HiPNeT-1: A Highly Pipelined Architecture for Neural Network Training

Krste Asanović [*] [†]
Brian E. D. Kingsbury [*] [†]
Nelson Morgan [*]
John Wawrzynek [†]

TR-91-035

October 1991

Abstract

Current artificial neural network (ANN) algorithms require extensive computational resources. However, they exhibit massive fine-grained parallelism and require only moderate arithmetic precision. These properties make possible custom VLSI implementations for high performance, low cost systems. This paper describes one such system, a special purpose digital VLSI architecture to implement neural network training in a speech recognition application.

The network algorithm has a number of atypical features. These include: shared weights, sparse activation, binary inputs, and a serial training input stream. The architecture illustrates a number of design techniques to exploit these algorithm-specific features. The result is a highly pipelined system which sustains a learning rate of one pattern per clock cycle. At a clock rate of 20MHz each "neuron" site performs 200 million connection updates per second. Multiple such neurons can be integrated onto a modestly sized VLSI die.

[*]International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704
[†]Computer Science Division, EECS Department, University of California at Berkeley, Berkeley, CA 94720

1 Introduction

Many neural network algorithms, including the popular error back-propagation algorithm, require a range of arithmetic precisions for the variables used. Although many of the proposed analog neural network implementations promise excellent performance in certain applications [Mea89, HTCB89], their relatively low precision limits their general applicability. Further, in general an artificial neural network will form part of a larger information processing system and so ease of system integration will be an important design criterion. For these and other reasons [Mor90], our group at ICSI is concentrating on the development of purely digital VLSI connectionist hardware.

Part of our research is the investigation of general architectural techniques which may be applied to such systems. We are also working on the development of CAD tools and library cells to support rapid VLSI layout generation [MAKW90]. A number of example systems are being implemented to guide our work. The first of these is an architecture to perform neural network training for a speech recognition system being developed at ICSI.

In this paper, we first describe the speech recognition application. We next explain modifications made at the algorithmic level to enable efficient implementation of the network training algorithm in digital VLSI. We then describe two alternative architectures to implement these algorithms. The first is a relatively low-cost solution, which makes one forward connection per cycle, but which takes 3 cycles for a complete connection update. By examining ways to improve the throughput of this system we arrive at a second highly pipelined system which learns at the rate of one pattern per clock cycle; each pattern requiring 10 connection updates per neuron in our algorithm. We compare the performance/cost of these two architectures with that of alternative systems. Finally we present our conclusions and the current status of the work.

2 Artificial Neural Network Algorithms

At ICSI we are developing a phoneme-based, speaker-dependent continuous speech recognition system. The system utilizes a layered ANN to generate emission probabilities for a hidden Markov model (HMM) recognizer. We have shown that this method is an effective way to smoothly estimate joint densities with a number of training samples that is insufficient for simple histogram-based techniques. The ANN is capable of performing statistical pattern recognition over an undersampled pattern space without many restrictive simplifying assumptions. The ANN can also combine multiple sources of evidence, such as multiple features and contextual windows, in a straightforward and efficient manner. Initial experiments indicate that this method compares favorably with conventional HMM speech recognition methods [MB90].

In this system, continuous features (such as spectra) are extracted from speech input and passed to a vector quantizer that maps the input speech frame into one of a set of prototype vectors or features. The ANN is fed a sequence of vector quantized frames that provide a sliding window into the input speech stream. The ANN uses this contextual information to recognize phonemes and/or generate the probability of each phoneme given the input.

Each vector quantized frame is represented using unary encoding, that is, using a binary input neuron for each possible feature value, only one of which can be active at a time. For the networks described in [MB90], the vector quantizer selects one from 132 features and the ANN is fed a window of 9 frames. This gives $132 \times 9 = 1188$ input layer neurons, of which only 9 will be active in a given pattern. Additionally, the bias is treated as an extra input neuron that is always on, so that there are a total of 10 active inputs per speech pattern. The output layer consists of 50–64 neurons, corresponding to the number of phonemes to be recognized. The best experimental results for this problem were obtained without hidden units, and so the input layer is directly and fully connected to the output layer. Error back-propagation training is used [Wer74, RHW86].

The network algorithms are summarized in the following equations. Input neuron i is connected to output neuron j by a weight w_{ij}. The output of neuron j is given by

$$o_j = f(s_j)$$

where

$$s_j = \sum_i o_i w_{ij}$$

and

$$f(x) = \frac{1}{1 + e^{-x}}$$

The training algorithm derives a weight update value Δw_{ij} using a cross entropy error criterion [Hin87].

$$\Delta w_{ij} = -\alpha(o_j - d_j)o_i$$

where d_j is the desired output value for neuron j, and α is the learning rate.

Figure 1 illustrates the network architecture.

Figure 1: Speech ANN Architecture

3 Algorithm Modifications for Digital Hardware Implementation

We require no multiplications, since input values are either 1 or 0. Within each frame only a single input is active. We can split weight memory into 9 banks, one per frame. The sum can then be efficiently computed as

$$s_j = \sum_k W(k, v(k))$$

where $W(k, v)$ selects weight v in weight bank k, and $v(k)$ is the feature recognized in input frame k. For each neuron, we require only 9 additions to sum the bias and the 9 connection weights for each input pattern. In effect we have introduced a level of indirection to implement the sparse activation. We broadcast only the address of the active input; since the inputs are binary there is no need to broadcast a corresponding value.

Some modifications were made to the weight storage. The 9 input frames presented to the network are grouped into three groups of three corresponding to past, present, and future frames. Within each group, the three weights corresponding to a given feature vector are tied together so that any update affects all three. This modification was originally added to improve generalization in the speech recognition system, but has the additional advantage of reducing the weight storage required in an implementation by a factor of three. The three frames within each group share a single bank of weight memory. An additional minor change is to restrict the number of features to 127 (plus 1 bias value) since powers of 2 are more natural in the memory design. This reduction of the vector quantization codebook by only 5 entries should have little effect on performance, which has been found in previous experiments to be fairly insensitive to codebook size over a larger range.

Note that o_i is 0 for inactive inputs so the corresponding weights w_{ij} need no updating during training. For the active inputs, the value of Δw_{ij} is $-\alpha(o_j - d_j)$ which is independent of i. Hence, only a single update value is needed for each neuron per pattern. For each neuron we require 10 additions to perform weight updates, 9 for the weights and one for the bias.

One of the most important modifications to the original algorithms is to operate with only the minimum required arithmetic precision. Reducing precision decreases weight storage requirements as well as reducing datapath circuitry. Our simulations indicate that a weight precision of 12–16 bits is sufficient for learning our speech task with back-propagation. Output values require 6–8 bits. These findings are consistent with those discovered by other researchers working with back propagation [BH88]. The systems we describe here use 12 bit weights and 6 bit output values. We use only the most significant 6 bits of the stored weight in calculating the output value, and restrict weight increments Δw_{ij} to 6 bits. The total weight storage per neuron is $3 \times 127 \times 12 = 4572$ bits (not including the bias value).

With no performance degradation, α can be restricted to negative powers of 2. This replaces the learning rate multiplication with an arithmetic shift.

Finally, since the network contains no hidden layer there is no interaction between output neurons and so each neuron can be trained independently. This allows an implementation to provide only a few physical neuron processors which are then multiplexed over the complete network.

The above modifications were simulated on a Sun Sparcstation-1. For a German language test set of 200 sentences, performance matched our previously reported results to within a few tenths of a percent. Thus it appears that a fixed-point algorithm with the required simplifications for efficient VLSI implementation works essentially as well as the original floating-point version.

4 System Architecture

Both architectures described here share common features at the system level. A complete system will typically consist of a small array of chips connected to an array controller, with each chip containing a number of processing nodes. The array controller is a simple microsequencer which will contain storage for microcode and training data. The array operates in a synchronous SIMD fashion with instructions and training data broadcast from the controller. In a research environment, the array controller will provide an interface to a host workstation to allow training data and programs to be downloaded, and to allow weight values to be initialized, trained, and subsequently read.

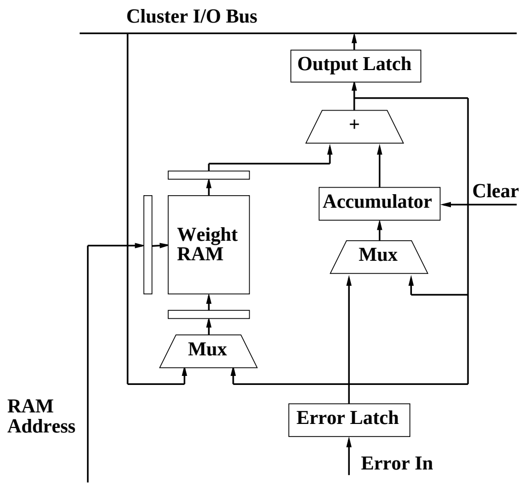

Figure 3: Synapse Unit Datapath

5 Simple Architecture

The first architecture uses a relatively simple datapath to implement the training algorithm. We include this architecture in the paper to help explain and evaluate the enhancements present in the highly pipelined architecture. The performance of this architecture is also indicative of what would be expected for more general purpose "neurocomputers".

In the simple architecture, calculations for each neuron are split between a synapse unit (SU) and an output processor (OP). The SU contains the weight and bias storage for a single neuron. The OP performs sigmoid and error calculations and is time-multiplexed between a small *cluster* of SUs as shown in Figure 2. Each chip contains an integral number of clusters. When training the network, each SU calculates $s_j = \sum_k W(k, v(k))$ and passes this value to the OP across the cluster I/O bus. The OP calculates $o_j = f(s_j)$, and forms the error term Δw_{ij} which is fed back to the SU across the error bus. The SU can then perform weight updates with this error value. The cluster I/O bus is also used to communicate weight values between the host and SU weight RAM over a global I/O bus.

Within each SU there are 384 words of 12 bit static RAM. This corresponds to $3 \times 127 = 381$ words for the three banks of shared weight storage, with a further word for the neuron bias value (the two remaining words are unused). The RAM is addressed with $2 + 7 = 9$ lines. The two most significant lines select one of the three banks of weights or the bias value, the lower lines select a weight value within a weight bank. The array controller broadcasts RAM addresses corresponding to feature numbers.

Figure 3 shows the SU datapath which includes a 14-bit saturating adder (i. e. an adder with overflow and underflow sums clamped to the extreme values), and a single accumulator register. In forward mode, the accumulator builds up s_j with weight and bias values read from memory. The sum is then stored in the output latch for further processing by the OP. When performing weight updates, the Δw_{ij} value is read from the error latch into the accumulator. Weight values are then read from RAM, added to the error value and written back to memory. The SU requires one cycle for forward connections, and two cycles for weight updates. The number of cycles corresponds to the number of RAM accesses required.

The OP datapath consists of a PLA to implement the sigmoid function f. Only the 6 most significant bits of the output o_j are used in calculating the sigmoid. The logic to implement the error calculation $o_j - d_j$ is folded into the PLA, with an extra single bit input added to indicate the desired output. This value d_j is derived by decoding the output neuron number given in the training data broadcast by the array controller. The output of the PLA is the error $o_j - d_j$. This is fed to a short sign-extending shifter which implements the learning constant α. The OP can process one sum per cycle, but each SU takes multiple cycles to form s_j and to update weights. To improve efficiency, a single OP is shared amongst 8 SUs. To increase performance, error calculations are overlapped with summations and weight updates.

Figure 4 shows the pipelining scheme in the simple architecture. The pipeline operates in two basic modes: forward mode when calculating s_j, and update mode when performing weight updates. The forward pipeline is straightforward; weight values are read from memory in one cycle and added to the accumulator in the next. In Figure 4 we show only 4 features per pattern to reduce the size of the diagram, but for the speech algorithm there will be 9 features plus 1 bias value per pattern. One cycle per weight is required to perform the forward summation.

Figure 4: Pipelining in Simple Architecture

In update mode, the value of Δw_{ij} is read from the error input latch and stored in the accumulator. Each weight is read out from the RAM, added to the update, and written back to RAM. To improve efficiency pairs of updates are interleaved in the pipeline. While the adder is summing the first weight and update, the second weight is read from RAM. The updated first weight is then written back to RAM while the second weight is added to the update. Finally, the second update is written back. Overall, it takes two cycles per weight to perform the update.

A further level of pipeline interleaving is used to overlap error calculations in the OP with weight calculations in the SUs. At the end of the first forward summation cycle, S_1 is latched into the output registers of each SU (S_n is s_j for training pattern n). The OP then sequentially scans these values and writes back Δ_1 to each SU (Δ_n is Δw_{ij} for training pattern n). Concurrently, the SUs form S_2. At the end of the second summation cycle, the SUs switch to update mode, reading in Δ_1 before starting to perform weight updates for the first training pattern. The OP concurrently performs error calculations for the second training pattern, writing Δ_2 back into the SU input registers. After the SUs finish performing weight updates for the first pattern, they read in Δ_2 from the input register and perform weight updates for the second training pattern. The OP is idle during this phase. After performing weight updates for the second pattern, the SUs switch to forming summations for the third pattern. The overall pattern of activity repeats at this point.

5.1 Pipeline Hazards

This pipelining scheme is reasonably effective at keeping the functional units busy, with each SU performing one connection update every three cycles. However this pipelining scheme does introduce a pipeline hazard. In the example above, there is a Read after Write (RAW) hazard if S_2 is computed using weight values that have yet to be updated with Δ_1. This hazard is unlikely to cause a problem with back-propagation learning, since delaying updates in this way is similar to using pooled updates over a few patterns. Our simulations indicate that ignoring this hazard has little effect on learning performance in our application.

6 Pipelined Architecture

Whereas the simple architecture requires 30 cycles to learn one input pattern, the second architecture learns one pattern per cycle. This performance increase is achieved by adding extra hardware, utilizing more extensive pipelining, and exploiting a greater degree of specialization.

6.1 Improving Memory Bandwidth

Memory bandwidth is a major bottleneck in the first architecture. The adders in the SUs are idle for half the cycles in the update phase, since each update requires two RAM accesses but only one addition. Overall, this means the adders are idle for 33% of the total training time.

The RAM only supports one access per cycle. We can save a third of this bandwidth if we note that during training, the same weight value is read out once to form s_j then again to form $w_{ij}+\Delta w_{ij}$. We can read the value out once and store it locally until the error value returns from the output processor. If we make the pipeline delay between the summation and update passes constant, we can use a shift register to provide this temporary storage.

We can further improve memory bandwidth by physically splitting the RAM into the three banks corresponding to past, present, and future frames. Bandwidth is improved both due to a wider datapath, and also due to the individual RAMs being faster than the larger combined RAM. In our design the smaller RAMs can now support a read and a write access every cycle. The read access is used to obtain a weight for forward summation, while the write is used to store an updated weight.

6.2 Increasing Training Input Bandwidth

The features in a training pattern are used to address the weight RAM. In the simple architecture, each input pattern requires that a feature address be broadcast 3 times: once to read out the weight for forward summation, once to read the weight to add an update, and once to write back the updated weight. We have already shown that the second RAM access can be removed by storing the read weight value in a shift register. We can remove the need to broadcast the third address by storing the address used to read out the weight in another shift register until the updated weight needs to be written back.

We can further reduce the amount of training data that is broadcast by training with a sequential speech input stream, as opposed to randomly selecting subsequences from the speech stream. For some problems, this could be undesirable, since sequences of repeated features may cause the network to overlearn in a certain direction. Our simulations have shown no significant loss of performance for sequential as opposed to randomized presentation of speech patterns for training, using the specific algorithm described above. This effectively reduces our I/O bandwidth requirement by a factor of 9 since we only need to broadcast one feature per cycle, storing previous frames' features in a further shift register.

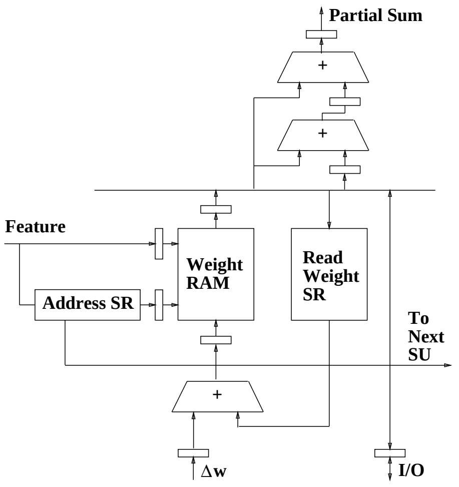

Figure 5: Synapse Unit

We can save a further factor of 3 in required weight memory bandwidth by noting that frames are presented sequentially, and in each weight bank weights are shared. When a new feature is shifted into one of the weight banks, we read the corresponding weight from RAM and store this in a shift register. For the next two patterns we reuse this weight rather than use the shifted feature address to reread the same weight bank.

6.3 Improving Computational Throughput

The simple architecture has only a single adder. This is used for both forward summation and backward weight update. Since individual synapse units take multiple cycles to build up s_j, a single output processor is shared between multiple synapse units.

In the highly pipelined system, each neuron has 3 SUs, one for each shared weight bank. The datapath of a single SU is shown in Figure 5. Control signals have been omitted for clarity. Each SU contains three adders, two for forming forward summations and one to perform weight updates. In total each neuron now has 9 adders in its SUs.

Since each neuron can now process a complete pattern in each cycle, it is necessary to have a separate OP for each neuron. The OP is shown together with the three SUs in Figure 6. The OP now contains a separate register to hold the bias value. There are 3 adders arranged in a two level tree to complete the forward summation from the values generated by the SUs and the bias register. The summation is passed through a sigmoid/error PLA to form Δw_{ij}. A chain of 2 adders accumulates error values for the 3 cycles that each input feature is used in a weight bank, so that only a single add and write is required in the SU weight RAM to implement the 3 weight updates for this feature. The bias register has a separate adder for updates.

In total, each neuron now contains 15 adders, each of which is busy on every cycle.

6.4 Pipeline Operation

The operation of a SU proceeds as follows. On every cycle a new feature address is passed into the SU. This address is used to read the RAM to get the current value of that feature's connection weight. The weight is moved into the forward adder chain to form a new partial sum with the preceding two weights. The read weight value is also stored in the "Read Weight Shift Register" so that it can be added to the update value when it arrives from the OP. The corresponding feature address is stored in the "Address Shift Register". The "Address S.R." is also used to delay the feature address for three cycles before passing it on to the next SU in the same neuron. As update values arrive from the OP, they are added to the weight value emerging from the "Read Weight S.R." and written back to weight RAM using the feature address emerging from the "Address S.R.".

The two stage adder tree in the OP sums the partial sums from the 3 SUs with the bias value and then passes this through the sigmoid/error PLA. The emerging error value is passed into a two adder chain which forms the sum of the new update value with the two preceding update values. This accumulated update value is then passed back to the SUs and the bias register.

6.5 Pipeline Hazards

The overall effect of these enhancements is to achieve a throughput of one pattern learned every clock cycle. However, as with all deeply pipelined systems, there is a danger of pipeline hazards.

The Read After Write hazard present in the simple case is still present but to a greater degree. The pipeline is deeper, so it takes more cycles for the weight update for a training pattern to be reflected in the weight RAM. In addition, when a weight value is read from RAM, it is used for several further patterns without rereading the value from RAM. Again, the overall effect of this Read After Write hazard is not necessarily harmful to learning performance, and our simulations indicate that we can ignore this.

The highly pipelined case also introduces a more serious Read After Write hazard. Consider a serial input stream with two neighboring identical features. The weight updates for patterns containing the first feature will not be written back to RAM before weight values are read out for the second feature. Sometime later the updates for the first feature will be added to

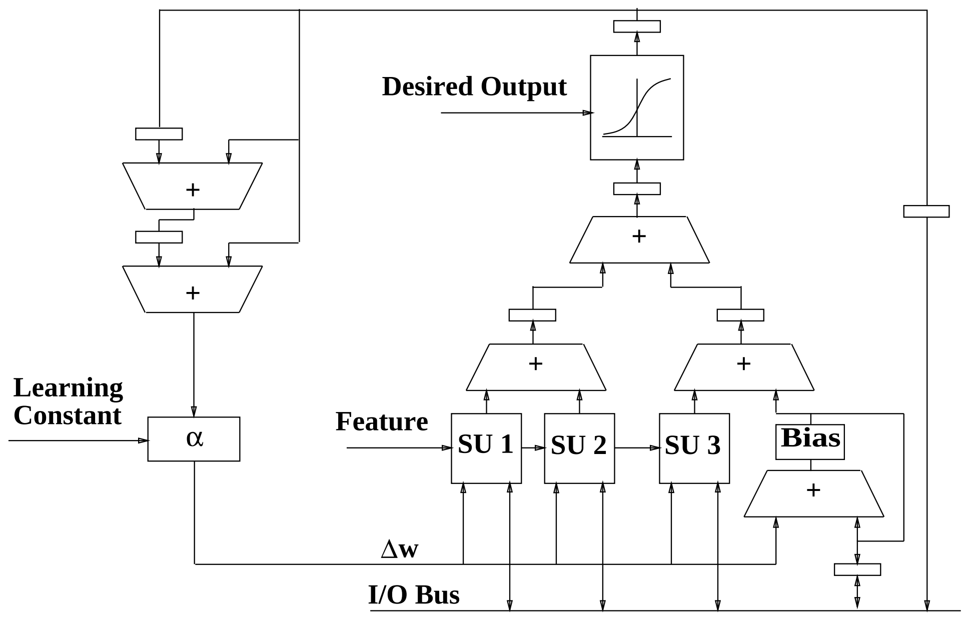

Figure 6: Neuron

the read weights and written back to RAM. However in the next cycle updates for the second identical feature will be added to the original weights and written to RAM, effectively overwriting the updates made for the first feature. In general, weight updates are lost for all but the last of a set of matching features in a time window equal to the depth of the pipeline.

Our simulations indicate that this weight update hazard is not detrimental to learning performance [Koh]. This is perhaps surprising, but one explanation is that successive updates to the same feature would tend to move the network too far towards one direction in the search space.

For other algorithms and for deeper pipelines this overwriting of weight updates may still be a problem. We can avoid this consequence of the Read After Write hazard by using a technique analogous to register forwarding in pipelined register-register architectures [HP90]. We store recently updated weight values together with their RAM addresses in an associatively searchable shift register. The shift register can be searched using the address of an incoming weight update. If there is a match, the updated weight value is read out from the shift register and summed with the new update. Otherwise, the previously read weight emerging from the "Read Weight S.R." is added to the update. The newly updated weight value is then shifted back into the weight update forwarding unit together with the RAM address, while concurrently being written to RAM. In this manner, we can sustain one pattern per cycle throughput without discarding updates. In practice, only one copy of the address shift register needs to be maintained for an entire array. Broadcast signals would control local multiplexers selecting either the read weight or the updated weight.

In order to simplify the hardware, we may choose to only catch weight update hazards over a time window smaller than the pipeline depth. The simplest scheme only checks if the current feature is the same as the last feature, and hence only needs to remember the last update. Our simulations indicate that this minimal scheme would catch 84% of these update hazards for our database.

7 Implementation

Figure 7 shows the layout of a test chip containing the SU datapath for the simple architecture. This has been successfully fabricated through MOSIS in a $2\mu m$ double-metal CMOS process. The device is operational at its design clock rate of 20MHz. The sigmoid PLA has also been successfully fabricated and tested at the same clock rate. The static RAM design is currently in fabrication.

The simple datapath includes most of the cells required for the highly pipelined architecture, and we are currently in the final stages of layout. A floorplan is given in Figure 8. This has been laid out as a row to facilitate placing multiple pipelined neurons on a die. Including routing area, each neuron processor occupies an area approximately $8000 \times 2000 = 16M\lambda^2$. The three synapse units are on the right, with the output processor on the left. In decreasing order of size, the three blocks within each synapse unit are

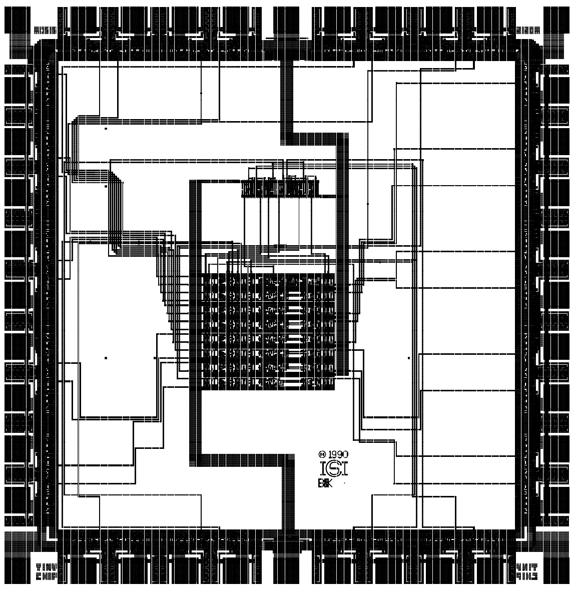

Figure 7: Simple Processor Datapath Chip Layout

the 32 × 48 bit weight RAM, the datapath (including the read weight shift register), and the address shift register. The three blocks in the output processor are the forward datapath including bias register, the error datapath including the α shift unit, and the sigmoid PLA (again in decreasing order of size). The figures in the following section are taken from this floorplan; we do not expect the final area to deviate significantly from these estimates.

8 Performance/Cost Comparisons

One of the major attractions of ANN algorithms is that they exhibit massive parallelism and require only moderate arithmetic precision, hence allowing a wide variety of solutions to attain high performance. Given that high performance is readily obtainable, it is necessary to account for cost when comparing alternative architectures.

One technology-independent way of expressing silicon resource consumption is to consider the die area expressed as the square of the minimum length unit, or λ parameter [Mea89]. In Table 9 we have abbreviated Connections Per Second as CPS, while the unit performance normalized by the scalable area (using λ^2) is called the COnnection Rate Density (CORD). Similarly, for the case in which learning is included, we give the Connection Updates Per Second (CUPS), as well as the corresponding area-normalized measure of Connection Update Rate Density (CURD). The ETANN performance figures come from Intel's data sheet, that for the Texas Instruments' TMS320C30 is from measured speed, and the performance of the circuits proposed here is derived from simulations. All these figures represent the maximum observable throughput for each architecture.

Note that in the table no figures are included for learning on the ETANN chip, since this is done off-chip on the host. Due to relatively slow tunneling mechanisms that are used for writing analog weights on this chip, another IC that did include on-chip learning would probably have a CURD measure that was roughly comparable to the TI chip. Note that the ETANN has the analog equivalent of 6-bit weights as compared to 32-bit floating point on the TI; for most applications, this would probably be insufficient for convergence of the back-propagation learning algorithm.

For forward activation, the ETANN chip is considerably faster than either of our designs. However, normalizing for area, we find that our circuits have similar performance. Furthermore, since our circuits are designed for sparse activation, they can be expected to operate much more efficiently for that case than the ETANN, where many synapses would be idle during a forward cycle.

The pipelined design represents an extreme in special-purpose design, and as such is the highest performance in the table. However, the simple datapath provides some flexibility over the the more complex design. The number of features per pattern can be varied easily, and there is no fixed separation of the input into weight banks. Forward propagation is faster than update mode for this case.

For some points of comparison on measured network learning performance, a single-board Ring Ar-

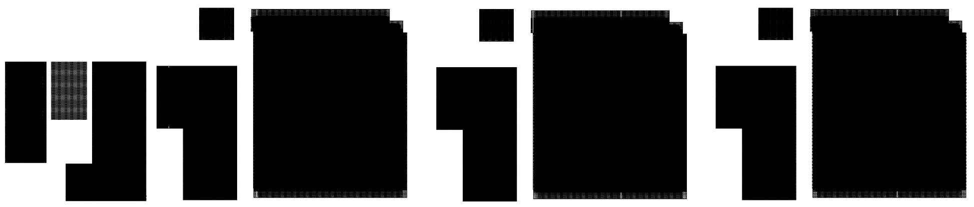

Figure 8: Highly Pipelined Architecture Floorplan

System	Area (λ^2)	CPS	CUPS	CORD	CURD
Simple Datapath	8 x 10^6	20 x 10^6	6.7 x 10^6	2.5	0.8
Highly Pipelined	16 x 10^6	200 x 10^6	200 x 10^6	12.5	12.5
ETANN	300 x 10^6	2000 x 10^6	-	6.7	-
TMS320C30	700 x 10^6	16 x 10^6	4 x 10^6	0.02	0.006

Figure 9: Performance/Cost Comparison

ray Processor (RAP) using 4 TMS320C30 chips achieves about 13 MCUPS [MBAB90], while a Sun SparcStation-1 attains around 0.4 CUPS. For the sparse activation case treated in this design, each of these figures is several times lower. The special purpose designs can sustain their maximum throughput on this algorithm. A typical Sun training run takes several hours, while the comparable figure for a system based around a single pipelined "neuron" would be expected to take a few seconds. This would permit real-time adaptation to changes in the acoustic environment, for instance, with a time constant of a few seconds. Thus, a single pipelined circuit gives several orders of magnitude higher throughput than the more versatile programmable systems.

One of the interesting lessons in this study, underscored by the layout diagrams, is that designs are limited by the required weight memory area. Consequently, there is probably little advantage to be gained by using analog processing, even in those cases when it is simpler and more area-efficient, if the information storage is still most effectively and flexibly done using digital techniques.

9 Conclusions

A highly pipelined neuron training architecture sustaining 200 million connection updates per second within a small silicon area ($16mm^2$ in a $2\mu m$ CMOS process) has been presented. Several such neuron sites can be integrated onto a modestly sized VLSI die to achieve raw performance comparable with special purpose analog designs.

The ANN training algorithm has a number of features which would significantly complicate an analog implementation. Although 6 bits are sufficient for forward propagation, weights need to be stored to 12-bit accuracy for successful learning. Analog circuitry can be designed with such precision, but such design is difficult and the results are unlikely to be compact or to scale well with shrinking process geometries.

Most analog neural net implementations have implemented non-sparse activation with a multiplier per synapse, using a simple 2-dimensional connection pattern [Mor90]. The speech algorithm targeted here has sparse activation, and so this simple approach would be extremely inefficient with less than 1% of the synapses active at a time. Further, only a similar small fraction of the system I/O will be usefully employed. In this case the digital approach is more efficient, since a single arithmetic unit operates on a large bank of weight memory, and only the addresses of active inputs are broadcast. An inactive synapse consumes no processing power or I/O bandwidth and occupies only a few relatively small RAM cells. Adding such virtualization to the existing analog schemes would be difficult, and it is likely that the necessary digital circuitry would dominate the processor area leaving little advantage, if any, to an analog approach.

Comparing the simple and highly pipelined digital architectures it is clear that in this case there is a significant performance/cost advantage to utilizing fewer high performance processors. The silicon area is dominated by the weight RAM. Adding more processor circuitry doubles the total area but increases performance by much larger factors. We believe this will be true even with more general purpose neural architectures, including those with off-chip weight memory. The maximum performance/cost should be first obtained with a single processor. Replicating processors can increase overall performance, but can at best maintain only the same level of performance/cost as the single processor being replicated.

While somewhat slower but programmable approaches (such as the programmable RAP) are perhaps better suited to research in the network algorithms themselves, architectures such as the one presented here demonstrate the performance that can be achieved with relatively conventional design methods for fully-defined applications. Fast and flexible CAD techniques are also being developed to make this kind of design a practical solution to systems that require this throughput [MAKW90] [Asa].

10 Acknowledgements

Phil Kohn ran many of the simulations discussed in the paper. We also thank James Beck for his constructive criticism and advice throughout this work. The National Science Foundation has provided explicit support for this project with Grant No. MIP-8922354, and also with Graduate Fellowship support for Brian Kingsbury. John Wawrzynek received support from the National Science Foundation through the PYI award, MIP-8958568. Finally, we gratefully acknowledge the support of the International Computer Science Institute.

References

[Asa] K. Asanović. OctC++: A C++ interface to the Oct database. ICSI Technical Report. In preparation.

[BH88] T. Baker and D. Hammerstrom. Modifications to artificial neural network models for digital hardware implementation. Technical Report CS/E 88-035, Department of Computer Science and Engineering, Oregon Graduate Center, 1988.

[Hin87] G. Hinton. Connectionist learning procedures. Technical Report CMU-CS-87-115, Carnegie Mellon, 1987.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture — a quantative approach.* Morgan Kaufmann, 1990.

[HTCB89] M. Holler, S. Tam, H. Castro, and R. Benson. An electrically trainable artificial neural network (ETANN) with 10240 "Floating Gate" synapses. In *International Joint Conference on Neural Networks*, pages II–191–196, 1989.

[Koh] P. Kohn. Personal communication.

[MAKW90] N. Morgan, K. Asanović, B. Kingsbury, and J. Wawrzynek. Developments in digital VLSI design for artificial neural networks. Technical Report TR-90-065, International Computer Science Institute, 1990.

[MB90] N. Morgan and H. Bourlard. Continuous speech recognition using Multilayer Perceptrons with Hidden Markov models. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*, pages 413–416, Albuquerque, USA, 1990.

[MBAB90] N. Morgan, J. Beck, E. Allman, and J. Beer. RAP: A Ring Array Processor for Multilayer Perceptron applications. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*, pages 1005–1008, Albuquerque, New Mexico, 1990.

[Mea89] C.A. Mead. *Analog VLSI and neural systems.* Addison-Wesley, 1989.

[Mor90] N. Morgan, editor. *Artificial Neural Networks: Electronic Implementations.* Computer Society Press Technology Series. Computer Society Press of the IEEE, Washington, D.C., 1990.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing. Exploration of the Microstructure of Cognition*, volume 1. MIT Press, 1986.

[Wer74] P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Dept. of Applied Mathematics, Harvard University, 1974.