Enabling a Real-Time Solution for Neuron Detection with Reconfigurable Hardware

Ben Cordes, Jennifer Dy, Miriam Leeser
Northeastern University, Boston, MA
{bcordes,jdy,mel}@ece.neu.edu

James Goebel
NeuralArts, Inc., Decatur, GA
jkgoebel@neuralarts.com

Abstract

FPGAs provide a speed advantage in processing for embedded systems, especially when processing is moved close to the sensors. Perhaps the ultimate embedded system is a neural prosthetic, where probes are inserted into the brain and recorded electrical activity is analyzed to determine which neurons have fired. In turn, this information can be used to manipulate an external device such as a robot arm or a computer mouse.

To make the detection of these signals possible, some baseline data must be processed to correlate impulses to particular neurons. One method for processing this data uses a statistical clustering algorithm called Expectation Maximization, or EM. In this paper, we examine the EM clustering algorithm, determine the most computationally intensive portion, map it onto a reconfigurable device, and show several areas of performance gain.

1. Introduction

An emerging field of research is in neural prosthetic devices, where probes are inserted into the brain, electrical activity is analyzed, and the results are used to manipulate an external device. Neural prosthetics provide promise for the physically disabled, from controlling a personal computer via a pointing device, to robot arms that can complete simple tasks. In the future, neural prosthetics may allow users to manipulate artifical limbs with a simple thought.

Neural signals are acquired as electrical impulses on a probe. In order to be useful, the impulses must be processed to determine which neurons are firing at a particular time; once a neuron or set of neurons has been isolated, the probe can detect later firings of that same neuron. The user can then be trained to fire it intentionally, providing a method of control. This isolation process is achieved using a statistical analysis algorithm called Expectation Maximization[4], or *EM clustering*. EM breaks a large number of datapoints into groups (or *clusters*), identified by mean and variance.

Software solutions for EM do not run fast enough to be used in a real-time processing environment. These solutions must record data from the probe onto a host PC and process it there; only after this processing is complete can the real work of training the user to issue control commands begin. We propose to move portions of the EM application into hardware in order to speed it up; a block diagram of our design, featuring an Annapolis MicroSystems WildCard II[2] reconfigurable platform, is shown in Figure 1. As in the software-only solution, the host computer is reponsible for reading the baseline probe data and passing it through principal component analysis (PCA). Our design performs part of the EM clustering application in hardware (see Section 4) and writes the results back to the host PC. Those results can then be fed into another part of the system which processes "live" data and extracts control signals from it.

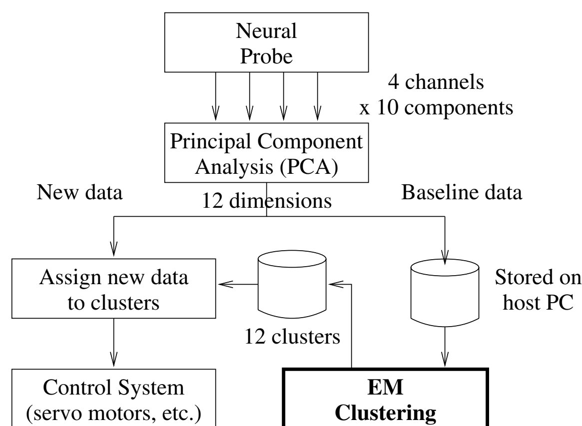

Figure 1. Block diagram of system

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

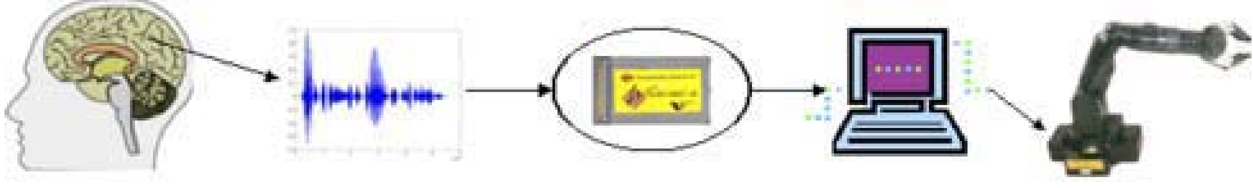

Figure 2. Neural Prosthetic System

We will examine the performance gains that are realized by this initial implementation and identify possibilities for further performance enhancement. By implementing the clustering algorithm in hardware, we not only show an improvement in the application's running time, but also take the first step towards a prosthetic system where the initial analysis is done in-line with the recording of the baseline data (see Figure 2). This hides most of the latency of the clustering application and provides the neuron isolation data more quickly. Eventually, with more hardware resources, we can design a system which performs the entire detection and classification process for both baseline and in-use data in a real-time environment, enabling the wide variety of possible applications of prosthetic devices.

The rest of this paper is organized as follows. Section 2 details the mathematical background behind EM clustering. Section 3 discusses related work in this field and draws parallels to our own efforts. Section 4 describes the parameters and tradeoffs that we examined in order to determine which portion of the computation we would consider for our initial design. In Section 5 we show the performance and error results achieved from comparing our design to a software-only solution. Section 6 highlights the directions in which the project could move forward and be expanded. Finally, we draw conclusions in Section 7.

2. Background

The goal of EM clustering is to assign each of many datapoints to a series of groups, or clusters. This is accomplished by iteratively determining the best fit of the datapoints to the current clusters (E-step), and then modifying the clusters to fit the newly grouped datapoints (M-step). Below we outline these two steps and the computation required. More detailed information is available elsewhere[4, 8, 9].

2.1. E-step

Figure 3 shows the equations that make up the E-step; here we explain the notation used. The E-step

$$Q(\Phi^{t+1}|\Phi^t) = \sum_{i=1}^{N}\sum_{j=1}^{K} E[z_{ij}] \cdot (\ln p_{ij}(y_i|\Phi_j) + \ln \pi_j) \quad (1)$$

Where

$$E[z_{ij}] = \frac{p_{ij}(y_i|\Phi_j)\pi_j{}^t}{\sum_{s=1}^{K} p_{is}(y_i|\Phi_s)\pi_s{}^t} \quad (2)$$

$$p_{ij}(y_i|\Phi_j) = 2\pi^{-\frac{D}{2}} \cdot (\prod_{k=1}^{D} \sigma_k{}^t)^{-\frac{1}{2}} \cdot e^{-\frac{1}{2}\sum_{k=1}^{D} \frac{(y_{ik}-\mu_{jk}{}^t)^2}{(\sigma_{jk}{}^t)^2}} \quad (3)$$

Figure 3. E-step Equations

takes the following inputs:

1. The dataset, a matrix y_{ik}. i varies over the number of datapoints, N. Each datapoint y_i is a vector of D dimensions.
2. A vector of mixture proportions, π_j. For each of K clusters, π_j gives the proportion of datapoints that belong to that cluster; in other words, the probability that any random datapoint belongs to that cluster. Thus, $\sum_{j=1}^{K} \pi_j = 1$.
3. The matrix z_{ij}, giving the probability that datum i belongs to cluster j. Likewise, for fixed i, $\sum_{j=1}^{K} z_{ij} = 1$.
4. A matrix of probability distribution means, μ_{jk}. For each cluster j, μ_{jk} is the mean in dimension k of the datapoints in cluster j.
5. A matrix of probability distribution covariances, σ_{jk}.

We must then compute $p_{ij}(y_i|\Phi_j)$ several times in order to find $E[z_{ij}]$, the expected new value of z_{ij}. By computing $p_{ij}(y_i|\Phi_j)$ for every current value of z_{ij}, we can compute $E[z_{ij}]$ by summing the set of all $p_{ij} \cdot \pi_j$ once for each cluster j and performing one division for each z_{ij}. The problem then becomes computing $p_{ij}(y_i|\Phi_j)$ as rapidly as possible.

Each $p_{ij}(y_i|\Phi_j)$ is a product of three factors:

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

- $2\pi^{-\frac{D}{2}}$, which is a constant
- $(\prod_{k=1}^{D} \sigma_{jk}{}^{t})^{-\frac{1}{2}}$, which can be computed once for each j (i.e. is constant across datapoints within a cluster)
- $e^{-\frac{1}{2}\sum_{k=1}^{D}\frac{(y_{ik}-\mu_{jk})^2}{(\sigma_{jk}{}^{t})^2}}$, which must be recomputed for every datapoint y_i

Computing p_{ij} for every datapoint takes $\mathcal{O}(K \cdot D \cdot N)$ operations. This inner loop consumes most of the computation time, and is therefore where we focused our efforts in Section 4.

2.2. M-step

Figure 4 shows the equations for the M-step. The M-step requires a large number of multiply-accumulate steps, followed by a few stray operations. However, it requires only the dataset y_{ik} and the probability matrix z_{ij} as inputs.

$$\pi_j{}^{t+1} = \frac{1}{N}\sum_{i=1}^{N} E[z_{ij}] \quad (4)$$

$$\mu_{jk}{}^{t+1} = \frac{1}{N\pi_j{}^{t+1}}\sum_{i=1}^{N} E[z_{ij}]y_{ik} \quad (5)$$

$$\sigma_{jk}{}^{t+1} = \frac{1}{N\pi_j{}^{t+1}}\sum_{i=1}^{N} E[z_{ij}](y_{ik} - \mu_{jk})^2 \quad (6)$$

Figure 4. M-step Equations

The computation of $\pi_j{}^{t+1}$ is a straightforward summation over a column of the z_{ij} matrix. $\mu_j k^{t+1}$ uses the same algorithm but requires multiply-accumulate steps instead of straight accumulation. $\sigma_{jk}{}^{t+1}$ is similar but requires more arithmetic before the MAC step. We employ a useful transformation, namely $N\pi_j{}^{t+1} = \sum_{i=1}^{N} E[z_{ij}]$; by sharing terms, each of these computations is $\mathcal{O}(N)$.

3. Related Work

Neural prosthetics are being widely studied in general, and in particular methods for improving the speed of data processing are of interest (see [6, 11]). Lewicki[7] provides a survey of different mechanisms for performing the initial waveform-to-neuron classification. Wood et al[13] demonstrated a method of reducing multidimensional data via PCA and then clustering it via EM, which is similar to our method.

The K-means clustering algorithm was examined and mapped to FPGAs in Estlick et al[5] and Belanović and Leeser[10]. K-means is a computationally simpler algorithm, and while it is similar to EM in its goal and method, it provides less accurate results. Our previous work on the FDTD algorithm[3] provided us with the libraries to model fixed-point calculations in C. These papers, provided useful computational shortcuts for our EM pipeline and programming tricks for implementing fixed-point routines in software.

4. Analysis

Application designers for reconfigurable hardware often stumble over the same problems time and time again. Determining what portion of the computation to move to hardware, how to get data in and out of the memories on the reconfigurable device, what format the data will take; these are common decisions that must be resolved for any application. Speed increase from hardware designs usually comes from parallelism, so determining which aspects can be parallelized, and by how much, is also a critical concern.

Our analysis for this application is based on a C implementation of EM clustering, instrumented with debugging and timing calls and parameterized to ease exploring different tradeoffs. Below we will examine each of these tradeoffs, and explain the logic behind the design that we chose.

4.1. Partitioning

Focusing on the E-step, we determined that the computation of p_{ij} took the largest amount of time by an order of magnitude. By comparison, the computation of $E[z_{ij}]$ and Q (both $\mathcal{O}(N \cdot K)$) were relatively short. This would make the p_{ij} computation the obvious target; however, as with all attempts to move software algorithms into hardware, we also had to examine the effects of data transfer between the host and device to determine its negative impact on the improved performance.

The p_{ij} computation takes the dataset y_{ik} ($N \cdot K$ elements) and the parameters π_j, μ_{jk}, and σ_{jk} ($K \cdot (2D+1)$) as its input. However, since the datapoints do not change over the course of the algorithm, they need only be uploaded to the hardware device once; the appropriate constants must be uploaded each time. The p_{ij} matrix download contains $N \cdot K$ elements.

The $E[z_{ij}]$ computation takes the p_{ij} matrix as input ($N \cdot K$) and produces an $N \cdot K$ matrix. Computing Q from that matrix produces a single output. From the data-passing perspective then, the best solution would

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

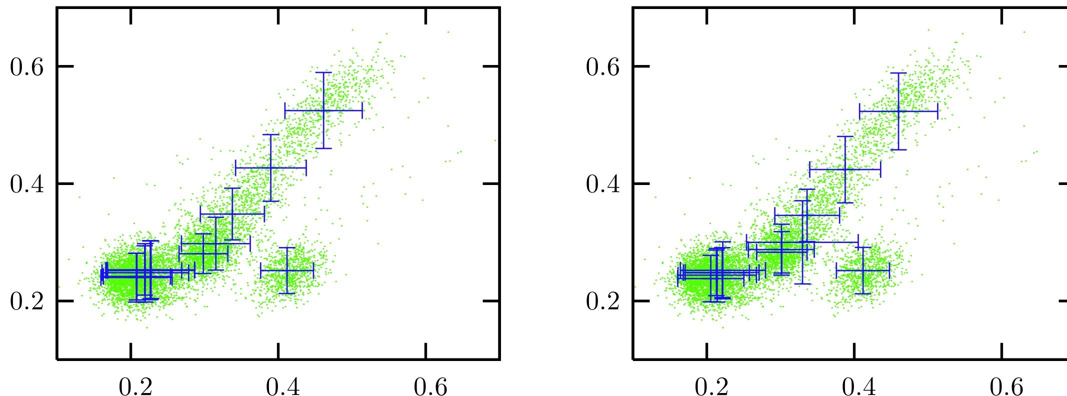

Figure 5. Clustering results, Floating-point(left) and Fixed-point(right)

be to include all three E-step computations in hardware, since it requires the least amount of input and the least amount of output.

However, it is not possible to compute p_{ij}, $E[z_{ij}]$, and Q as part of the same unified pipeline, as the results of each step must be completely computed before the next can begin. This means that including all three computations in the same hardware design significantly increases complexity as well as memory requirements, since many temporary results must be stored between pipelines. Also, each pipeline would have to run independently of the others, reducing the percentage of active logic at any one time. The WildCard board has fairly limited memory resources compared to other FPGA coprocessor boards, so we decided against designs which required a large amount of memory.

In the end we focused on a design which included only the p_{ij} computation. To save space, we will process one cluster at a time, download the results to the host, and then process the next cluster. This reduced design complexity and increased logic utilization at the potential cost of large amounts of data transfer.

4.2. Number format

Our options for representing numbers in hardware range between fixed-point and floating-point implementations, as well as the width of the numbers in memory. Fixed-point numbers have a significant advantage from the perspective of logic, since fixed-point arithmetic units are significantly smaller and easier to implement in hardware. However, a certain amount of error is introduced due to the fact that only a small range of numbers may be represented in fixed-point. Also, since the software computations will be performed in double-precision floating-point, a conversion routine must be introduced.

We modified our C code to include library functions which performed operations in fixed-point instead of floating-point in an attempt to determine the amount of error that would be introduced. Different widths of integral and fractional parts of the fixed-point number were also included as parameters. The code was run each time with the same random seed, and the final clustering results were compared. We used relative error as our criterion, defined as

$$\delta_{rel} = \left| \frac{(\Phi_{float} - \Phi_{fixed})}{\Phi_{float}} \right| \quad (7)$$

For each parameter, the best match was created between the clusters determined by the fixed- and floating-point runs. Then parameters (π_j, μ_{jk}, and σ_{jk}) were compared using relative error.

Width of the data word is an important factor in the design, because it affects not only the amount of data that must be transferred to and from the chip, but also the error inherent in the fixed-point representation. Bitwidth is also a critical concern due to the issue of storing data on-chip and retrieving it for processing by the datapath.

The WildCard system contains a 32-bit interface to an SRAM, so it makes the most sense to choose a bitwidth that is a factor of 32. The obvious choices would be 4x8, 3x10, or 2x16 bits. Scaling was used on the data to force the datapoints and parameters to be in the range 0.0 - 1.0, meaning that only fractional bits need be uploaded to the board (though full-width words must be downloaded back to the host).

The software model was run with several different bitwidths. A 32-bit representation using 16 integral

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

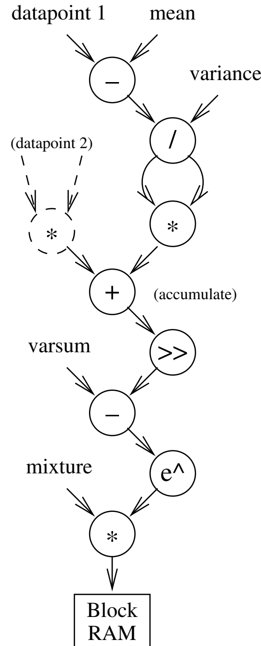

Figure 6. Dataflow diagram of p_{ij} computation

bits and 16 fractional bits yielded a δ_{rel} which averaged around 1.5% and did not exceed 5%. This was judged to be an acceptable amount of error, especially in light of the significant increases in error with smaller fractional bitwidths. Figure 5 shows sample results from the floating-point implementation and a 32-bit fixed-point implementation. The crosses represent the mean of each cluster, with the error bars representing the variance. Datapoints are shown as dots in the background. Note that this is only two dimensions out of the 12-dimensional data; clusters which appear to overlap in these two dimensions will naturally not overlap across other dimensions.

4.3. Parallelism

Usually, the speedup that comes from the hardware implementation is derived from exploiting parallelism in the software algorithm. Multiple functional units can be created to replicate operations of a loop, provided that there is enough memory bandwidth to feed data to every unit. With a 32-bit interface to memory and 16-bit datapoints (since we are only concerned with numbers in the range 0.0 - 1.0, all we need are the fractional bits of the number), two datapoints can be read from the memory per clock cycle.

Figure 6 shows a dataflow diagram of the computation involved in the p_{ij} pipeline. The topmost portion computes data for each dimension, and the output feeds into a K-wide adder tree to sum the subterms across all K dimensions. However, we implement the adder tree as an m-input adder feeding into an accumulator, since our subterms will be created over time. Since we can only read two pieces of data out of memory at a time, we can create two copies of the head of the pipeline, or $m = 2$. Therefore, the accumulator turns out a result every $K/m = 6$ cycles. Note that this is a limitation of the hardware, and not of the algorithm; with greater memory bandwidth, this number could easily be reduced.

4.4. Implementation

Once the design parameters had been explored using the C code, the portion to be moved into hardware was written in VHDL, using the templates provided by Annapolis MicroSystems. Data sent from the host arrives on the LAD bus and is channeled to the SRAM or to small register arrays. Once all parameters have been uploaded, a start signal causes the memory controller to begin pulling data out of the SRAM and feeding it to the pipeline. The final results are written into an on-chip BlockRAM, where the DMA controller reads them and ships them back to the host.

In summary, then, the hardware design has the following parameters or features. Numbers are represented with a 32-bit fixed point representation, 16 integral bits and 16 fractional bits. The SRAM contains the datapoints (y_{ik}); the uploaded datapoints and parameters have no integral bits, so they are 16 bits wide. The SRAM/FPGA interface is 32 bits wide, so two datapoints can be read per cycle. Two parallel pipelines produce one summation output every six cycles (see Section 4.3, above); these 32-bit results are held in a dual-ported BlockRAM. Datapoints are returned to the host via DMA transfer, mastered by the WildCard board.

5. Results

The hybrid system was run on a 3.2GHz Pentium desktop machine running Windows 2000 with 1GB of RAM. A PCI-to-PCMCIA card was installed to provide an interface to the Annapolis WildCard. The C code was compiled with gcc and run under the Cygwin UNIX compatibility layer. Debugging information was mostly turned off, except for the lines necessary to print the timing data. The design was simulated using ModelSim. Synthesis was handled by a toolflow included with the WildCard system, and featured the Synplify[12] synthesis package.

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

Category	Used	Avail.	
flip-flops used	5,661	28,672	19%
4-LUTs used as logic	2,943	28,672	10%
4-LUTs used as routing	143		
4-LUTs used as ROMs	784		
occupied slices	5,747	14,336	40%
IOs used	110	484	22%
BlockRAMs instantiated	2	96	2%
Multipliers instantiated	10	96	10%

Table 1. Device Utilization

Timing numbers were acquired by inserting for-loops around blocks of the code, running the code many times (10,000 or more) and then dividing to reach an average runtime result. The hardware is started and stopped by API calls, so wrapping these in a for-loop was also trivial. Due to the accuracy of the C system calls employed in the instrumentation, the timing numbers are accurate to two significant digits.

The pure-software system (using double-precision floating-point numbers) took 470us to run the portion of the E-step that we focused on, i.e. the computation of the p_{ij} matrix. Our hybrid system, with the p_{ij} computation moved into hardware, took only 140us. Not surprisingly, transferring the data from the WildCard to the host required 130us, or around 96% of the total processing time. These results show the following speedup over the software-only solution:

$$\Delta = \frac{t_{sw}}{t_{hyb}} = \frac{470us}{140us} \approx 3.4$$

Despite the modest speedup, this is a good result in that it shows that even this very limited implementation of the EM clustering algorithm can result in a significantly faster runtime. There are many places where further speedup can be found; a list of suggestions can be found in Section 6.

Lastly, Table 1 shows statistics from the synthesis process related to the utilization of the FPGA device on the WildCard. Only around one-third of the available resources were used; as shown in Section 4.2, this is largely due to a memory bottleneck inherent in the reconfigurable platform used. With more memory devices available, a wider pipeline could be implemented, and more logic resources could be exploited.

6. Future Work

Our project focused primarily on proof-of-concept, and an exploration of the effectiveness of moving portions of this algorithm into hardware. Figure 7 shows a block diagram of the current system, with the probe data first downloaded to the host, then transferred to and processed on the WildCard. Looking forward, a neural-signal processing system could include a similar device between the probe and the controlling host, removing the host entirely from the initial clustering process.

Furthermore, additional hardware could be used to take the clustering results and apply them to new, incoming data. By inserting the hardware into the data processing stream, the time cost of data transfer is amortized and further speedups are possible. In this conceptual stage, however, the data that was processed had already been collected and stored; our hardware processed the data after the collection and provided a result to compare against the software-only solution.

Since our goal was to create a simple implementation of the algorithm, there were many opportunites to increase efficiency and speed that were identified but not exploited. What follows is a brief exploration of these areas and suggestions for future improvements.

First, we did not take advantage of all the resources available to us on the WildCard board. The synthesized VHDL design used only around 30% of the available resources, so a significant portion is still available. While memory bandwidth was a limiting factor in our design decisions, it can be alleviated to a certain degree with the extra space. For instance, input data could be sent to a parallel array of on-chip BlockRAMs instead of the on-board SRAM; this could potentially result in larger memory bandwidth to the head of the pipeline, allowing further logic usage and pipeline parallelism.

Likewise, with a sufficiently large FPGA system (one with both more logic capacity and more memory bandwidth), it would be possible to rethink the tradeoffs mentioned in Section 3. More of the E-step could be moved onto the hardware, or more copies of the current pipeline portion could be implemented. Eventually with a large enough system the process could be moved entirely into hardware, further reducing necessary data transfer.

Converting the results from the WildCard from

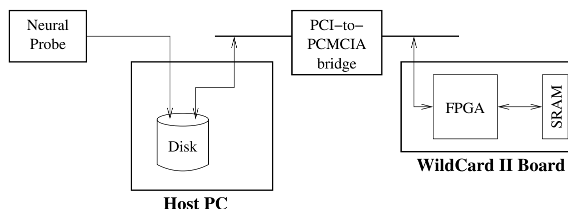

Figure 7. Block diagram of system

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

fixed-point to floating-point was done inefficiently and in software, resulting in significant performance reduction. While the timing results presented above do not include this performance loss, extra available logic could be used to perform this computation in hardware, almost completely negating the lost time. Alternately, the hardware could be modified to perform floating-point computation instead of fixed-point. This would remove the introduced error and eliminate the conversion routine, but would increase complexity.

As in most hardware solutions, data transfer uses up a large amount of time which could otherwise be spent processing. In this case, we have employed the fastest available mechanism (DMA transfer) for the largest transfer which happens most often, that is, the download of the p_{ij} matrix. DMA could also be employed during datapoint upload, and potentially during parameter upload as well, although these gains may not be as significant.

Recently, FPGA chips with embedded general-purpose processors in them (Altera's Excalibur[1] and Xilinx's Virtex-II Pro[14], for example) have been developed and are beginning to gain popularity. Our hybrid solution would map well onto one of these devices; it would benefit greatly from the close connection of the processor and the FPGA, reducing data transfer times. The integrated CPU could also take on some of the less time-critical control logic (like loop counters), leaving more room on the FPGA for datapath logic.

7. Conclusions

Neural prosthetic devices are an exciting field of research with many possible beneficial applications. Control systems based on neural signal processing could have far-reaching effects on every aspect of life, from mundane tasks like driving a car, to highly specialized applications such as robotic manipulation of dangerous materials. Improving the speed at which these devices can be "trained" is a critical aspect of this research, as it affects the usefulness and can lower the bar of technical knowledge required to use such a device.

We have created a hybrid host/FPGA solution that offloads some of the expensive computation of this training process into hardware. This results in a modest amount of speedup due to the limitations of the specific platform chosen. However, when this design is implemented on a larger FPGA or one with an embedded processor, there are ample opportunities for significantly larger gains. We conclude that this algorithm benefits from implementation with a hybrid software/hardware solution, and opens up a new set of potential applications for reconfigurable hardware.

References

[1] Altera, Inc., San Jose, CA. *Excalibur Embedded Processor Solutions.* `http://www.altera.com/products/devices/excalibur/exc-index.html`.

[2] Annapolis Microsystems, Inc., Annapolis, MD. *WildCard II Data Sheet.* `http://www.annapmicro.com/wildcard2.html`.

[3] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport. An FPGA Impelementation of the Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm. In *ACM/SIGDA Field-Programmable Gate Arrays (FPGA)*, pages 213–222, Monterey, CA, February 2004.

[4] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[5] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski. Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware. In *ACM/SIGDA Field-Programmable Gate Arrays (FPGA)*, pages 103–110, 2001.

[6] C. Kemere, K. V. Shenoy, and T. H. Meng. Model-based neural decoding of reaching movements: a maximum likelihood approach. *IEEE Transactions on Biomedical Engineering*, 51:925–932, 2004.

[7] M. S. Lewicki. A review of methods for spike sorting: the detection and classification of neural action potentials. *Network: Comput. Neural Syst.*, 9(4):R53–R78, November 1998.

[8] G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions.* John Wiley, New York, 1997.

[9] G. J. McLachlan and K. E. Basford. *Mixture Models, Inference and Applications to Clustering.* Marcel Dekker, New York, 1988.

[10] P. Belanović and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. In *Field Programmable Logic and Applications (FPL)*, pages 657–666, Montpellier, France, September 2002.

[11] R E Isaacs and D J Weber and A B Schwartz. Work toward real-time control of a cortical neural prosthesis. *IEEE Trans. on Rehabilitation Engineering*, 8(2):196–198, June 2000.

[12] Synplicity, Inc., Sunnyvale, CA. *Synplify Pro.* `http://www.synplicity.com/products/synplifypro`.

[13] F. D. Wood, M. Fellows, J. P. Donoghue, and M. J. Black. Automatic Spike Sorting for Neural Decoding. In *Proc. IEEE Engineering in Medicine and Biology Society*, pages 4009–4012, September 2004.

[14] Xilinx, Inc., San Jose, CA. *Virtex-II Pro Platform FPGAs.* `http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs`.

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY