FPGA Based Platform for Neural Spike Sorting

Bach Vu
(SID: 203-775-875)

3/13/2015
Table of Contents

Table of Contents........................................................................................................................................2

1 Introduction..................................................................................................................................................3
  1.1 Motivation...............................................................................................................................................Error! Bookmark not defined.
  1.2 Project Goals..........................................................................................................................................Error! Bookmark not defined.
  1.3 GiDEL PROCStar IV 530 FPGA........................................................................................................Error! Bookmark not defined.

2 Current Design...........................................................................................................................................4

3 Implementation of the Spike Sorting........................................................................................................5
  3.1 Spike Detection......................................................................................................................................5
  3.2 Spike Alignment......................................................................................................................................5
  3.3 OSORT.....................................................................................................................................................6
    3.3.1 Original Algorithm..........................................................................................................................6
    3.3.2 Updated OSORT Algorithm..........................................................................................................7
    3.3.3 Inputs and Outputs.............................................................................................................................8
    3.3.4 State Machine....................................................................................................................................8
    3.3.5 Comparison of New vs Original Algorithm......................................................................................10

4 Results......................................................................................................................................................11
  4.1 Accuracy..................................................................................................................................................11
    4.1.1 Spike Detection.................................................................................................................................11
    4.1.2 Spike Alignment.................................................................................................................................11
    4.1.3 OSORT...............................................................................................................................................12
  4.2 Performance Metrics...............................................................................................................................15
    4.2.1 Timing Analysis.................................................................................................................................15
    4.2.2 Processing Speed..............................................................................................................................16

5 Future Work.............................................................................................................................................17

6 Conclusion................................................................................................................................................17

References....................................................................................................................................................18
1 Introduction

1.1 Motivation

Recently, there has been an increase in demand for biomedical engineering applications. One such application is the analysis of neural activity. Neural spikes are detected via an electrode. The electrode detects the neuron’s activity as a voltage. The voltage is then sampled at a rate between 10-40kHz. The resolution is anywhere from 12-24 bits. However, as with any analog signal, there is a large amount of noise associated with the signal. This leads to the use of digital signal processing techniques since they generally have lower noise than analog techniques.

The next thing to consider is the amount of data you collect. For a neural spike sorting application, an 8 hour experiment would collect about 100GB of data [1]. This means you need a tremendous amount of computational power to sort the data quickly. Using a conventional CPU and processing tools would take approximately 30 hours to sort the data (data rate of 0.95MBps) [1]. Using dedicated hardware like a field programmable gate array (FPGA) or application specific integrated circuit (ASIC) can help greatly increase the sorting speed.

1.2 Project Goals

The goal of the project is to develop the groundwork for an OSORT clustering algorithm on the PROCStar IV 530 FPGA. We want to implement an algorithm that can quickly and efficiently detect, align, and sort neural spike data. Other projects were done with the board concurrently to help lay some basic building blocks for the OSORT algorithm. A detection and alignment algorithm were made to go hand in hand with the OSORT algorithm, but modifications and improvements were made to those to increase the scope of the sorting capability. Creating a block with high speed and minimum latency can potentially lead to real time applications.

1.3 GiDEL PROCStar IV 530 FPGA

To reduce the runtime needed to sort and cluster the spikes, a high speed high performance FPGA is needed. Having a high speed, high performance FPGA helps increase the data throughput for the spike detection, alignment, and clustering. For this project, we use the GiDEL PROCStar IV 530. The system has 16 GB of external memory and 2 GB of on board memory (18 GB total), is capable to running up to 300MHz, and is an 8-lane PCIe hosted system with four Altera Stratix IV 530 FPGAs. A top level system view of the FPGA is shown in Figure 1 [4].
A program called ProcWizard also comes with the GiDEL package. It is a hardware-software integration application designed to facilitate project development. ProcWizard automatically generates internal buses and PCI drivers. Items in the design such as modules, registers, etc. can be defined in the program. A template with its corresponding hardware and software design files can then be automatically generated. These generated files are the interface modules and regulate the communication protocol for the system.

As seen in Figure 1, the text in blue is developed by the developer. ProcWizard automatically generates and configures the other blocks. Again, this is to help facilitate the design process.

2 Current Design

The current design is based on Sarah Gibson’s paper on an FPGA based spike sorting algorithm [1]. The input spikes are quantified by a word length of 16 bits. There are 8 signed integer bits, and 8 fractional bits. The 8 signed integer bits can support values between -128 and 127. The 8 fractional bits allow for precision of 1/256.

Currently, the data is stored in binary files. Each file is treated as a different channel. An existing MATLAB script provided by Sarah converts the data to this format.
The binary files are then the inputs to the FPGA’s detection block. Whenever a spike is detected, it passes the information and spike to the alignment block. The outputs of the alignment block go to the clustering block where the OSORT algorithm is then performed.

The output files are save as .dat files. This goes to the host computer where the GUI shows you the spikes, detected timestamp, and cluster means.

### 3 Implementation of the Spike Sorting

#### 3.1 Spike Detection

The purpose of this block is to determine if a spike has been detected or not. This block was implemented by Aria Sarraf as part of his MS project [4]. The module is implemented in verilog for the purpose of decoupling it from the other blocks. That way, this module does not interfere with the other modules and vice versa. However, additional changes and improvements have been made to make the system have a wider range of capabilities.

The inputs are the current sample of the spike you are looking at and the detection threshold. The output is a flag that tells you if a spike has been detected or not.

The original block was only able to detect positive spikes. The way it worked was it would compare the current sample with a user defined threshold. If the sample was greater, then a spike is detected. Otherwise, you check the next sample. But what if you wanted to detect spikes that have negative values? The module in its current configuration would not be able to do that.

Changes were made such that the module would be able to detect a spike against a negative threshold. Changes were made to the state machine and the logic to allow for this.

So now the module can detect a spike against a positive or negative threshold depending on what the user inputs. The changes were also made in such a way that neither an additional input nor multiplexor are needed. The module will perform the correct actions whether you put in a negative or positive threshold. It is automatic. Removing the need for a multiplexor can trim down the hardware requirements and latency when detecting spikes.

Additionally, the module was designed in such a way that additional changes can be made to the module. The system is open to design changes and a more complex detection algorithm if that is desired.

#### 3.2 Spike Alignment

The spike alignment block was implemented by Aria Sarraf [4]. However, additional changes and improvements were made to make a more capable system.

The inputs are the current timestamp and the current spike value. The outputs are the max or min spike value, depending on whether you’re detecting a positive or negative spike, and the timestamp at which the max or min occurred.

After a spike has been detected, the next step is spike alignment. If a spike was detected by the detector, then it passes the information to the alignment block. The alignment block saves the timestamp and value of the current sample that has surpassed the threshold.
The alignment then grabs the next timestamp and checks the spike value. If it is greater than your previous value, then it is saved as your max value. This process is repeated until you reach your spike width. The block then outputs your max value and the timestamp at which it occurred.

The first sample taken is the first after you have surpassed the threshold. The alignment block then collects the next N samples where N is the user desired spike width. The red highlights the samples that have been taken. The spike is then saved to memory. The maximum (or minimum) and timestamp at which it occurred are sent to the OSORT block.

As stated earlier, additional changes were made to Aria’s original program to make a better spike alignment program. The previous iteration was only able to save a positive value and compare against a positive threshold. The new version can take a signed input and carry out the logic with a minimum negative value in addition to the previously done positive value.

No additional input or multiplexor is needed. The alignment block will carry out the correction logic depending on the inputs from the detection block. This is beneficial because we get additional functionality with minimal additional hardware.

### 3.3 OSORT

The preceding blocks as mentioned are the detection and alignment block. The detection block detects spikes, the alignment block gives you a spike max or min and an associated timestamp, and the OSORT block outputs the cluster means. A block diagram shown in Figure 3 shows the data flow path. The major inputs and outputs are included in the diagram.

![Figure 2: Data Flow Path](image)

#### 3.3.1 Original Algorithm

The original OSORT algorithm is shown below in Figure 4. It is taken from Sarah Gibson’s thesis paper [3].
The advantage of using the OSORT clustering algorithm versus other clustering algorithms is that OSORT has the combination of low complexity and good accuracy. There are other more accurate clustering techniques, but they are far more complex and as a result would require more hardware and processing power. So for the application of neural spike sorting, OSORT is the preferred method.

### 3.3.2 Updated OSORT Algorithm

At a glance, you can see that there are some changes that can be made to the algorithm to make it more efficient. In step 2, instead of using the Euclidean distance ($L_2$ norm), we can use the $L_1$ norm.

The $L_2$ norm is calculated as:

$$L_2 \text{ norm} = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

The $L_1$ norm is calculated as:

$$L_1 \text{ norm} = (y_2 - y_1) + (x_2 - x_1)$$

This greatly reduces the calculations which can reduce the latency and hardware requirements. This is because the $L_2$ norm requires a square root and squaring calculation, which are not trivial to implement in Verilog.

The next change we can make is with step 4. Step 4 states that you need to check every cluster with every other cluster to make sure they are far away enough to be their own cluster. However, this requires unnecessary calculations. In most cases, the merging threshold and sorting threshold are the same. So it does not help to check every cluster with every other cluster. This is because if they didn’t merge when the spike first comes in and you initially check, then they will not merge when you do the sorting check (if the merging and sorting threshold are the same.) So rather than checking every cluster
with each other cluster, you only need to check a cluster against the others if the cluster mean changes. So if a spike comes in and you merge it to an existing cluster, then that cluster is the one you need to check with the others.

3.3.3 Inputs and outputs

The inputs to the OSORT block are; the system clock, reset, spike detect, spike x position (timestamp), spike y position (max or min value), the merging threshold, the sorting threshold, and done. The outputs are the cluster x position, the cluster y position, and ready.

As stated earlier, in general, the merging threshold and sorting threshold are the same. The spike detect is high if a spike x and spike y value are fed into the OSORT block. The ready flag then becomes low. The ready flag becomes high only when the block is done processing the current spike. The done input tells the module that there are no more incoming spikes and to print out the cluster values.

3.3.4 State Machine

Figure 5 details the state machine. Each state will be described in depth further down. It is also worth noting here that the difference between merging and sorting is whether or not you are looking at a new spike or not. When a new spike comes in, you are sorting it, so we refer to this as the cluster sort check. If it happens to merge with an existing cluster, then we check if it is close enough to any clusters to merge with it. This is referred to as the merge check.
The first incoming spike brings the state machine to state 1, assigning the first spike to its own cluster. It then goes to state 2.

State 2 is an idle state. If nothing is being done, the system does nothing. It waits until the next piece of information comes in to be processed.

If a new spike comes in, and it’s not the first spike, we go to state 3, the cluster sort check. We check and see if the current spike is close enough to any existing clusters to merge with it.

If the new spike is not close enough to merge with any existing clusters, we go to state 4, the new cluster state. We assign the current spike to its own cluster. After this is done, the system returns to state 2, the idle state.

If the spike does merge with an existing cluster, we go to state 5, the cluster merge check. First we recalculate the cluster mean, and then check if the new cluster can merge with any existing cluster. If it does, then we stay in state 5 and repeat the process until the clusters can no longer be merged. Otherwise, we go back to state 2, the idle state.
3.3.5 Comparison of New vs Original Algorithm

Changes to the state machine are not the only changes that were implemented in the updated version of OSORT. Some changes were made to the logic to cut down on the required hardware and latency as well.

The first change in our algorithm is the way we process the spikes. In the old algorithm, the entire spike would be passed to the OSORT block [1]. So the number of samples passed to the clustering block would be equal to the spike width. In Sarah’s algorithm, the spike width was 36 [1]. However, in the updated OSORT, only the spike max/min and associated timestamp are passed to the OSORT block. The spike can be saved to memory in the alignment block and be accessed later if need be. But the only information needed to calculate the cluster mean are the spike max/min and timestamp. This reduces the latency of the processing actions by N clock cycles where N is the spike width. So rather than update the cluster mean using all of the spike samples, you can update it by just using the max/min and timestamp. This greatly reduces clock cycles required to process the data and thereby the latency as well.

Additionally, the logic is set up in such a way that the new cluster mean can be recalculated in a single clock cycle. We know that to calculate the mean of N different numbers, we sum the numbers up and divide by N. The mathematical formula is shown below.

\[ \frac{x_1 + x_2 + x_3 + x_4}{4} = \text{old cluster mean} \]

So to calculate the cluster mean, we simply need to sum up the timestamps and divide by the number of spikes, and sum up the max/min values and divide by the number of spikes. This gives us a 2-dimensional cluster mean. At a glance, it seems that you have to read all the spike values from memory, sum them up, and divide by the number of spikes to calculate your cluster mean. But there is a much faster and easier way.

The only additional information you need is the number of spikes in a given cluster. So say a new spike comes in and it needs to be merged with an existing cluster. All you need to do is take the existing cluster mean, multiply it by the number of spikes in the cluster excluding the new spike, add the new spike value to it, and divide it by the number of spikes in the cluster including the new spike. This way, you don’t have to pull the max/min values and timestamps from memory to recalculate the cluster mean. The steps below show how to recalculate the mean for a 1-dimensional case. The process needs to be done for the spike timestamp and max/min value to obtain the new 2-dimensional cluster mean.

\[ \frac{x_1 + x_2 + x_3 + x_4}{4} = \text{old cluster mean} \]

\[ x_5 = \text{new spike} \]

\[ x_1 + x_2 + x_3 + x_4 + x_5 = \text{cluster mean} \times 4 + x_5 \]
Because of the changes to the OSORT algorithm, the latency of the system is considerably less. In Sarah’s algorithm, the latency was 266 clock cycles. In this system, it is 55 clock cycles.

We get this number by first assuming the maximum number of clusters is ~10. This comes from the size of the electrodes. Each electrode is connected to 2-10 neurons [3]. That means that there will be 2-10 clusters. In general, there are 5-6 clusters after sorting [3]. But we will assume 10 clusters at the end to look at the worst case latency of the system. So given the max number of clusters, the worst case latency of the system is 55 clock cycles. This comes from the changes we implemented in the OSORT algorithm.

Assume that there are already 10 separate clusters. If a new spike were to come in, then it would have to be sort checked against all 10 existing clusters. If it were to merge with the last cluster, then you have to recalculate the cluster mean and check the new cluster against every existing cluster. This takes 10 clock cycles. Now you have to check the new cluster against 9 clusters. If it merges on the 9th cluster, then you recalculate the mean. This takes 9 clock cycles. If this process continues and the clusters keep merging all the way down to 1 cluster, then the number of clock cycles can be described by the equation below.

\[
\sum_{i=0}^{\max \text{ number of clusters}} (\max \text{ number of clusters} - i)
\]

And since we are assuming the maximum number of clusters is 10, we get a system latency of 55 clock cycles. This is a great improvement and allows for a greater data processing capabilities.

4 Results

The test for the spike detection and spike alignment were run with 16 bit inputs sampled at 27.77 kHz. The spike threshold was set to 30. The spike width for the spike alignment was also set to 30.

4.1 Accuracy

4.1.1 Spike Detection

The spike detection was successful in detecting spikes. This was expected considering the simple detection method we used.

4.1.2 Spike Alignment

The spike alignment was successful in recording the maximum spike value and timestamp at which it occurred. Figure 5 shows a plot detailing the alignment block.
Figure 5: Spike Alignment [4]

The first red sample shows when the spike was first detected. The red shows the spike data you saved. The blue is the raw data. The alignment accuracy was tested by first looking at the data and manually recording the spike max value and timestamp. That was then compared to the output of the alignment block. The output was exactly what we expected.

4.1.3 OSORT

We first want to test the accuracy of the OSORT block with known test cases to make sure the logic is doing what we want. We first make sure the OSORT block can correctly store both negative and positive values correctly. This means that no merging will take place. We input the spike x position as being 10-50 in increments of 10. We input spike y position of being the same as the x positions. We set the merging and sorting threshold to 10. This means that the L1 norm will equal 20 or greater in all cases, meaning every spike will be its own cluster. Figure 6 shows the results.
We now want to make sure our system can do the same thing, but with negative values. Figure 7 shows the results.

Now we want to check if our system will know when to correctly merge clusters and keep them separate. We input the spike x position as 10-100 with increments of 10. We input the y position as the same thing. We set the merging and sorting threshold as 20. We see immediately that the 2nd spike will merge with the first giving us a cluster mean of (15,15). The next spike which is (30,30) will be too far to be considering close to the one existing cluster. The L1 norm is (30-15) + (30-15) = 30, which is greater than the sorting threshold. So it will be put into its own cluster. The next spike is (40,40) which merges with (30,30) to give us a cluster mean of (35,35). This process will continue until there are no more spikes. We see that the cluster mean values will be (15,15), (35,35), (55,55), (75,75), (95,95). Figure 8 shows the results.
The final test case we input is only slightly more complicated. We keep the merging and sorting threshold at 20. We then input the spikes (10,10), (22,22), and (16,16) in that order. (10,10) will be assigned to its own cluster. (22,22) will not merge with the existing cluster because the L1 norm is equal to 24, which is greater than the sorting threshold. When (16,16) comes in, it will merge with (22,22) because the L1 norm is only 12. The new cluster mean will be (19,19), which is then checked against (10,10). We see that the L1 norm is 18, so the two clusters will merge. This gives us a final cluster value of (16,16). Figure 9 shows the results.

The (22,22) is shown simply for testing purposes. We wanted to make sure the cluster mean was getting stored. The index in the array however is disabled since it has already been merged, so the next spike to come in would not be checked against it.

The OSORT algorithm is able to correctly store and merge clusters, which is exactly what we want it to do. These simple test cases show that the program can be extended to larger sets of data with confidence that it would correctly cluster the data.
4.2 Performance Metrics

As stated earlier, the improvements in this OSORT algorithm have reduced the latent clock cycles of the OSORT block to a maximum of 55 clocks cycles. This is an improvement over Sarah’s existing OSORT algorithm, which has a latency of 266 clock cycles [1]. That means the latency of the new OSORT block is roughly 20% of the old one, not accounting for the clock speed. So we are confident that with a better FPGA that is capable of running at higher speeds and improved latency, we are able to sort spikes at a higher data rate than what was previously done.

However, we are not able to do a direct comparison because we are missing the read and write to memory, python wrapper, and MATLAB wrapper. But we can extrapolate performance based on previous work. For example, it is reasonable to assume an equivalent python and MATLAB wrapper. We can also assume that the read and write to memory on this FPGA is faster than the read and write to memory on the previous FPGA. However, we will assume that they are the same to provide for a worst case processing speed.

4.2.1 Timing Analysis

Figure 10 shows the timing analysis of Sarah Gibson’s spike sorting program [1].

![Figure 10](image)

As shown earlier, the latency of our program is only 20% of what was done previously. Therefore, we can assume the FPGA processing goes to 20% of its value. This is assuming the timing of everything else stays the same. This would make the FPGA processing for us only 10µs. The timing analysis for a similar system based on our work is shown in Figure 11.
Theoretically, we should have a 13.5µs improvement over the whole cycle. This is limited to the time constraints outside of the FPGA. The bulk of the time is spent on the system buffer and system call. A fair amount is spent on the memory read and python interpreter. Very little time is dedicated to FPGA processing.

### 4.2.2 Processing Speed

The old algorithm was able to process an 8120-byte packet in 136µs on average [1]. This comes to a processing speed of 60 MBps. With our improved algorithm, we can extrapolate that it would take is 122.5µs for the same amount of data. This gives us a data rate of roughly 66 MBps. That would be a 10% improvement. This is also a significant improvement over other previous spike sorting methods. Table 1 shows a performance benchmark when compared to other sorting methods.

<table>
<thead>
<tr>
<th></th>
<th>Software, 1 core computer</th>
<th>Software, 2 core computer</th>
<th>Software, 16 node cluster</th>
<th>Hardware, Sarah Gibson</th>
<th>Hardware, This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Rate</td>
<td>0.4 MBps</td>
<td>0.9 MBps</td>
<td>2.4 MBps</td>
<td>60 MBps</td>
<td>66 MBps</td>
</tr>
<tr>
<td>Improvement</td>
<td>165x</td>
<td>73x</td>
<td>28x</td>
<td>1.1x</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 1: Performance Benchmarks**

In theory, this hardware tool is able to process data 165x more quickly than a single core computer, 73x faster than a 2 core computer, and 28x faster than a linux 16 node cluster running a software version of OSORT. It was also able to run 10% faster than the previous hardware implementation. Specifications for the computers that were tested can be found in Sarah’s paper [1].
4.2.3 FPGA Utilization

Figure 12 shows the resource utilization of the OSORT program.

<table>
<thead>
<tr>
<th>Logic utilization</th>
<th>1 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational ALUTs</td>
<td>5,246 / 424,960 (1 %)</td>
</tr>
<tr>
<td>Memory ALUTs</td>
<td>0 / 212,480 (0 %)</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>746 / 424,960 (&lt; 1 %)</td>
</tr>
</tbody>
</table>

**Figure 12: FPGA Utilization**

The total logic utilization for the OSORT block was less than 1% of the FPGA. It uses 1% of the combinational ALUTs (adaptive look up tables), no memory ALUTs, and less than 1% of the dedicated logic registers. However, this number is subject to change since the memory read and write was not included in this OSORT program for simplicity. All cluster means were kept in internal registers. However, storing the cluster means in memory will not take a significant amount of memory, so the total logic utilization will not increase by a significant amount.

The minimal use of FPGA resources leaves a large amount of room for scalability and more complex algorithms. The number of channels could easily be scaled up to 64 without utilizing all FPGA resources. This also leaves room for different clustering schemes or methods to be used if desired.

5 Future Work

The spike sorter is designed to be modular, meaning it can be changed and modified as needed. Additional work to be done on the program includes but is not limited to; more complex detection and alignment methods, different clustering methods, and improved python interpreter or a different interpreter altogether.

The spike sorting algorithm is bandlimited by the python wrapper and system buffer. They take such a large amount of time that it is the bottleneck for the data transfer. The latency of the system was reduced by 80%, but it only had a marginal effect on the overall data rate simply because the system buffering took so much time. Improving the system here could greatly increase the data rate.

The system can also be possibly adapted to real time applications. At a system clock rate of 125MHz, the latency of the FPGA processing would be approximately 10µs. If the latency of the entire system were to be reduced, the spike sorting would have the potential to be used in real time.

6 Conclusion

This paper demonstrates the ability of the PROCStar IV 530 FPGA to be used in a spike sorting application. Additionally, changes were made to the previously implemented OSORT algorithm to reduce the latency from 266 clock cycles to 55 clock cycles. This is an 80% improvement. However, the data rate is bandlimited by the python wrapper and system buffer. So a better and faster implementation of those components would lead to improved data rates.
This paper lays down the groundwork needed for future neural spike sorting applications.

References

[4] Aria Sarraf, Multi-Channel Neural Spike Detection and Alignment on GiDEL PROCStar IV 530 FPGA Platform, MS Project