A Framework for Assessing the Feasibility of Learning Algorithms in Power-Constrained ASICs

Alexander Neckar
with David Gal, Eric Glass, and Matt Murray (from EE382a)

1 Introduction

Whether due to injury or neurodegenerative disease, millions of paraplegics suffer from an inability to actuate their limbs despite functional thought processes. Neuroscientists have developed methods [1] of monitoring and interpreting neural activity. By surgically implanting an electrode array in the motor cortex of the brain, scientists can tap into the real-time neural activity of an individual. These neural observations can be interpreted as an intended actuation which can then be realized on behalf of the individual. A paraplegic, once lacking the agency to perform simple physical tasks, can now do so by thought alone; a revolutionary improvement in quality of life. Current neural decoding implementations render the patient tethered to a cart of computers with visible wires emerging from his or her head. Collection of neural activity occurs beneath the skullcap while processing is done remotely. There is much room for improvement on this setup; patients are embarrassed by the physical appearance of the apparatus and can only utilize it in the controlled environment of a university laboratory.

The natural solution to this problem is to embed the processing that currently takes place remotely into the same physical space as the data collection - an embedded processor implanted in the brain. To date, however, no such system has been designed. It is not immediately obvious that such a system could exist. The environment of the human brain presents design challenges and specifications not typically present in traditional computing systems. Primarily, any implanted system may not raise the temperature of brain tissue by more than 1º C. This implies a low power design. In addition, the complete system area is constrained to a small physical space, ideally less than the size of the existing 4 mm x 4mm electrode grid.

As a group, we explore the requisite hardware resources necessary to implement the dynamic portion of the current neural-decoding algorithms within the confines of a human brain. We develop a framework with which to test the feasibility of such a design and conclude that such a device is indeed possible within the design constraints. Individually, I then go on to extend our evaluation framework to apply to any learning algorithm written in C or Matlab. I use this framework to assess the feasibility of implementing the static regression portion of the decoding algorithm on-chip. Finally, I demonstrate the flexibility of the framework by assessing the performance of a different learning algorithm to the neural decoding problem.

2 Neural Decoding Algorithm: Kalman Filter

To implement neural decoding, a Kalman filter is used. This is necessary due to the large amounts of noise inherent in the collection of neural observations. The system is modeled with linear dynamics. There is a linear relationship between kinematic state of the cursor (x,y,z position and velocity) and neural observations. Noise and uncertainty is Gaussian. This can be summarized as follows:

$$x_{t+1} = Ax_t + w_t, \qquad y_t = Cx_t + v_t$$

Where *x* is the kinematic state of the cursor, *y* is neural observations, and *w* and *v* are Gaussian noise sources. The training sets are collected as follows: A monkey with the electrode grid planted in its motor cortex is fitted with a special glove. This glove controls a cursor on a 3D screen. The monkey is given targets on the screen. It moves the cursor to these targets in exchange for a reward. Each “reach” test takes on the order of one second. Every 50 ms, the cursor's kinematic state is measured and recorded along with the number of neural spikes

observed on each individual electrode over the course of the timestep. Each element of the training set thus consists of the following:

- A 96-element neural observation vector containing the binned spike count for the last 50 ms on a single electrode (there are 96 electrodes in the grid)
- A 7-element vector that describes the cursor's x and y position, velocity, and acceleration as well as the intercept term

The algorithm that is implemented is a Kalman filter with some special modifications [1]. The algorithm essentially has two phases: The initial regression phase or “static phase”, where observations are regressed on cursor state and future cursor state is regressed on present cursor state, and the Kalman update phase, where, given a new neural observation, the likely cursor state is predicted and the present uncertainty in position is updated and the time update is performed. The basic Kalman updates to cursor state and uncertainty and the time updates, respectively, are as follows:

$$\begin{aligned}\hat{x}_{t|t} &= \hat{x}_{t|t-1} + \Sigma_{t|t-1}C^T\left(C\Sigma_{t|t-1}C^T + V\right)^{-1}\left(y_t - C\hat{x}_{t|t-1}\right)\\ \Sigma_{t|t} &= \Sigma_{t|t-1} - \Sigma_{t|t-1}C^T\left(C\Sigma_{t|t-1}C^T + V\right)^{-1}C\Sigma_{t|t-1}\end{aligned}$$

$$\hat{x}_{t+1|t} = A\hat{x}_{t|t}, \qquad \Sigma_{t+1|t} = A\Sigma_{t|t}A^T + W$$

3 Framework for Analysis of ASIC Implementations of Learning Algorithms

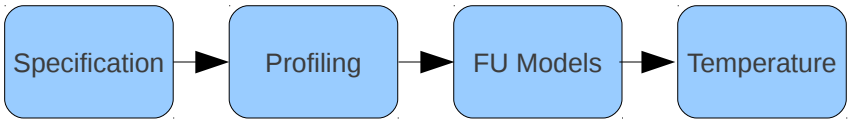

Fig. 1: ASIC analysis Framework

For the purposes of the original project, we focused on the implementation of the more time-critical and necessarily on-chip dynamic phase of the algorithm. We envision that initial implementations of this device will not include hardware to perform static calculations. The static regression phase is very computationally heavyweight compared to the dynamic Kalman update, and since the current framework for training set-collection requires patient proximity to computers anyway, it seems reasonable to have those computers perform the regression (where computing resources are much cheaper) and pass only the static variables needed by the Kalman update to the device.

After studying the execution profile of the Kalman update step and making a full assessment of its computational demands, we envisioned a very simple ASIC device upon which to implement the algorithm, composed of a single execution unit that carries out the necessary floating-point math and a single memory bank in which to hold the data. Control circuitry is assumed to introduce an overhead which does not dominate the total power consumption.

We then used tools to generate operational power and area models for each of these components. Each component is abstracted as having three parameters: static leakage power, per-access energy, and area. By inputting the profile obtained for the update, from these models we were able to derive a power and area model for our ASIC.

Using this area and this total chip power, we used a thermal simulator to study the effect of the device implementing the algorithm on brain temperature, determining the feasibility of the design. In our original study, we found that the Kalman update is easily implemented within the constraints, with the device causing only a 0.04º C increase in brain temperature. Further, we derived the maximum power consumption budget of such a

device to be roughly 12 mW.

I extended our framework to allow for any Matlab or C program to be easily instrumented to obtain the necessary figures and partially integrated some layers of the existing framework, essentially making the framework much more portable and useful to a first-time user. I describe the overall framework, starting with the original components used in our initial study:

3.1 Memory System Simulation: CACTI

In order to generate a power and area model for our memory system, we used the CACTI simulator [5]. While originally only for modeling caches, CACTI is now well-suited to model a variety of memory architectures. For our purposes, CACTI's most important output are the per-access energy, leakage power, and area of the memory. Given the frequency of memory access obtained from the profiled code, this allows us to calculate a power consumption for our memory. Additionally, CACTI contains several transistor models and allows the user to vary parameters such as feature size, enabling a thorough exploration of the design space. Given the most attractive choices of low-power DRAM, low standby-power SRAM, and low operation-power SRAM, low standby-power SRAM proves to be the best-performing in our application.

3.2 Floating-Point Unit Simulation: McPAT

Just as we used CACTI to simulate the memory system, we used McPAT [3] to model the FPU. McPAT is a powerful tool with many features which we did not explore. It allows the user to specify a complete multicore chip in great detail and returns power and area figures for many different subsystems. In our case, we only use it to obtain leakage power, per-operation energy, and area for the FPU. McPAT would also help to pin down the number of FPUs that would be required by the design. Like CACTI, McPAT allows the user to vary feature size and memory technology. Analogous to what we found for our memory system, LSTP transistors prove themselves be optimal for the FPU.

McPAT also provides a rough basis for determining the number of FPUs needed by a given execution profile and time bound. While the timing model for the FPU is not detailed, the one modeled is a 20-stage pipelined design capable of operating at at least 1 GHz. Given a time bound and a factor of algorithm parallelism, the number of FPUs required can be estimated.

3.3 Hotspot Thermal Model

To simulate the thermal behavior of our system, we turned to HotSpot [2][4]. Hotspot allows the user to generate a detailed thermal model of an integrated circuit, given a floorplan divided into functional blocks and power consumption values for those blocks. Normally, Hotspot is used to obtain a very fine model of on-die temperatures. However, the specification of our device places no demands on the temperature of the die itself, only on the surrounding tissue. While Hotspot holds ambient temperature as a fixed constant, through clever modification of the tool, we are able to model the temperature of the tissue adjacent to the device.

3.4 Extensions to the Framework

In order to assess the feasibility of an ASIC implementation of a general learning algorithm, I modified our original non-portable profiling techniques to allow for profiling on a variety of platforms and to function as a simple wrapper to existing code written in C or Matlab. To this end, I selected the PAPI [6] suite as the primary profiler. PAPI provides an easy-to-use API to access hardware counters during program execution. I make use of the floating point operation and memory reference counters. PAPI works on most current CPU architectures can be integrated with C or Matlab programs.

Additionally, I integrated several layers of the tool. After applying the customized PAPI wrapper to the algorithm to be assessed and specifying an time constraint on the algorithm, a script is run that automatically passes the results of the profiling to the necessary functional models, which generate a power and are model. If the device is meant for implantation in the brain, the power and area model can be passed to the thermal model as well. In order to determine the memory footprint, necessary for the power and area model, the user must first profile the code in either Valgrind or using the Matlab memory profiler and pass this parameter when the script is called.

4 Application of the Framework

As it is possible to envision future calibration scenarios that do not require the recipient of the device to interface with outside computers, and given the greatly improved power of the framework, I then focus on the aspects of the existing algorithm that we did not examine in great depth. In particular, I examine whether the cost of performing the regression phase can be minimized to an acceptable level such that it may be performed on-chip. I then offer further demonstration of the framework's power in assessing the possibility of alternatives to the Kalman filter itself.

Using the framework to evaluate the current implementation of the static phase, and focusing on very expensive regression of observations on cursor state, I find that for the complete training set of 19772 points, LMS regression by solution of the normal equations demands about 17.5 million FPU operations and 24 million memory references. For comparison, each update of the Kalman filter only contains 27,000 FP operations and 38,000 memory references. For the per-access energies and leakages of the FPU and memory units that I derived, this implies that the regression is indeed possible on-chip provided that the computation is spread out over more than .911 seconds. This, however, represents a power consumption 70 times in excess of what is required to do Kalman updates. If this factor is reduced to 10, about 30 seconds are required to do computation.

4.1 Attempted Optimization with Stochastic Gradient Descent

Initially, I attempt to achieve a lower power consumption via use of stochastic gradient descent to perform the regression. SGD seemed an appealing choice due to the small cost of a single update to the hypothesis. Indeed, the number of FPU ops required is only about 750,000. If SGD could converge within 23 iterations, it could possibly outperform the solution of the normal equations. This, however, proved to be unrealistic. The training data proved to not be so weakly linear that convergence cannot occur in an efficient manner.

4.2 Batch Gradient Descent:

Given the failure of stochastic gradient descent, I analyze batch gradient descent as well. However, I find that a full sweep of the training set costs 224 million FPU operations. Unlike for SGD, BGD converges on this training set but only with very careful manipulation of the learning rate and a huge number of iterations. Consequently, BGD's energy demands are in great excess of those of the normal equations.

4.3 Alternatives to the Kalman filter

As an experiment testing the framework, I attempt assess the power demands of a direct regression of cursor state on neural observation (opposite to what the static phase of the Kalman filter demands). Given weak linearity of the training set, I choose to evaluate an algorithm which captures additional patterns in the data, locally-weighted linear regression. I find that even for a greatly reduced training set, a single state prediction for LWLR takes more than a billion operations, and corresponds to a huge power consumption, provided that one prediction must be made every 50 ms.

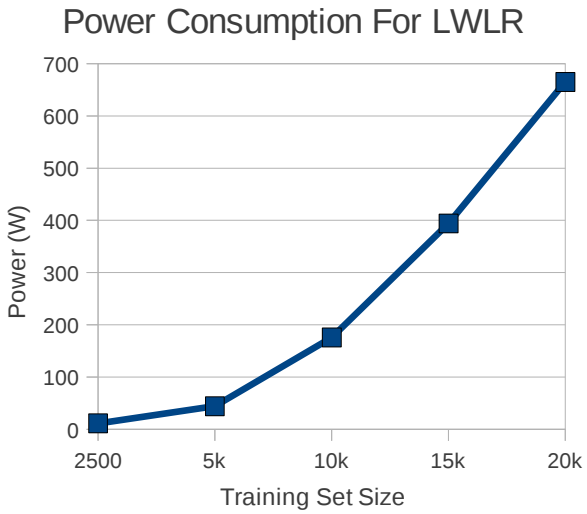

Figure 2: Power consumption for LWLR across various training set subsets, 45 nm technology

5 Future Work

While the integrated framework is already a useful tool, some additional work could be done to make its use even simpler. Primarily, the hand-profiling to assess the memory footprint of the algorithm could be integrated. With regards to the neural-decoding problem examined here, additional regression techniques could be applied and evaluated with the framework.

6 Conclusion

In increasing the power, ease-of-use, and portability of our original framework. I have created a useful tool for those seeking to implement a learning algorithm on an ASIC with a power-constrained application. Given the increasing deployment of embedded systems, and the large class of biomedical, battery operated, and other low-power applications that can be imagined, this could be an extremely useful tool for those doing initial feasibility studies for such a device. Using the original form of this framework we have already shown that a brain-implanted neural-decoding device is possible, and I have demonstrated how the expanded framework may be applied to assess additional learning algorithms.

7 Acknowledgements

I would like to thank Vikash Gilja for his support and energy in allowing us to pursue this project. Additional thanks to Christos Kozyrakis for suggesting and advising the original project.

References

[1] V. Gilja. Towards Clinically Viable Neural Prosthetic Systems. PhD thesis, Stanford Unviersity, 2010.
[2] W. Huang, S. Ghosh, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: Thermal modeling for cmos vlsi systems. In IEEE Transactions on Component Packaging and Manufacturing Technology, 2009.
[3] L. Sheng. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In MICRO-42 (2009), pages 469–480, 2009.
[4] K. Skadron, K. Sankaranarayanan, S. Velusamy, D. Tarjan, M. Stan, and W. Huang. Temperature-aware microarchitecture: Modeling and implementation. In ACM Transactions on Architecture and Code Optimization, pages 94–125, March 2004.
[5] S. Wilton. Cacti: an enhanced cache access and cycle time model. In IEEE Journal of Solid State Circuits. 31.5 (1996), pages 677–688, 1998.
[6] Moore, S.; Terpstra, D.; London, K.; Mucci, P.; Teller, P.; Salayandia, L.; Bayona, A.; Nieto, M.; , "PAPI deployment, evaluation, and extensions," *User Group Conference, 2003. Proceedings* , vol., no., pp. 349- 353, 9-13 June 2003