM* will only compare the distance between the new spike and *clusterID*s 1 to $K - N_m$, where K denotes the number of supported clusters and N_m denotes the number of cluster merges. This avoids comparing to waveforms that rarely appear multiple times in the data, thus reducing the number of required comparisons. While OSort converges to the most commonly occuring spike waveforms, created transient clusters are handled in an efficient manner. Transient clusters will populate the higher indeces of the cluster memory, while, due to the merge down scheme, the true and accurate clusters of commonly occurring spike waveforms will populate the lower indeces of cluster memory. Transient clusters that are created due to mis-shapen or false-positive detected spikes will be overwritten by new transient clusters to conserve the memory. The more commonly occurring clusters are safely maintained in the lower indeces of the cluster memory. The convergence latency can be defined as the time it takes for OSort to accurately converge upon a couple of accurate cluster representations of the more commonly occurring waveforms. Because OSort will only update the cluster averages upon newly detected spikes, the convergence latency is not a deterministic value. Although the convergence latency is not deterministic and is also dataset dependent, the convergence latency can be estimated approximately as the duration of time before OSort no longer merges clusters.

For real-time spike sorting, the latency between actual spike appearance and classification is of vital importance. Because the *Assign Stage* has access to average waveforms, regardless of the status of the *Averaging* and *Merge Check* stages, the sorting latency only depends on the *Assign Stage*, in particular the number of currently populated clusters. Naturally, due to the detection buffer and alignment scheme, there will be some processing prior to OSort which will incur additional latency. After the initialization of threshold parameters, all clusters are seen as empty by the *Assign Stage* and skips any distance computation and comparison, thus OSort has a latency of one clock cycle for the first spike. During subsequent operation, the sorting latency can be calculated as $10+(A-N_m)$, where A denotes the number of assigned cluster averages and N_m denotes the number of cluster merges. Because clusters that are merged together populate lower-indexed memory locations, the more commonly appearing spike waveforms will gravitate toward the lowest 3-6 clusters, which indicates that the *Assign Stage* does not have to read the last N_m cluster averages and saves clock cycles by doing so. Ignoring the detection latency, the OSort module running at the sampling frequency of 24 kHz has a sorting latency ranging from 0.04 ms to 1.3 ms. In practical applications, it is recommended that OSort be run at a frequency much higher than the sampling frequency, as latency on the order of milliseconds may not be acceptable for real-time experiments.

V. Spike Sorting System Configuration and Simulation Results

The top-level block diagram of the parallelized OSort-based spike sorting system is shown in Fig. 9. The inputs to the system are the neural signal, the spike threshold for NEO-based spike detection, and the OSort maximum distance threshold. The detection and maximum distance thresholds can be changed in real-time during the system operation for fine-tuning.

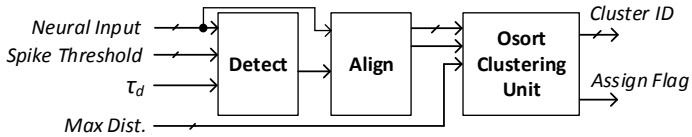

Fig. 9: The top-level block diagram of the parallelized OSort-based spike sorting system.

The dataset used for testing is the publicly available Wave_Clus dataset [7]. The dataset consists of simulated spike waveforms and is widely used as a benchmark for quantifying the performance of different spike sorting algorithms due to its ground truth information. We have performed testing using the Easy1_noise01, Easy2_noise005, and the Difficult1_noise005 datasets [7]. The datasets consist of spike waveforms from three different spike classes, along with background noise caused by distant spikes with signal-to-noise-ratios of 0.1 and 0.05, respectively. As presented in [7], the simulated datasets offer three distinct spikes with a Poisson distribution and a mean firing rate of 20 Hz. A refractory rate of 2 ms is applied to each spike class such that spiking cannot occur within that time frame. The noise added to the waveform is constructed from spike waveforms themselves, which renders the spike sorting more challenging, as the noise demonstrates a similar power spectrum to the actual spike data itself. The input to the

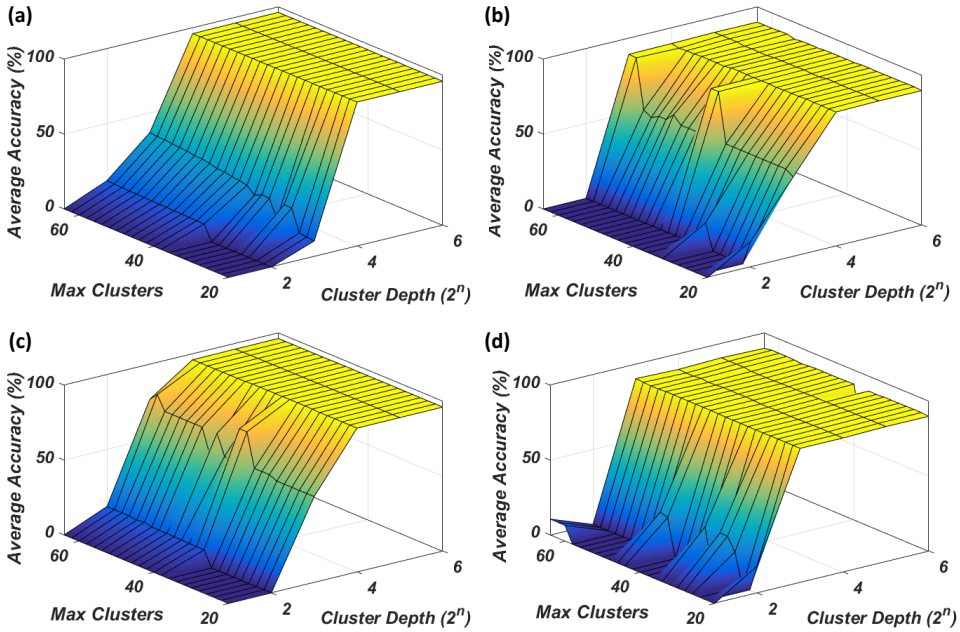

Fig. 10: (a) The floating-point software simulation results for the Easy1_Noise01 dataset; (b) the floating-point software simulation results for the Difficult1_Noise005 dataset; (c) the hardware simulation results for the Easy1_Noise01 dataset; and (d) the hardware simulation results for the Difficult1_Noise005 dataset for varying numbers of supported clusters and cluster depths.

system is represented using the (*WI*,*WF*) fixed-point numerical format, where *WI* and *WF* denote the number of bits allocated for the integer and fractional portions, respectively. The data is bandpass filtered (300 Hz – 3000 Hz) in MATLAB and then passed to our spike sorting system in (5,11) fixed-point format. For simulation and verification, the OSort module is configured to support 20 clusters with depths of 16 spike waveforms each stored in one row of the cluster memory. Note that all of these parameters are user-configurable during design elaboration. The distance computation units and averaging units are configured to match the number of samples in a spike waveform (64 in our case) to minimize the processing latency.

For an efficient in-vivo realization, it is important to estimate the number of clusters to support and to estimate the depth of the clusters for accurate sorting. For ASIC implementation, the number of supported clusters and depth of each cluster is fixed after implementation. We have performed an exhaustive simulation for the two given datasets while varying the maximum number of supported clusters and cluster depths. Figs. 10.a – 10.d show the floating-point and hardware simulation results for both datasets for a varying number of maximum clusters and varying cluster depths. For the Easy1 dataset, the NEO detection threshold scalar is set to $C = 5.6$, which results in a spike detection accuracy of 97% and the OSort sorting accuracies of 94%, 88%, and 88% for the three spike classes specified in the dataset, respectively. For the Easy2 dataset, the NEO detection scalar is set to $C = 4.04$ and results to a 98% detection accuracy. The OSort sorting accuracies for the Easy2 dataset are 96%, 94%, and 94% for the three distinct spike classes. For the Difficult1 dataset, the NEO detection threshold scalar is set to $C = 1.29$, which results in 97% spike detection accuracy. The OSort clustering accuracies are 85%, 85%, and 83% for the three distinct spike classes specified in the dataset, respectively. The clustering accuracy, also referred to as the sorting accuracy, is defined as the number of spikes assigned to the correct cluster divided by the number of spikes in the ground truth dataset that are associated with that cluster. One can see that the minimum number of supported clusters for accurate sorting performance is 20, while the minimum cluster depth is 16. It is estimated that this number of clusters and depth configuration is sufficient for accurate spike sorting.

We have quantified our system's performance with the commonly employed F-Score metric [18]. The F-score expresses the mean precision and sensitivity of the system as:

$$F = \frac{2T_P}{2T_P + F_P + F_N},$$

where T_P, F_P, and F_N denote the number of true positives, the number of false positives, and the number of false negatives, respectively. True positives denote spikes that have been detected and accurately classified by the system. True positive spikes have been verified against the ground truth dataset by matching the shape of the waveform as well as the time that the spike has occurred. False positives are defined as spikes that have been detected and classified by the system, but do not exist in the dataset. False negatives are defined as spikes that exist in the dataset, but are not detected and classified by the system. In testing the Easy1, Easy2, and Difficult1 datasets, our designed system achieves F-scores of 0.9493, 0.9694, and 0.9150, respectively.

Fig. 11 shows the generated spike train for parallel OSort clustering for the Difficult1 dataset. The x-axis shows the

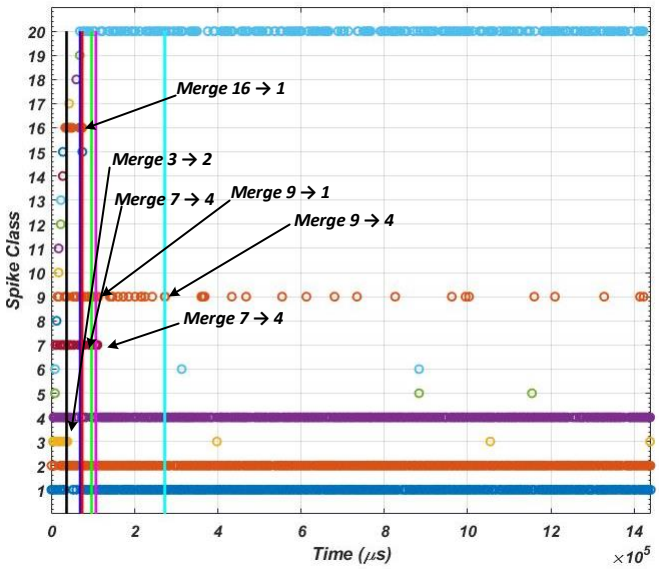

Fig. 11: Generated spike train for parallel OSort clustering with annotated cluster merging for the Difficult1 dataset.

time units in microseconds and the y-axis shows the spike classification (or class). Each dot on the plot represents the classification that OSort assigns to detected spike waveforms. The instances of cluster merging are also annotated. It can be seen that all merge instances merge toward the three most commonly occuring waveforms, clusters 1, 2, and 4. The reason that the assignment to cluster 20 (the final cluster) appears nearly fixed is because it serves as the holding point for the transient clusters that do not match existing average waveforms.

VI. FPGA and ASIC Implementation Results

Table I gives the characteristics and implementation results of our developed OSort-based spike sorting system along with those of the previously published work on FPGAs. Note that the variance of the results given in Table I is not primarily due to the differences among various FPGA devices. Different FPGA families are fabricated using different integrated circuit technologies and are optimized for various design constraints. For example, Xilinx Virtex FPGAs are generally intended for optimized performance (stronger drivers, more on-chip dedicated resources, etc.), while Xilinx Spartan FPGAs are optimized for lower power consumption and lower cost. For a fair comparison, we have synthesized our proposed OSort architecture on the same target devices as those used by the other previously published work, when able. Schäffer et. al. [19] presented an OSort-based clustering module. For a fair comparison, we have synthesized our design using the same target FPGA. While the number of registers, lookup-tables (LUTs), and block RAMs (BRAMs) used by our design are smaller, our design utilizes slightly more DSP48 dedicated multiplication units, which are required to parallelize the OSort algorithm. The work in [19] reports a maximum sorting latency of 18127 clock cycles, while our design will cluster and classify a spike with a maximum latency of 32 clock cycles. For both implementations, the latency is defined as the number of clock cycles for a spike to be clustered and classified from when it appears at the input of the OSort module. The design in [20] presents a spike sorting system based on PNN. They utilize a customized floating-point numerical representation consisting of 1 sign bit, 8 exponent bits, and 7 fraction bits. We have implemented our design on the same FPGA in fixed-point format with the same wordlength. While our design requires more resources, our design achieves a higher maximum clock frequency while clustering spikes with a significantly smaller latency. Note that the decrease in latency is not due to the higher maximum operating frequency, but because of arrays of processing elements used to compute spikes averages and distances between spikes in parallel, which significantly decreases computation time. For example, if the maximum frequency of our design was also 100 MHz, our sorting latency would be 0.32 μs, which is 560 times shorter than that of [19]. The work in [21] presents a real-time spike sorting system that uses Hebbian learning to implement PCA for projecting input spikes to features of interest. They present results for 16-bit wordlength, as well as modeling their designs using MATLAB's fixed-point toolbox, but do not report if any bits are allocated for the fractional portion of signals. Their target device is a Xilinx Spartan-6 FPGA and their resource utilization was reported based on the reconfigurable slices. We also implemented our designs using the same wordlength on the same target device. Note that the design in [21] only reports the implementation results of the Hebbian Eigenfilter hardware. Therefore, for a fair comparison, we have chosen to compare their Eigenfilter hardware to our implemented OSort clustering hardware, as both hardware units classify the detected spikes accordingly. We have estimated their register and LUT usage using the conversion formula in [25], which contains information about the Spartan–6 FPGA device family. It is shown that our OSort implementation uses fewer registers and BRAMs for the classification of spikes. The authors of [21] have not reported their operating frequency, nevertheless their reported projection time is 3 times longer than our clustering latency. Their spike sorting accuracy is 95% and 87% when sorting spikes from three and four neurons, respectively.

For a comparative analysis, we have also implemented a template matching-based spike sorting system [22] on a Xilinx Virtex-6 FPGA. Template matching avoids the computationally-daunting tasks of feature extraction and on-line clustering by performing them offline on a workstation. Pre-processing of neural recordings is used to estimate spike detection thresholds as well as generate spike templates. The templates are then utilized in a similar fashion to the cluster averages as described in Section III. As given in Table I, while the template matching-based design utilizes significantly fewer reconfigurable resources, it requires pre-processing of neural recordings for generating a set of template spikes. In contrast, the OSort-based spike sorting system can actively compute and maintain cluster averages without prior signal information. Due to its parallel computation, the designed OSort-based spike sorting system also clusters and classifies spikes in about half the time required by the template matching-based system.

TABLE I: The characteristics and implementation results of various spike sorting systems on FPGAs.

Work	Algorithm	Device	WL.WF	Regs. (%)	LUTs. (%)	BRAMs (%)	DSP48s. (%)	Max. Freq. (MHz)	Clustering Latency	Sorting Accuracy
[19]*	OSort	Zynq-7000	16.0	12150 (11%)	14037 (16%)	102 (72%)	120 (54%)	101	179.4 μs	–
Ours*	OSort	Zynq-7000	16.11	6103 (5.7%)	12701 (23.8%)	29 (20.7%)	128 (58.1%)	114	0.25 μs	–
[20]**	PNN	Virtex-6	16.7	3936 (1%)	13776 (7%)	7 (1%)	54 (4%)	100	6.7 μs	–
Ours	OSort	Virtex-6	16.11	8444 (2 %)	16472 (9 %)	29 (4 %)	130 (9 %)	123	0.25 μs	87%
[21]*	Hebbian	Spartan-6	16.0	∼8904 (5%)	∼6678 (7%)	65 (24%)	–	–	0.96 μs	95%
Ours*	OSort	Spartan-6	16.11	6127 (3%)	14188 (15%)	43 (16%)	128 (71%)	105	0.30 μs	87%
Ours [22]	Template Matching	Virtex-6	16.11	4880 (1%)	6635 (3%)	0 (0%)	5 (0.6%)	122	0.55 μs	90%
[23]	BOTM	Virtex-6	20.0	29000 (6%)	190000 (83%)	24 (6%)	–	–	2.65 ms	–
[24]	OSort	Virtex-5	16.8	16245 (27%)	23567 (40%)	63 (25%)	29 (4%)	100	11.1 ms	–
Ours	OSort	Virtex-6	16.11	8444 (2%)	16472 (9%)	29 (4%)	130 (9%)	123	0.26 μs	87%

*The authors of [19] and [21] present the FPGA implementation results for only the clustering and classification portions of their design, respectively. For a fair comparison, we also use the FPGA implementation results of our OSort clustering module only.
**The authors in [20] utilize the floating-point number representation with 1 sign bit, 8 exponent bits, and WF fraction bits.

TABLE II: The ASIC characteristics and implementation results of different spike sorting systems.

Design	Ours	Ours [22]	[26]	[27]	[28]	[29]	[30]	[31]	[32]
Algorithm	OSort	Template Matching	Feature Extraction	OSort	OSort	Feature Extraction	Feature Extraction	BOTM	Feature Extraction
Technology (nm)	32	45	90	65	45	45	130	40	65
Core voltage (V)	1.16	0.25	0.55	0.27	–	1.1	1.2	1.1	0.54
Operating frequency (kHz)	24	24	4000	480	56	960	160	500	3200
Area per channel (mm^2)	2.57	0.30	0.06	0.07	0.07	2.7	0.023	0.0175	0.003
Power per channel (μW)	2.78	0.064	2	4.68	10.3	20	0.75	19	0.175
Average accuracy (%)	87	90	77	75	93	84.5	–	93	86
Data rate reduction	1600×	3200×	11×	240×	278×	240×	–	–	257×

The design presented in [23] has implemented the Bayes optimal template matching (BOTM) algorithm for spike sorting on a Virtex-6 FPGA. While the maximum operating frequency has not been reported, the sorting latency was stated as 53 sampling cycles at a 20 kHz sampling rate. Our design achieves a sorting latency of at most 32 clock cycles. If both designs are operated at the sampling rate of 20 kHz, the sorting latency of [23] and ours are 2.65 ms and 1.6 ms, respectively. The work in [24] presents an FPGA-based spike-sorting accelerator employing the OSort algorithm. The design in [24] is intended for high-speed data processing of neural recordings on a workstation (offline), and the FPGA communicates with the workstation via a PowerPC processor on the interface board. The latency of 11 ms in Table. I is assumed with 24 kHz sampling rate. This is the worst case sorting latency, and correlates to 266 clock cycles. Our design achieves a significant reduction in real-time sorting latency, at only 32 clock cycles, which assuming a 24 kHz sampling rate, is only 1.33 ms. The design in [33] presents an FPGA-based, 128-channel spike sorting system, which utilizes OSort as a learning phase for template generation. Unfortunately, the resource utilization and the clustering latency of their design have not been reported. Note that both work in [19] and [24] present FPGA implementations of the original OSort algorithm. Our proposed parallel OSort algorithm results in reducing the number of utilized on-chip resources while significantly reducing the clustering latency. Similar to our template matching-based design, the work in [34] presents the template matching-based spike sorting system on a small Nano Igloo FPGA. While the relative resource utilization of their system is reported to be 98%, the utilization of registers, DSP48s, block RAMs, etc. is not specified and therefore, the work in [34] is not listed in Table I. Note that for a fair comparison with the reported results in the previously published work, the clustering latencies given in Table I refer to the OSort hardware operating at its maximum frequency. The FPGA implementation results are thus given as a benchmark and a relative measure to compare our implementation with other spike sorting realizations. Due to the parallelization of the algorithm and the novel memory configuration structure, the OSort module can be operated at the lower sampling frequency f_s of the neural signal, leading to a reduced power consumption.

The following steps are taken to use our proposed OSort-based spike sorting system in the laboratory. The user first sets initial detection threshold, clustering, and merging threshold values; The output of the recording electrodes is then fed into the spike sorting system. If spikes are being detected infrequently, the user may lower the detection threshold to allow more spikes to be detected, aligned, and clustered. For a better estimate of the threshold parameters, the user can use software tools to initially estimate the threshold values from previously recorded data, if it is available. One of the main advantages of using FPGAs in such a setting is that the reconfiguration of the system can be performed in a matter of minutes, should the user decide that more clusters should be supported or that the depth of each cluster is not sufficient. Additionally, tools such as integrated logic analyzers can be used to quantify the number of spikes assigned to each spike, and to view the merging and averaging of cluster waveforms in real time.

While our design supports online sorting of a single neural

channel, the advent of multi-electrode arrays (MEAs), such as the Utah Array [35] with the number of electrodes on the order of hundreds, provides challenges when sorting a relatively large amount of neural data from a large number of channels. Due to the algorithmic changes and novel memory configuration structure, our design need not be run at its maximum operating frequency for proper spike sorting. Because the maximum operating frequency of our design is more than 4000 times faster than that of typical electrode recording configurations (24 kHz), the OSort module can be time-multiplexed among different electrodes for multi-channel sorting. By time-multiplexing all of the arithmetic units in the OSort module, the main design constraint would be the storage overhead incurred by storing spike waveforms for each channel, with a memory offset counter used as an index for the cluster memory for each individual channel. The memory requirements for 20 clusters, each with a depth of 16, would be about 15 BRAMs on Xilinx FPGAs. For 64–channel sorting, the OSort module would need to operate at 1.53 MHz and the memory requirement would be about 960 BRAMs; 128–channel sorting would require a 3.07 MHz operating frequency and about 1920 BRAMs. These large-scale solutions would be suited for high-end FPGAs, such as Xilinx Virtex and Kintex devices. Note that the sorting latency would remain the same for all channels.

Various ASIC implementations if spike sorting algorithms have recently been presented [22], [26]–[32]. In [22], we implemented the template matching-based spike sorting system using three templates in a 45-nm CMOS process. The work in [26] performs NEO spike detection, aligns detected spikes to maximum derivative, and implements feature extraction via discrete derivatives. Their design consists of four 16-channel modules. The work in [27] detects spikes with the absolute value detection scheme and implements OSort clustering for a single 16-channel module. Our increased memory utilization is attributed in part by the parallelization of the OSort algorithm, in which incoming spike waveforms can be processed concurrently to the cluster management. Note that the numerical resolution of neural samples used in [27] is 8 bits per sample and 48 samples are used per neural waveform, which allows for significantly fewer storage elements required for implementing the OSort clustering. Similar to [27], in [28] spikes are detected using a voltage threshold, detected spikes are aligned to a maximum absolute amplitude, and OSort-based clustering is utilized. The work in [29] performs single-channel spike sorting via NEO-based detection, maximum amplitude alignment, and discrete-derivative feature extraction. It supports an unsupervised learning process, similar to the other feature extraction and OSort-based implementations. The work in [30] presents a multi-channel spike sorting ASIC based on feature-extraction. Their design uses decision trees for spike sorting in place of memory units. The design in [31] presents a multi-channel BOTM-based spike sorting ASIC with a built-in OSort learning phase. The work in [32] presents a multi-channel spike sorting processor based on integer coefficient feature extraction and clustering.

We have also implemented our designed OSort-based spike sorting system using a standard 32-nm CMOS process. The chip layout of our OSort-based spike sorting ASIC is shown in Fig. 12, which is estimated to occupy $2.57\ \mathrm{mm}^2$ of silicon area and consume $2.78\ \mu\mathrm{W}$ of power from a 1.16 V supply while operating at 24 kHz. Synthesis was performed using Synopsys Design Compiler, while place-and-route was performed using Synopsys IC Compiler. After the chip layout is completed, a simulation model is generated and used in a simulation setting. A variable change dump (VCD) file is then saved for estimating the power consumption of our ASIC design based on processing non-random, realistic input data. The memory units in the cluster memory module, implemented in BRAM resources on FPGAs, are implemented using standard cell library SRAM cells for ASIC. Table II gives the ASIC characteristics and implementation results of various spike sorting systems. The average sorting accuracy is defined as the number of correctly classified/sorted spikes per cluster compared to the spikes existing in the ground truth dataset [7]. Our template matching-based spike sorting ASIC [22], as well as the work in [28], [29], [31], [32], have utilized the same ground truth dataset to quantify the accuracy of their sorting system. The work in [26] used both synthetic data and real data, but the synthetic data is not the same data set as that given in [7]. The work in [27] and [30] both use real neural data for evaluating the accuracy of their system. For a general and well-accepted metric quantifying the performance of sorting, our ASIC and FPGA implementations of the proposed modified OSort algorithm achieve the same F-Scores. As the ASIC chips in [26], [27], [30]–[32] are multi-channel designs, the area and power consumption results for single-channel sorting are listed for a fair comparison. During spike sorting operation, the sampling rate is 24 Kbps. Each sample is represented using 16 bits, which results in an input bitrate of 384 Kbps. With an average neuron spiking rate of 40 spikes per second [1], and representing the *clusterID* with $\lceil log_2(20) \rceil = 5$ bits, and appending an assign flag as the 6–th bit, the output bit rate is $40\ \mathrm{spike/s} \times 6\ \mathrm{bits/spike} = 240\ \mathrm{bps}$. This results in a 1600 times data reduction rate compared to the sampling data rate. This large reduction in data rate significantly reduces the amount of transmission power required for the transmission of the cluster IDs. According to [36], the energy required to transmit one bit of data is 3 nJ, which results in a transmission power of roughly $0.72\ \mu\mathrm{W}$. Thus, the total power of our ASIC chip is $3.5\ \mu\mathrm{W}$, and the power density is $1.36\ \mu\mathrm{W/mm}^2$, which satisfies the tissue-safe requirements for brain implantable devices [37]. Therefore, our OSort-based spike sorting design consumes less power than some of the other published designs, reduces the data rate significantly compared to all other published designs, while delivering comparable spike sorting accuracy.

VII. Conclusion

This article presented the design and implementation of a spike sorting system utilizing the unsupervised OSort clustering algorithm. We proposed a modified OSort algorithm which significantly reduces the number of memory accesses and in turn reduces the number of numerical operations required for assessing the similarity between spike waveforms. The

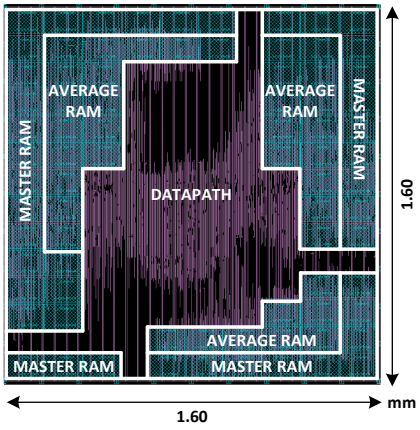

Fig. 12: The chip layout of the designed and implemented OSort-based spike sorting system.

proposed OSort hardware architecture utilizes a novel modified memory configuration scheme, allowing the system to process incoming spike waveforms and the cluster memory management concurrently, which parallelizes the original OSort algorithm. It is shown that our field-programmable gate array implementation offers sorting performance comparable to that of traditional software, while minimizing processing latency. Compared to the recently published works, our design can operate at higher frequencies, while minimizing the processing latency, which is ideal for real-time analysis of single-unit activity. Functional verification was performed by verifying the performance of our implemented system using a well-known simulated dataset. Our ASIC implementation results showed that our design consumes less power than the other state-of-the-art spike sorting ASIC implementations, has the highest data-rate reduction, while achieving an approximate sorting accuracy. The feasibility of our ASIC for in-vivo operation was discussed and confirms that the power dissipation of our ASIC implementation of the parallel OSort algorithm is safely within the tissue-safe requirements.

ACKNOWLEDGMENT

This work was supported by the Center for Neurotechnology (CNT), a National Science Foundation Engineering Research Center (EEC-1028725).

REFERENCES

[1] S. Gibson, "Neural spike sorting in hardware: From theory to practice," Ph.D. dissertation, University of California Los Angeles, 2012.
[2] M. S. Lewicki, "A review of methods of spike sorting: the detection and classification of neural action potentials," *Network: Comput. Neural Syst.*, vol. 9, no. 4, pp. R53 – R78, 1998.
[3] I. Obeid and P. D. Wolf, "Evaluation of spike-detection algorithms for a brain-machine interface application," *IEEE Transactions on Biomedical Engineering*, vol. 51, no. 6, pp. 905–911, 2004.
[4] J. F. Kaiser, "On a simple algorithm to calculate the 'energy' of a signal," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1990, pp. 381 – 384.
[5] K. H. Kim and S. J. Kim, "A wavelet-based method for action potential detection from extracellular neural signal recording with low signal-to-noise ratio," *IEEE Transactions on Biomedical Engineering*, vol. 50, no. 8, pp. 999 – 1011, 2003.
[6] H. Hotelling, "Analysis of a complex of statistical variables into principal components." *Journal of Educational Psychology*, vol. 24, no. 6, p. 417, 1933.
[7] R. Quiroga, Z. Nadasdy, and Y. Ben-Shaul, "Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering," *Neural Computation*, vol. 16, no. 8, pp. 1661 – 1687, 2004.
[8] J. J. Capowski, "The spike program: A computer system for analysis of neurophysiological action potentials," in *Computer Technology in Neuroscience*, P. B. Brown, Ed. Washington: Hemisphere Publishing Corporation, 1976, ch. 17, pp. 237–251.
[9] X. Liu *et al*., "A fully integrated wireless compressed sensing neural signal acquisition system for chronic recording and brain machine interface," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 10, no. 4, pp. 874–883, 2016.
[10] J. Zhang *et al*., "A closed-loop compressive-sensing-based neural recording system," *Journal of Neural Engineering*, vol. 12, no. 3, pp. 1–17, 2015.
[11] A. Eftekhar, E. P. Sivylla, and G. C. Timothy, "Towards a next generation neural interface: Optimizing power, bandwidth and data quality," in *Biomedical Circuits and Systems Conference*, 2010, pp. 122–125.
[12] G. Bi and M. Poo, "Synaptic modification by correlated activity: Hebb's postulate revisited," *Annual Review of Neuroscience*, vol. 24, no. 1, pp. 139–166, 2001.
[13] Z. Nadasdy, R. Q. Quiroga, Y. Ben-Shaul, B. Pesaran, D. A. Wagenaar, and R. A. Andersen, "Comparison of unsupervised algorithms for on-line and off-line spike sorting," in *Proceedings of the Annu. Meeting Soc. for Neurosci.*, 2002.
[14] A. Zviagintsev, Y. Perelman, and R. Ginosar, "Low-power architectures for spike sorting," in *IEEE EMBS Conference on Neural Engineering*, 2005, pp. 162–165.
[15] T. I. Aksenova, O. K. Chibirova, O. A. Dryga, I. V. Tetko, A.-L. Benabid, and A. E. Villa, "An unsupervised automatic method for sorting neuronal spike waveforms in awake and freely moving animals," *Methods*, vol. 30, no. 2, pp. 178–187, 2003.
[16] U. Rutishauser, E. Schuman, and A. Mamelak, "Online detection and sorting of extracellularly recorded action potentials in human medial temporal lobe recordings, in vivo," *Journal of Neuroscience Methods*, vol. 154, pp. 204 – 224, 2006.
[17] Z. Frehlick, I. Williams, and T. G. Constandinou, "Improving neural spike sorting performance using template enhancement," in *IEEE Biomedical Circuits and Systems Conference*, 2016, pp. 524–527.
[18] C. J. V. Rijsbergen, *Information Retrieval*, 2nd ed. Newton, MA, USA: Butterworth-Heinemann, 1979.
[19] L. Schäffer, Z. Nagy, Z. Kineses, and R. Fith, "FPGA-based neural probe positioning to improve spike sorting with OSort algorithm," in *IEEE International Symposium on Circuits and Systems*, 2017, pp. 1–4.
[20] D. Wang, Y. Hao, X. Zhu, T. Zhao, Y. Wang, Y. Chen, W. Chen, and X. Zheng, "FPGA implementation of hardware processing modules as coprocessors in brain-machine interfaces," in *International Conference of the IEEE Engineering in Medicine and Biology Society*, 2011, pp. 4613 – 4616.
[21] B. Yu, T. Mak, X. Li, F. Xia, A. Yakovlev, Y. Sun, and C. Poon, "Real-time FPGA-based multichannel spike sorting using hebbian eigenfilters," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 4, pp. 502 – 515, 2011.
[22] D. Valencia and A. Alimohammad, "An efficient hardware architecture for template matching-based spike sorting," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 3, pp. 481–492, 2019.
[23] J. Dragas, D. Jäckel, A. Hierlemann, and F. Franke, "Complexity optimization and high-throughput low-latency hardware implementation of a multi-electrode spike-sorting algorithm," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 23, no. 2, pp. 149–158, 2015.
[24] S. Gibson, J. W. Judy, and D. Marković, "An FPGA-based platform for accelerated offline spike sorting," *Journal of Neuroscience Methods*, vol. 215, no. 1, pp. 1–11, 2013.
[25] Xilinx, "Spartan-6 FPGA configurable logic block user guide," 2010.
[26] V. Karkare, S. Gibson, and D. Markovic, "A 130-μw, 64-channel neural spike-sorting DSP chip," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 5, pp. 1214–1222, 2011.
[27] ——, "A 75-μw, 16-channel neural spike-sorting processor with unsupervised clustering," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 9, pp. 2230–2238, 2013.
[28] Y. Liu, J. Sheng, and M. C. Herbordt, "A hardware design for in-brain neural spike sorting," in *IEEE High Performance Extreme Computing Conference*, 2016, pp. 1–6.

[29] M. Zamani, D. Jiang, and A. Demosthenous, "An adaptive neural spike processor with embedded active learning for improved unsupervised sorting accuracy," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, no. 3, pp. 665–676, June 2018.

[30] Y. Yang, S. Boling, and A. Mason, "A hardware-efficient scalable spike sorting neural signal processor module for implantable high-channel-count brain machine interfaces," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 4, pp. 743–754, 2017.

[31] H. Xu *et al*., "Unsupervised and real-time spike sorting chip for neural signal processing in hippocampal prosthesis," *Journal of Neuroscience Methods*, vol. 311, pp. 111–121, 2019.

[32] A. Do *et al*., "An area-efficient 128-channel spike sorting processor for real-time neural recording with 0.175 μW/channel in 65-nm CMOS," *IEEE Transactions on VLSI Systems*, vol. 27, no. 1, pp. 126–137, 2019.

[33] J. Park, G. Kim, and S. Jung, "A 128-channel FPGA-based real-time spike-sorting bidirectional closed-loop neural interface system," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 25, no. 12, pp. 2227–2238, 2017.

[34] S. Luan *et al*., "Compact standalone platform for neural recording with real-time spike sorting and data logging," *Journal of Neural Engineering*, vol. 15, no. 4, 2018.

[35] R. R. Harrison *et al*., "A low-power integrated circuit for a wireless 100-electrode neural recording system," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 1, pp. 123–133, 2007.

[36] F. Chen, A. P. Chandrakasan, and V. M. Stojanovic, "Design and analysis of a hardware-efficient compressed sensing architecture for data compression in wireless sensors," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 3, pp. 744–756, 2012.

[37] S. Kim, P. Tathireddy, R. A. Normann, and F. Solzbacher, "Thermal impact of an active 3-d microelectrode array implanted in the brain," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 15, no. 4, pp. 493–501, 2007.