Training Neural Networks with SPERT-II

Krste Asanović[†§], James Beck[†§], David Johnson[§],
Brian Kingsbury[†§], Nelson Morgan[†§], and John Wawrzynek[†],

[†] University of California at Berkeley
Department of Electrical Engineering and Computer Sciences
Berkeley, CA 94720-1776

[§] International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704-1105

January 8, 1997

Abstract

SPERT-II is a high-performance system for neural network and other signal processing applications. SPERT-II is based on T0, a custom fixed-point vector microprocessor, and serves as an attached processor to a standard workstation host. These systems are primarily being used to speed neural network training for our speech recognition research. We present details of the system and describe how we have mapped neural network training algorithms to SPERT-II.

1 Introduction

Applying artificial neural networks (ANNs) to the problem of computer-based speech recognition has been a research and development focus at ICSI since 1988. This work is characterized by two parallel tasks: adjusting the neural net algorithms to improve recognition scores and building special purpose hardware and software to speed up net training. The task of improving speech recognition also requires research on the non-neural network steps of the process, which will not be covered in this chapter. See [Mor95b] for further information on our complete speech recognition effort.

Our first effort to build a training accelerator was called RAP, for Ring Array Processor. Based on a commercial floating-point digital signal processor (DSP), the RAP uses the simplest communication scheme possible between nodes, a unidirectional ring. A total of 9 RAP installations in the US and Europe were placed in service starting in mid-1990. RAP users collaborate on code development, and the machines have primarily been used for training neural networks for speech recognition. The largest RAP assembled at one time contained 40 nodes, and was benchmarked at a rate of 341 MCPS (millions of connections per second) and 45 MCUPS (millions of connection updates per second) on a neural network we commonly use in our recognition work. (234–1024–61 nodes on the input, hidden and output layers, respectively). On larger networks with equal numbers of units in each layer, the same RAP configuration was measured at up to 573 MCPS and 102 MCUPS. RAP systems in use at customer sites vary in size from 8–16 nodes. The architectural details and programming model for the RAP are similar to the MUSIC system developed at ETH Zentrum, [see Chapter X] and will not be detailed here. Full information on the RAP system is available in [Mor92].

In early 1991, we were considering a follow-on project. Commercial DSP performance was not advancing rapidly, and RAP systems were expensive to replicate and proved inefficient on smaller neural networks. Rather than proceed with a second generation RAP using commercial DSPs, we entered into a collaboration with the EECS Department of the University of California at Berkeley with the goal of designing a full-custom single-chip microprocessor to accelerate neural network algorithms. The result was the T0 vector microprocessor. The first systems built around T0, called SPERT-II, were delivered to customers beginning in the fall of 1995. At the time of writing, there are 25 SPERT-II systems installed at 8 sites in the USA and Europe. We estimate that SPERT-II performance for on-line backprop training of large networks is roughly equivalent to that possible with 20 nodes of a hypothetical RAP-II redesigned to use 1995-era commercial DSPs, at about one fifth the cost. Performance on smaller networks is even more competitive.

In this chapter, we first describe how the algorithms used in our speech recognition research evolved together with our computing systems. We then discuss the design decisions that led to the T0 vector microprocessor. In Section 4 we outline the SPERT-II accelerator board and its software environment. In the next two sections, we detail how we mapped two neural net training algorithms, backprop and Kohonen self-organizing feature maps, to SPERT-II and present the resulting performance before concluding.

2 Algorithm Development

In 1987, Hervé Bourlard showed that in principle, neural networks trained as classifiers using common error criteria would produce estimates of the posterior (class) probabilities [Bou89]. He

further showed that these estimates could in principle be used to derive emission probabilities for use with Hidden Markov Models in sequence recognition problems such as speech recognition. In the following years, researchers learned a number of practical lessons about this approach, and developed systems that exploited this principle [Rob91, Coh92, Ren94, Mor95a]. Our speech recognition group at ICSI has been very active in this area.

From an algorithmic perspective, this approach appeared to offer numerous advantages: no strong reliance on feature distributions, potential for combining binary and continuous features, straightforward incorporation of long term windows of feature vectors, and the availability of a simple discriminative training algorithm.

From a computational perspective, however, the method was inherently more complex during training than the dominant training methods used in speech recognition research. More standard approaches trained density estimates for each class separately, so that each training pattern only affected a small portion of the parameters. In order to share parameters over classes so that the training was discriminative, the neural network approach typically required modification of all parameters for each input pattern. Thus, training was computationally intensive, requiring months of computation on 1990-era workstations for our common tasks.

Initially, this was a major concern, and so we limited our networks by using only binary input vectors. For instance, one could match an input spectrum to one of 256 possible spectra and use as network input a single "1" and 255 "0's". In practice we used 9 such vectors, as well as binary vectors associated with temporal derivatives, plus a few other features. This led to over 5000 inputs, but since nearly all of them were zeroes, the computational load was small and consisted mainly of additions. Furthermore, in practice we found that a hidden layer was not useful for this case, so the net was really quite trivial — a single layer of sigmoidal units with the inputs pointing to weights that should be added in to the pre-sigmoid total. The computational load of this system was so small that it permitted us to work on large problems using conventional workstations.

When some early experiments with continuous inputs showed improved recognition performance, we abandoned the binary inputs method. Moving to continuous inputs, however, meant that we needed a system that could perform the required arithmetic at a high enough speed to facilitate our research. Most of the computational load of this research became the back propagation training that was used for the multi-layered neural networks. While we learned to train these networks with a very small number of passes over the data, each pass still often required trillions of arithmetic operations. Large networks (from 100,000 to 2,000,000 parameters) proved to be better at the overall task than smaller ones, and the data required to train these large networks often consisted of millions of feature vectors.

Once the RAP was developed, such trainings became routine in our Institute, and we were able to share results and divide labor between collaborating labs on an international basis. The RAP achieved good efficiencies, typically 20–90% of peak, on our larger networks, but performance on our smaller networks was much worse. As we looked forward to building a new system, we wanted to have one that was both cheaper, so that we could have many of them, as well as being more efficient for smaller networks.

3 T0: A Vector Microprocessor

During our initial design studies, it became clear that no existing or planned commercial processor would meet our design criteria, so we decided to develop our own programmable VLSI processor.

Competing with commercial microprocessors is a daunting challenge, especially for an academic research group with limited resources and no access to advanced semiconductor processes. However, neural network algorithms are massively parallel and require only moderate arithmetic precision, and we can exploit these features to improve performance.

Fast digital arithmetic units, multipliers and shifters in particular, require chip area proportional to the *square* of the number of operand bits. In modern microprocessors and digital signal processors, a single floating-point multiply-add unit takes up a significant fraction of the chip area. High-precision arithmetic units also require high memory bandwidth to move the large operands. Studies by ourselves and others indicated that fixed-point arithmetic would suffice for back propagation training, with 16-bit weights and 8-bit activation values providing classification performance similar to 32-bit single-precision floating-point [Asa91, Bak88]. The small size of these reduced precision functional units would allow multiple parallel datapaths to be integrated on a single die.

In considering the organization of our processor we were very sensitive to Amdahl's Law [Amd67], namely that if we accelerate only certain portions of the computation then our overall speedup will be limited by those we do not. Although back propagation training execution time is dominated by weight matrix operations on conventional workstations, a complete net training system has to perform many other tasks. These tasks include managing training and cross-validation databases, data format conversion, control of the learning schedule, checkpointing, and status reporting. Furthermore, neural algorithms are often just one component within real-world applications. For example, our speech recognition system must execute various signal processing steps on the raw audio signal to extract spectral features to feed into the neural network used for phoneme estimation, and then perform dynamic programming using the output of the network to find the most likely word sequence. Finally, our algorithms continually evolve as our research progresses, requiring that the processor perform well even for algorithms invented after its design was completed. Together, these considerations led us to design a very general purpose architecture.

We began with the industry-standard MIPS-II RISC instruction set architecture (ISA) and extended this with a fixed-point vector coprocessor; we call the resulting vector ISA "Torrent". The Torrent ISA is very similar to that of a traditional vector supercomputer [Russ78], including vector registers, vector length control, strided and scatter/gather vector memory instructions, and conditional operations.

The combination of a fast scalar processor and a tightly coupled vector coprocessor has proven very successful in attacking a wide range of scientific and engineering problems when implemented in the form of a vector supercomputer. We believe such vector architectures are also very suited to attacking a wide range of neural network, multimedia, and other signal processing tasks when implemented in the form of inexpensive vector microprocessors. In particular, a vector ISA provides a succinct abstraction of data parallelism, one that is both easy to program and that enables straightforward implementations controlling multiple parallel and pipelined functional units.

T0 (for Torrent-0) is the first implementation of the Torrent ISA, and the main components are shown in Figure 1. They include a MIPS-II compatible RISC CPU with a 1 KB on-chip instruction cache, a vector unit coprocessor, an external memory interface and an 8-bit wide serial host interface port (TSIP). The external memory interface supports up to 4 GB of memory over a 128-bit wide data bus. The system coprocessor provides a 32-bit counter/timer and registers for host synchronization and exception handling.

The vector unit contains a vector register file and the VP0, VP1 and VMP vector functional units. The vector register file contains 16 vector registers, each holding 32 elements of 32 bits each.

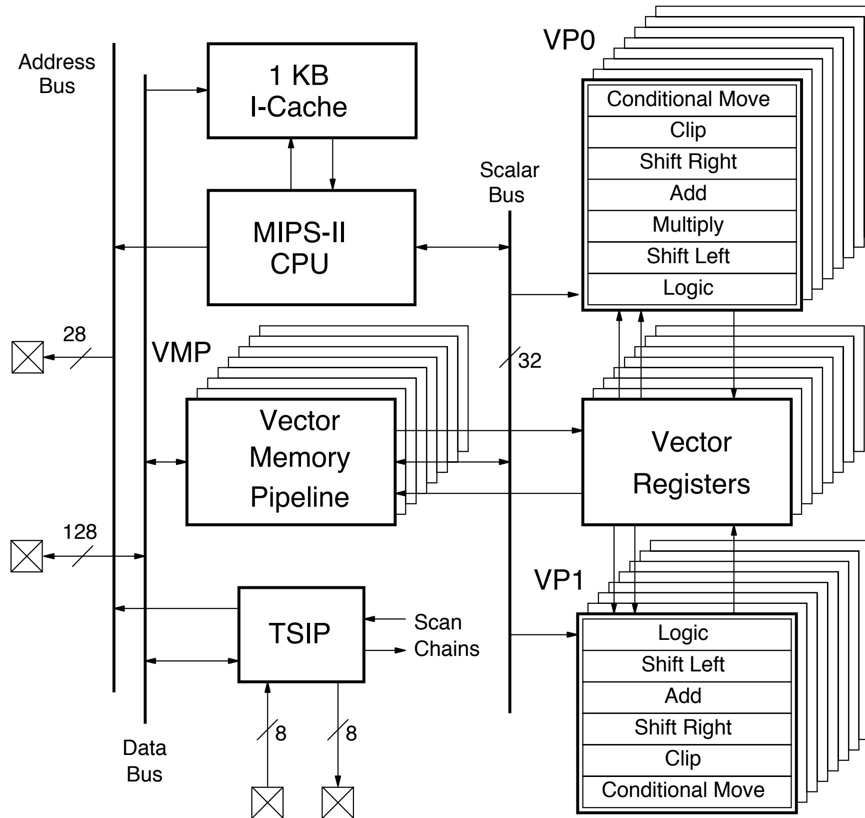

Figure 1: Block Diagram of T0 Micro-architecture.

VP0 and VP1 are vector arithmetic functional units that can perform 32-bit integer arithmetic and logic operations and also support fixed-point scaling, rounding and saturation. Multiplication is supported only in VP0, with 16-bit × 16-bit multiplies producing 32-bit results. Division is performed in software by iterative long division, using an estimate of the divisor's reciprocal to obtain 12 quotient bits per iteration. VMP, the vector memory unit, handles all vector load/store operations, scalar load/store operations, and the vector insert/extract operations.

Vectors are addressed in external memory with three types of load/store options – unit stride, non-unit stride, and indexed access. In unit stride addressing, vector elements occupy consecutive memory locations. In non-unit stride, elements are separated by a constant distance. With indexed access, a vector register provides a set of pointers to the elements of the operand vector. This option efficiently implements parallel table-lookup functions for function approximation, and also supports sparse vector and matrix operations.

All three vector functional units are composed of eight parallel pipelines, and each unit can produce up to eight results per cycle. The T0 memory interface has a single memory address port, limiting non-unit stride and indexed memory operations to a rate of one element transfer per cycle. All vector pipeline hazards are fully interlocked in hardware, and so instruction scheduling is only needed to improve performance, not to ensure correctness. The elements of a vector register are striped across all eight pipelines. With the maximum vector length of 32, a vector functional unit

can accept a new instruction every four cycles. T0 can saturate all three vector functional units by issuing one instruction per cycle to each in turn, leaving a single issue slot open every four cycles for the scalar unit. In this manner, T0 can sustain up to 24 operations per cycle while issuing only a single 32-bit instruction per cycle.

T0 was fabricated in Hewlett-Packard's CMOS26G process using 1.0 μm scalable CMOS design rules and two layers of metal. The die measures 16.75mm $\times$ 16.75mm, and contains 730,701 transistors. At a clock frequency of 40 MHz, the device consumes less than 12 W of power from a 5 V supply. First silicon was received in April 1995, and is fully functional with no known bugs. Full details of the Torrent ISA and the T0 implementation are available in [Asa97a, Asa97b].

4 The SPERT-II Workstation Accelerator

The SPERT-II hardware is a double slot SBus card for use in Sun-compatible workstations. The board contains a 40 MHz T0 vector microprocessor with 8 MB of off-chip SRAM and a Xilinx FPGA device for interfacing with the host, see Figure 2.

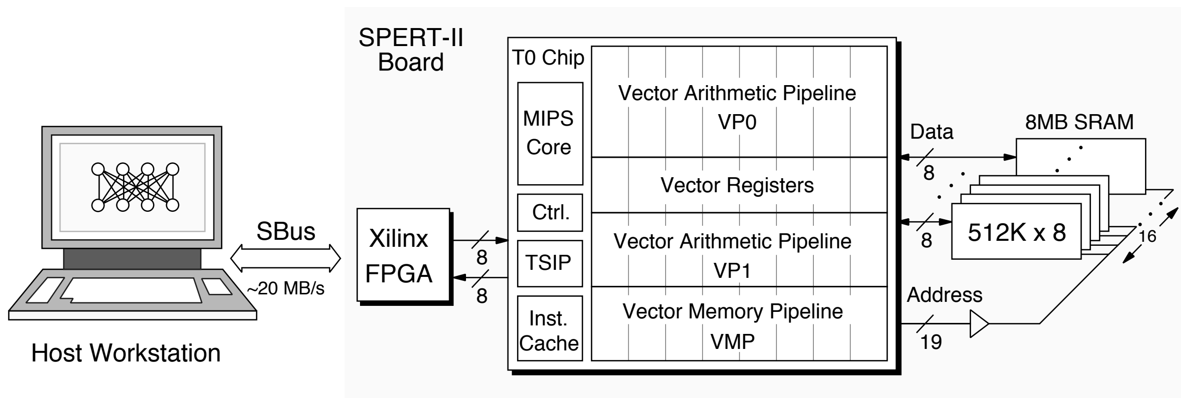

Figure 2: SPERT-II System Organization.

The SPERT-II software environment was designed to be similar to that of conventional workstations to ease the task of porting existing applications and to provide a comfortable environment for developing new code. For a process running on the SPERT-II board, its "operating system" comprises two programs: a small kernel that runs on T0 and a user level server running on the host. To run a program on SPERT-II, a user invokes the server, passing the name of the SPERT executable as an argument. The server resets T0, loads the kernel and SPERT executable and then waits to handle I/O requests on behalf of the SPERT program. The software structure is shown in Figure 3.

By basing our design on a MIPS scalar architecture, we gain access to a wealth of existing software support, mostly taken from the popular GNU tool set. We use an unmodified version of `gcc`, an optimizing scalar C and C++ cross-compiler. Although T0 does not include a hardware floating-point unit, MIPS-II floating-point instructions are trapped and emulated by the kernel, simplifying porting. We have modified the `gdb` symbolic debugger to debug T0 programs remotely from the host, and to give access to the vector coprocessor registers. We extended the `gas` assembler to include support for the new vector instructions and added an optimizing scheduler to avoid vector

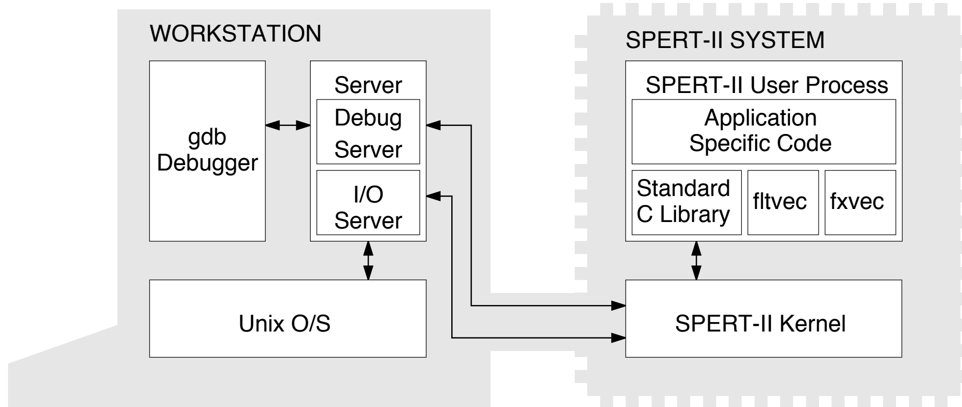

Figure 3: SPERT-II Software Environment.

instruction interlocks. We also employ the GNU linker and other binary utility programs such as library archivers.

Currently, the only access we provide to the vector unit is either via library routines or by coding directly in assembler. We have developed an extensive set of optimized vector library routines including many fixed-point matrix and vector operations. Vectorizable floating-point code can make use of our vectorized IEEE single precision floating-point emulation library which can achieve up to 14 MFLOPS. Another group is developing a vectorizing compiler for T0 [Vri96].

5 Mapping Back Propagation to SPERT-II

For our speech recognition work, we mostly use a simple feed-forward multi-layer perceptron (MLP) with three layers. Typical MLPs for this task have 100–400 input units. The input layer is fully connected to a hidden layer of 100–4000 hidden units. The hidden layer is fully connected to an output layer that contains one output per phoneme, typically 56–61. The hidden units incorporate a standard sigmoid non-linearity, usually $f(x) = 1/(1 + e^{-x})$. The output units compute either a sigmoid or a "soft-max" activation function:

$$f(x) = \frac{e^x}{\sum_i e^{x_i}}$$

We can classify back propagation training techniques according to the number of patterns that are presented between weight updates. An "on-line" procedure updates weights after every pattern presentation. An "off-line" or "batch" procedure accumulates weight updates over the entire training database before updating the weights. Batch mode training is easier to parallelize and tends to give greater raw MCUPS numbers. However, for large realistic problems such as speech or image recognition, the data sets tend to be quite redundant, and we have found that a single pass of on-line training will tend to accomplish much more than a single pass of batch training.

While some researchers have found slow convergence using backprop to train smaller networks with smaller data sets, we experience relatively fast convergence using on-line backprop to train large networks with large real world data sets. We have found when training large nets (40,000 to

over a million weights) using up to several million pattern presentations and starting with random weights, that only 3–10 passes are required to obtain convergence. In some cases, when adapting a previously initialized network, only a single pass over a new training set is needed.

The most compute-intensive operations in on-line back propagation training are those involving the weight matrices. Three operations are performed: forward propagation, error back propagation, and weight update. On T0, we store weight matrices in "input-major" form, where element `weight[i][j]` holds the weight connecting input `i` to neuron `j`. The three on-line backprop matrix operations then correspond to three standard linear algebra routines in the T0 library: vector-transpose $\times$ matrix multiply, matrix $\times$ vector multiply, and scaled outer-product accumulation, respectively. We chose our weight matrix orientation to favor forward propagation because this is performed more often than back propagation; error back propagation is not required on the input-to-hidden weight matrix during training, and our cross-validation procedure monitors network convergence by running forward propagation on a separate test database once per epoch.

On-line forward propagation uses the vector-transpose $\times$ matrix multiply library routine. This routine processes large matrices 128 columns at a time, with matrix elements read using unit-stride. Each 16-bit element from the source vector is loaded into a scalar register, then multiplied by 128 16-bit elements loaded from one row of the source matrix, with the products accumulated into 128 32-bit sums held in four vector registers. To reduce memory bandwidth further, two 16-bit source vector values are read at once using a single 32-bit scalar load. This loop requires one cycle to read two source vector elements, then 32 cycles to read the 2×128 matrix elements. The multiplies and adds can be overlapped with the memory accesses, giving a peak throughput of 310 MCPS, or nearly 97% of peak. After processing all elements in each of the 128 columns, the accumulated 32-bit sums are stored to memory at the rate of 4 per cycle, and the next set of 128 columns are processed. When there are fewer than 128 columns remaining, the routine switches to a different loop that processes up to a single vector register full of columns at a time. This loop also fetches two 16-bit source vector operands in one cycle, and takes 9 cycles to process up to 64 multiplies and adds, achieving up to 284 MCPS, or nearly 89% of peak.

On-line error back propagation uses the matrix $\times$ vector multiply library routine. The matrix is now accessed in a transposed manner compared to vector-transpose $\times$ matrix multiply, with dot-products calculated across the rows of the matrix. One approach would be to stride down a column to collect the source matrix elements that are multiplied by each source vector element. However, non-unit-stride accesses only run at the rate of 1 element per cycle on T0 and so this approach would yield only 1/8 of peak performance. Instead, the library routine operates on 12 rows of the source matrix at a time, using 12 vector registers to hold 32 partial sums for each row. The source vector is now accessed with a unit-stride vector load, and multiplied element-wise with 12 vectors of 32 matrix elements. This loop takes 4 cycles to load the 32 16-bit source vector elements, then 12×4 cycles to load the 12×32 16-bit matrix elements. Again, multiplies and adds can be fully overlapped, giving a peak performance of 384 multiply-adds every 52 cycles, or 92% of peak for large matrices. However, there is an additional overhead at the end of each set of 12 rows. The 12 vector registers contain 32 partial sums that must be reduced to single 32-bit values before they are stored in the result vector. This process can take up to 140 cycles for all 12 vectors and is a considerable overhead for matrices with few columns. When there are fewer than 12 rows remaining, the routine drops down to perform 4, 2, and finally 1 row at a time, with corresponding decreases in performance.

The most time-consuming routine in on-line back propagation training is weight update. This

maps to a vector outer-product and matrix accumulate operation. The routine operates on 8 rows of the matrix at a time, first loading 8 16-bit elements from the first source vector into 8 scalar registers. These values are reused across the entire width of the matrix. The matrix is updated in blocks of 8 rows by 32 elements. First, 32 16-bit elements are read from the second source vector into a vector register. These values are then multiplied by each of the 8 scalar values in turn with the products summed in to each 32 element section of each of the 8 matrix rows. The routine is dominated by the time taken to read the 16-bit weights in from memory and write them back out. The process takes 68 cycles to perform 256 multiply-adds, and so runs at up to 47% of peak.

For on-line back propagation training of large 3-layer neural networks, with 16-bit weights and equal numbers of units in the input, hidden, and output layers, we can calculate that the maximum asymptotically achievable performance will be 310 MCPS and 86 MCUPS. The measured performance on several real matrix sizes is given below. Note that the rate depends on the ratio of input to output units; with fewer output units less time is spent on error back propagation and the achievable MCUPS rate will be higher. With fewer output units, the maximum achievable performance approaches 101 MCUPS.

Although the $O(n^2)$ matrix operations dominate performance for large networks, the other $O(n)$ operations required to handle input and output vectors and activation values would cause significant overhead on smaller networks if they were not also vectorized.

The sigmoid activation function is implemented using a library piecewise-linear function approximation routine. This routine makes use of the vector gather operations to perform the table lookups. Although T0 can only execute vector gather operations at the rate of one element transfer per cycle, the table lookup routine can simultaneously perform all the arithmetic operations for index calculation and linear interpolation in the vector arithmetic units, achieving a rate of one 16-bit sigmoid result every 1.5 cycles. Similarly, a lookup-table based vector `logadd` routine is used to implement the soft-max function, producing one result every 2.25 cycles.

We have standardized on the IEEE floating-point format to store our training databases on disk. T0 uses vector library routines to convert single precision IEEE floating-point format to the internal 16-bit fixed-point representation. These conversion routines operate at the rate of 2.4 cycles per element converted.

We compared SPERT-II performance against two commercial RISC workstations. One was a SPARCstation-20/61 containing a single 60 MHz SuperSPARC+ processor with a peak performance of 60 MFLOPS, 1 MB of second level cache, and 128 MB of DRAM main memory. The other was an IBM RS/6000-590, containing the RIOS-2 chip-set running at 66.7 MHz with a peak performance of 266 MFLOPS, 256 KB of primary cache, and 768 MB of DRAM main memory. The SPERT-II system contains a single T0 processor running at 40 MHz with a peak performance of 640 million fixed-point arithmetic operations per second and 8 MB of SRAM main memory, mounted in a SPARCstation-5/70.

Figure 4 shows the performance of the three systems for a set of three-layer networks on both forward propagation and back propagation training. Table 1 presents performance results for two speech network architectures. We used three-layer neural networks with total connectivity between adjacent layers. In Figure 4, for ease of presentation, we use networks with the same number of units per layer. The networks presented in the table have a different number of units per layer, as indicated there. The sigmoid function was the hidden layer activation function, and the soft-max function was the output layer activation function.

The workstation version of the code performs all input and output and all computation using

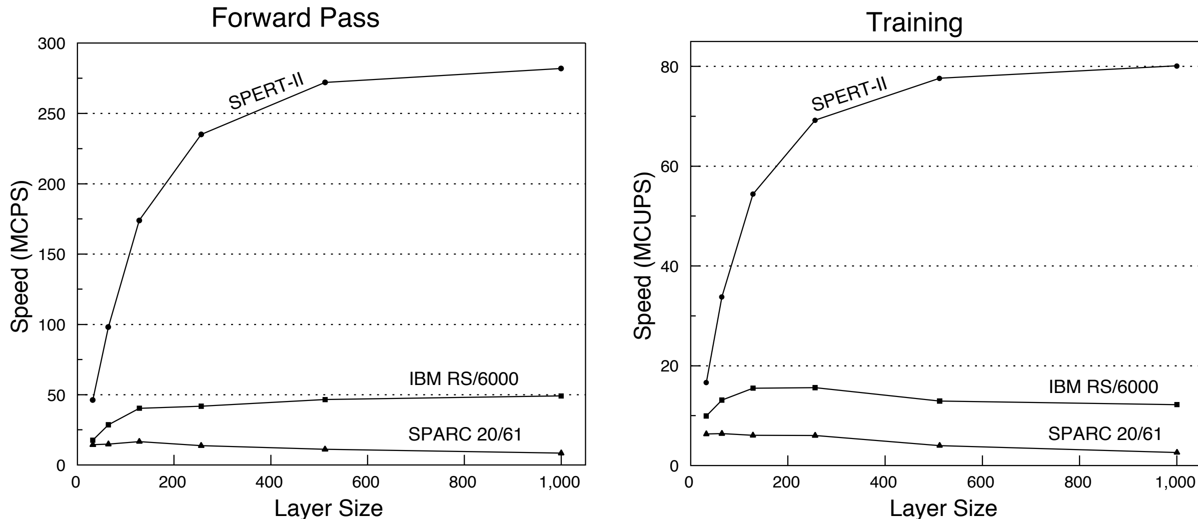

Figure 4: Performance evaluation results for on-line backpropagation training of 3 layer networks, with equal numbers of units in the input, hidden, and output layers. The workstations perform all calculations in single precision IEEE floating-point, while SPERT-II uses 16 bit fixed-point weights.

IEEE single precision floating-point arithmetic. We extensively hand optimized the matrix and vector operations within the back propagation algorithm using manual loop unrolling with register and cache blocking. The SPERT-II timings below include the time for conversion between floating-point and fixed-point for input and output.

Net Type	Net Size (In–Hidden–Out)	SPERT-II	Sparc-20/61	IBM RS/6000-590
Forward Pass (MCPS)				
Small speech net	153–200–56	181.0	17.6	43.0
Large speech net	342–4000–61	276.0	11.3	45.1
Training (MCUPS)				
Small speech net	153–200–56	57.1	7.00	16.7
Large speech net	342–4000–61	78.7	4.18	17.2

Table 1: Performance Evaluation for Selected Net Sizes.

The SPERT-II system clearly performs better even for the smallest networks, such as those with only 32 units per layer. For large networks, the SPERT-II system performed 30 times better than the Sparcstation-20/61 and about 6 times better than the IBM RS/6000-590. On-line back propagation training exhibits poor temporal locality. Every other weight is touched in between accesses to any single weight and so if the weight matrices are too large to fit in cache, there is marked drop in performance. With more modern workstations, such as the Sun Ultra-1/170, we have observed an even more extreme drop off due to the increased gap between floating-point performance and sustainable memory bandwidth.

In [Asa91,Waw96], we report on various experiments that show that the fixed-point trained net matches the floating-point net performance for frame-level phoneme classification. However, we have discovered that, in some cases, when we subsequently try to use the fixed-point network to generate probabilities for use in word or utterance level recognition, we attain noticeably poorer recognition scores compared with the floating-point trained network. We determined that the problem is due to loss of small weight updates through underflow caused by the limited range of the 16-bit weights.

Our solution to this problem was to extend the weight representation with a second 16-bit weight matrix holding low order bits. The forward and error back propagation routines are unchanged and operate on the upper 16-bit weight matrices as before. A new weight update routine is provided that concatenates the two 16-bit halves to form a full 32-bit weight, adds in the update, then stores the updated weight back into its two component matrices. Using this technique we have found that word level recognition accuracy is now indistinguishable from nets trained with IEEE single precision floating-point. However, this routine is slower than before, requiring twice as much memory traffic for each weight update. The inner loop updates weight matrices 1 row by 32 elements at a time and requires 17 cycles to update 32 weights. This represents only 23% of peak arithmetic performance.

When using the 32-bit weight updates, the maximum achievable forward propagation rate is unchanged at 310 MCPS. The asymptotic maximum achievable training rate for equal numbers of input, hidden, and output units drops to 54 MCUPS. With small output layers, the maximum rate is 59 MCUPS.

Between the two extremes of on-line and off-line training lies what we refer to as "bunch" mode training, where we update weights after every group, or bunch, of patterns. Our initial experiments have found that bunches of between 32 to 1000 patterns appear to give comparable results with comparable convergence speed to the on-line approach, while permitting an amortization of many sources of per-pattern overhead. We retain the same weight matrix format as before, but now the three steps in back propagation training can be formulated as variants of matrix $\times$ matrix multiply. Bunch mode is also more efficient for large weight matrices on cached workstations, allowing reuse of cached weight values and correspondingly improving training performance [Bil97]. We note, however, that bunch mode back propagation training is not always possible. For example, with recurrent neural network architectures there are inter-pattern dependencies that prevent this optimization [Rob91].

Bunch-mode forward propagation is equivalent to a matrix $\times$ matrix multiply. The inner loop of the T0 matrix $\times$ matrix multiply library routine computes a 4 row by 32 element block of the destination matrix. Each iteration of the inner loop reads four 16-bit values into scalar registers from the first source matrix and 32 16-bit elements from the second source matrix into a vector register. Now four vector $\times$ scalar products can be calculated and accumulated into the four vector registers holding the 32-bit sums. This loop is dominated by the multiplies and adds and runs at the peak speed of 320 MCPS.

Bunch-mode error back propagation is equivalent to a matrix $\times$ matrix-transpose multiply. The inner loop of the T0 library routine computes the sums for a 12 row by 32 element block of the destination matrix. The first matrix is accessed with scalar loads, while the second matrix is accessed with a non-unit stride vector load. The overhead of the strided vector access is amortized over 12 vector multiplies and adds. Each loop iteration requires 53 cycles to calculate 384 multiply-adds, and so achieves 91% of peak. Although the peak rate is slightly lower, this loop is more

efficient than the on-line back propagate routine in practice, since it avoids the reduction step.

Bunch-mode weight-update is equivalent to a matrix-transpose × matrix multiply accumulated into the destination matrix. Because the overhead of loading and storing the two halves of each 32-bit weight is now amortized over the patterns in a bunch, we have coded only the 32-bit weight update version of this routine. The inner loop is similar to the matrix × matrix multiply routine, with the order in which scalar values are loaded from the first source matrix transposed. Again, the routine is multiply-add dominated and the inner loop achieves 100% of peak.

The maximum achievable rates for bunch-mode training with 32-bit weight updates, for large bunches and for large matrices with equal numbers of input, hidden, and output units is 320 MCPS and 125 MCUPS. For small output layers, the maximum achievable training rate increases to 160 MCUPS. The bunch-mode routines have less overhead than the on-line routines, and perform better on smaller weight matrices. In addition, the amount of data operated on by each call to the $O(n)$ routines is increased by the bunch size, reducing the overhead per element.

6 Mapping Kohonen Nets to SPERT-II

Kohonen self-organizing feature maps (KSOFM) are a form of neural network that can learn to cluster data into topological maps with no external supervision [Koh82]. They have been used in a wide range of applications including image classification and data compression [Myk95].

In practice, Kohonen nets are quite small [Myk95], and the neighborhood radius shrinks rapidly so that it is less than a few neurons wide for most of the training process. This makes training difficult to parallelize efficiently, first because of the need to find the minimum distance neuron, and second because only a few neurons' weights are updated at any time step. The algorithm can be modified to run in bunch mode, with weights updated less frequently to allow a higher update rate on parallel implementations, but this can lead to much slower convergence giving longer run times overall [Cor94].

We have implemented two vectorized library routines for KSOFMs with a Euclidean distance measure. The first routine, `forward`, takes an input vector and a weight matrix and finds the neuron with the minimum squared Euclidean distance from the input vector. The matrix is stored in input-major form, as in our back propagation routines, and the routine works on 32 columns at a time. Each 16-bit input vector element is read into a scalar register, then subtracted from a vector of up to 32 16-bit weights loaded from one row of the matrix. The differences are squared, and accumulated into a vector register. This inner loop is dominated by the three arithmetic operations per connection and takes 6 cycles to compute 32 connections, giving an asymptotic peak of 213 MCPS. To locate the winning neuron, the routine keeps an index for each vector register element in a vector register that is initialized from memory at the start of the routine. These indices are incremented by 32 using a vector add as the routine steps over the columns of the matrix. Two global vector registers hold the 32 minimum sums seen so far and the 32 indices of these minima. At the end of the columns, the new sums are compared to the global sums at each element position, and if smaller, the sums and indices at each element position are copied into the global vector registers. At the end of the routine, the 32 global minimum sums and indices must be reduced down to a single minimum sum and index, which takes 45 cycles. The `forward` routine works for any neural map topology, including 1-D, 2-D, and 3-D grids. It is the responsibility of the calling routine to convert the column index of the minimum sum neuron back into coordinates in the neural map. This can be easily achieved using pre-computed lookup tables. Table 2 gives the

measured performance of the forward pass routine on T0 for various sized networks taken from [Myk95]. Performance is given in MCPS (millions of connections per second).

Application	Neuron Topology	Input dimension	SPERT-II (MCPS)
Speech coding	10×10	12	100.1
	16×16	12	132.9
	20×20	12	134.0
Radar clutter classification	10×10	11	93.4
	20×20	11	130.9
Gas concentration	12×12	32	159.3
Binocular Receptive Fields	16×16	256	208.9

Table 2: Performance of SPERT-II KSOFM forward pass on real-world networks from [Myk95].

The second routine, `update`, modifies weights for some number of neurons located in contiguous columns in the array. One input argument is a vector of update factors, one per neuron, that give the strength of the update for each neuron. The `update` routine is typically called multiple times to update all neurons in the neighborhood of the winner. For example, in a 2-D grid, it would be called once for each row in the neighborhood. In this manner, the routine can support any shape of neighborhood on any map topology. The routine works on one neuron at a time, accessing the weights using strided vector memory operations. The weights are subtracted from the input vector elements to find the element distances, then multiplied by the update factor for this neuron, before being added back into the weight which is then stored back. The routine requires one cycle to load the factor, then $2N$ cycles to load and store N weights. Though all arithmetic can be overlapped, this routine runs at only 9.4% of peak arithmetic performance. Fortunately, training neighborhoods shrink rapidly in practice, and so only a small fraction of the neurons need to be updated.

To measure training performance, we used a benchmark supplied by EPFL, Switzerland [Cor94], which uses a KSOFM to implement speech coding with vector quantization. This benchmark has 12-dimensional input vectors mapped to a 2-dimensional neuron grid. EPFL supplies the training data already converted to a 16-bit fixed-point representation. Over time, the adaption rate of the weights varies as:

$$\alpha(t) = \frac{\alpha_0}{1 + K_\alpha \cdot t}$$

where t is the number of patterns presented to the network, α_0 is the initial adaption rate, and K_α is a time constant that controls how fast the adaption rate decreases

The size of the neighborhood decreases over time according to:

$$R(t) = 1 + \frac{R_0}{1 + K_R \cdot t}$$

where R_0 defines the initial radius, and K_R is a time constant that controls the rate at which the radius shrinks. Within the neighborhood, the adaption rate drops linearly, from $\alpha(t)$ at the winning neuron to 0 for neurons outside radius $R(t)$. The benchmark uses the Manhattan distance on the 2-D grid to determine how far a neuron is from the winner.

Computing the change in neighborhood parameters at every pattern can be time consuming. Fortunately, updating the parameters more slowly seems to have little effect on convergence. For the timings presented below, we updated the adaption rates and radius every 100 patterns. We calculate the parameters using software-emulated floating-point on T0, convert to fixed-point, and cache them in an array. We have found no significant difference between floating-point and fixed-point trained nets in either convergence rate or final quantization distortion.

Table 3 shows training performance on the EPFL speech coding benchmark. Training time is given over all 30,000 patterns. These are small networks and small training databases, with total run times of around one second, yet T0 still achieves high performance. On larger networks and with larger training sets, the performance will asymptotically approach the peak forward pass rate of 213 MCUPS.

Neuron Topology	R_0	SPERT-II Time (s)	SPERT-II (MCUPS)
10×10	5	0.795	45.2
16×16	8	1.072	86.0
20×20	10	1.431	100.6

Table 3: Performance of SPERT-II on the EPFL benchmark for KSOFM training for all 30,000 training patterns. The neighborhood is updated after every 100 patterns with the intial radius, R_0, set to half the grid dimensions. The other training parameters are $\alpha_0 = 0.1$, $K_\alpha = 0.0025$, and $K_R = 0.02$.

7 Conclusions

We have presented a workstation accelerator based on a unique custom vector microprocessor, and have described how we have vectorized several neural network training tasks. We have evaluated the training and forward propagation performance of the system for a range of neural network sizes, and showed that SPERT-II is efficient in practice even on smaller networks common in real applications.

By adopting a vector architecture, we have not had to sacrifice either generality or a convenient software environment to achieve this high efficiency. We have also achieved promising results in a wide range of other important application areas, including image and audio signal processing, audio synthesis, image compression, and cryptography. This leads us to believe that vector processors will be an important class of processors in the future.

Acknowledgments

Thanks to Bertrand Irissou for his work on the T0 chip, John Hauser for Torrent libraries, John Lazzaro for his advice on chip and system building, and Jerry Feldman. Primary support for this work was from ONR URI Grant N00014-92-J-1617, ARPA contract number N0001493-C0249, NSF Grant No. MIP-9311980, and NSF PYI Award No. MIP-8958568NSF. Additional support was provided by ICSI. IBM donated the RS/6000.

References

[Amd67] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, AFIPS Conference Proceedings, 30, 1967, pp. 483–485.

[Asa91] K. Asanović and N. Morgan, Experimental determination of precision requirements for back-propagation training of artificial neural networks, *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 9-16, Kyrill&Method Verlag, Munich, 1991.

[Asa97a] K. Asanović and D. Johnson, Torrent architecture manual, Technical Report CSD-97-930, Computer Science Division, University of California at Berkeley, 1997

[Asa97b] K. Asanović and J. Beck, T0 engineering data, Technical Report CSD-97-930, Computer Science Division, University of California at Berkeley, 1997

[Bak88] 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.

[Bil97] J. Bilmes, K. Asanović, C. Chin, and J. Demmel, Using PHiPAC to speed error back propagation, *to appear International Conference on Acoustics, Speech, and Signal Processing*, 1997.

[Bou89] H. Bourlard and C.J. Wellekens, Links between Markov models and multilayer perceptrons, *Advances in Neural Information Processing Systems 1*, D.J. Touretzky (ed.), San Mateo: Morgan Kaufmann, pp. 502-510, 1989.

[Coh92] M. Cohen, H. Franco, N. Morgan, D. Rumelhart and V. Abrash, Hybrid neural network/hidden Markov model continuous speech recognition, *Proc. of Intl. Conf. on Speech and Language Processing*, pp. 915-918, Banff, CANADA, 1992.

[Cor94] T. Cornu and P. Ienne, Performance of digital neuro-computers, *Proceedings Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, pp. 87–93, September 1994.

[Ham90] D. Hammerstrom, A VLSI architecture for high-performance, low-cost, on-chip learning, *Proceedings of the International Joint Conference on Neural Networks*, pp. II–537–543, 1990.

[Kan92] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

[Koh82] T. Kohonen, Self-organizing formation of topologically correct feature maps, *Biological Cybernetics*, 43(1):59-69, 1982.

[Mor92] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J .Beer, The Ring Array Processor (RAP): A multiprocessing peripheral for connectionist applications, *Journal of Parallel and Distributed Computing,* Special Issue on Neural Networks, 1992, v14, pp. 248-259, 1992.

[Mor95a] N. Morgan and H. Bourlard, Continuous speech recognition: An introduction to the hybrid HMM/connectionist approach. *Signal Processing Magazine*, pp 25-42, May 1995.

[Mor95b] N. Morgan and H. Bourlard, Neural networks for statistical recognition of continuous speech, Proceedings of the IEEE, pp 742-770, May 1995.

[Myk95] G. Myklebust and J. G. Solheim, Parallel self-organizing maps for actual applications, *Proceedings of the IEEE International Conference on Neural Networks*, Perth, 1995.

[Ren94] S. Renals, N. Morgan, H. Bourlard, M. Cohen and H. Franco, Connectionist probability

estimators in HMM speech recognition *IEEE Transactions on Speech and Audio Processing*, part II, pp. 161-174, Jan 1994.

[Rob91] A.J. Robinson and F. Fallside, A recurrent error propagation network speech recognition system, *Computer, Speech and Language*, vol. 5, pp. 257-286, 1991.

[Russ78] R. M. Russel, The CRAY-1 computer system, *Communications of the ACM*, vol. 21, no. 1, pp 63–72, January 1978.

[Vri96] D. De Vries and C. G. Lee, A vectorizing SUIF compiler, *Proceedings of the First SUIF Compiler Workshop*, Stanford University, January 1996, pp 59–67.

[Waw93] J. Wawrzynek, K. Asanović and N. Morgan, The design of a neuro-microprocessor, *IEEE Journal on Neural Networks*, 4(3), 1993.

[Waw96] J. Wawrzynek, K. Asanović, B. E. D. Kingsbury, J. Beck, D. Johnson, and N. Morgan, SPERT-II: A vector microprocessor system, *IEEE Computer*, March 1996, vol. 29, no. 3, pp 79–86.

Figures

Figure 5: Photograph of T0 on board.

Figure 6: Photograph of top of SPERT-II board.

Figure 7: Photograph of bottom of SPERT-II board.