A Fast Kohonen Net Implementation for Spert-II

Krste Asanović

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720-1776

Abstract. We present an implementation of Kohonen Self-Organizing Feature Maps for the Spert-II vector microprocessor system. The implementation supports arbitrary neural map topologies and arbitrary neighborhood functions. For small networks, as used in real-world tasks, a single Spert-II board is measured to run Kohonen net classification at up to 208 million connections per second (MCPS). On a speech coding benchmark task, Spert-II performs on-line Kohonen net training at over 100 million connection updates per second (MCUPS). This represents almost a factor of 10 improvement compared to previously reported implementations. The asymptotic peak speed of the system is 213 MCPS and 213 MCUPS.

1 Introduction

Spert-II is a workstation accelerator constructed around the T0 vector microprocessor [1]. Although most production use of Spert-II systems has been to accelerate error backpropagation training of multi-layer perceptrons used within continuous speech recognition systems, we designed Spert-II as a flexible, general-purpose accelerator. In this paper we report on a second neural net application, the Kohonen Self-Organizing Feature Map (KSOFM). We first review the KSOFM classification and training algorithms, then describe pertinent details of the T0 vector microprocessor and the Spert-II system. In Section 4, we describe the implementation and performance of the two core library routines, pattern classification and weight update, and show how these are used within a complete benchmark application. We compare the benchmark results with previously reported numbers before concluding.

2 Kohonen Net Algorithms

Kohonen self-organizing feature maps (KSOFM) are a class of artificial neural networks that can form a non-linear mapping from a higher dimensional input space to a lower dimensional neuron map with no external supervision [2]. They have been used in a wide range of applications including image classification and data compression [3].

Once trained, the network can be used to classify novel patterns by finding the neuron with the weight vector closest to the input vector. Typically, a Euclidean distance measure is used, and we can select the *winning* neuron, $c(t)$, for an input pattern $\mathbf{X}(t)$, using

$$c(t) = \underset{i}{\operatorname{argmin}}\,(\|\mathbf{X}(t) - \mathbf{W}_i\|^2) \tag{1}$$

where $\mathbf{W}_i$ is the weight vector for neuron i, and t is a discrete time index. The location of the winning neuron within the neural grid represents the category of the input vector.

During training, the weights are adapted according to

$$\mathbf{W}_i(t+1) = \mathbf{W}_i(t) + \alpha(t) \cdot \Lambda(t, d(i, c(t))) \cdot (\mathbf{X}(t) - \mathbf{W}_i(t)) \tag{2}$$

where $\alpha(t)$ is a global adaption rate that is typically reduced over the course of the training procedure. The function $d(i, c(t))$ measures the distance in the neural map of neuron i from the winner $c(t)$. The neighborhood function $\Lambda(t, d(i, c(t)))$ restricts the weight updates to a neighborhood around the winning neuron. Typically, the size of the affected neighborhood is also reduced during the training procedure.

Kohonen nets used in practice are quite small [3], and during training the neighborhood radius shrinks rapidly so that it is only a few neurons wide for most of the time. These attributes make Kohonen nets difficult to parallelize efficiently. Finding the winning neuron requires global communication for every pattern. During on-line training, only a few neurons' weights are updated at any time step and this must be completed before presenting the next pattern. The training algorithm can be modified to run in "bunch" mode, with weights updated only after presenting a group, or bunch, of patterns to allow a higher presentation rate on parallel implementations, but this can cause slower algorithmic convergence and longer overall training times [4].

3 T0 and Spert-II Overview

For the Spert-II project, we desired an architecture that was efficient at running not only a variety of neural network algorithms, but also other computation intensive components within typical real-world applications. We also required that the architecture be straightforward to program. These goals led us to design a new vector instruction set architecture (ISA), "Torrent" [5], based on the industry standard MIPS-II RISC ISA [6]. The Torrent ISA is very similar to that of a traditional vector supercomputer [7], including vector registers, vector length control, strided and scatter/gather vector memory instructions, and conditional operations.

T0 (Torrent-0) is the first implementation of the Torrent ISA, and is a full-custom single-chip vector microprocessor developed in a collaboration between UCB and ICSI [8]. T0 was fabricated using $1.0\,\mu$m scalable CMOS design rules and two layers of metal. First silicon was received in April 1995, and is fully functional with no known bugs. The die measures 16.75mm $\times$ 16.75mm, and contains 730,701 transistors.

T0 includes a MIPS-II compatible RISC CPU with a 1 KB on-chip instruction cache, a fixed-point vector coprocessor, and an external memory interface with a 128-bit data bus. The vector coprocessor 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.

VP0 and VP1 are vector arithmetic functional units that can perform 32-bit integer arithmetic and logic operations with support for fixed-point scaling, rounding and saturation. Multiplication is supported only in VP0, with 16-bit $\times$ 16-bit multiplies producing 32-bit results. Both arithmetic units contain eight parallel pipelines and can produce eight results per cycle.

VMP, the vector memory unit, handles all vector load/store operations. Vectors in memory can consist of signed or unsigned, 8-bit, 16-bit, or 32-bit operands, and are accessed with three types of vector load and store instructions: unit stride, non-unit stride, and indexed access. For unit stride accesses, vector elements occupy consecutive memory locations and VMP can saturate the memory system, moving 128 bits per cycle. The unit stride instructions also allow a parallel arbitrary post-increment of the scalar base address register to be specified. For non-unit stride, elements are separated by a constant stride specified in a second scalar register. With indexed access, a vector register provides a set of offset pointers that are added to a scalar base address to form the effective address of the vector elements. The T0 memory interface has only 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 full vector chaining is implemented. 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 complete up to 24 results per cycle while issuing only a single 32-bit instruction per cycle.

Spert-II is a double-slot SBus card that integrates a T0 processor running at 40 MHz with 8 MB of external SRAM memory, and is used as an attached processor for Sun-compatible workstations. The Spert-II programming environment runs on the host workstation and is built around the GNU tool set. The tool set includes an unmodified version of the `gcc` scalar optimizing C and C++ cross-compiler, and a version of the `gas` cross-assembler that we've extended to recognize vector assembler instructions and to perform instruction scheduling to avoid interlocks. Access to the vector unit is provided through an extensive set of libraries or by coding directly in assembler. A small kernel running on T0 implements the usual Unix operating system services by forwarding requests to a server process running on the host. T0 has no floating-point coprocessor, but the T0 kernel traps and emulates any MIPS-II floating-point instructions to simplify software porting.

To date, 25 Spert-II systems have been installed at 8 sites in the USA and Europe. For further information on T0 and Spert-II see [1, 5, 8].

4 Mapping Kohonen Nets to T0

We have implemented two vectorized library routines for KSOFMs. These are both coded in assembler, but provide a standard C function interface. The routines have been designed to support arbitrary neural map topologies and arbitrary neighborhood functions. In addition, we have coded a full implementation of Kohonen net training for a speech coding benchmark provided by EPFL.

4.1 Forward Pass

The first routine implements (1) and takes an input vector and a weight matrix and finds the neuron with the minimum squared Euclidean distance from the input vector, using the following C interface:

```
int forward(size_t n_inputs, size_t n_neurons,
            const short* X, const short* W, int rshift,
            size_t* c, int* sum)
```

The weight matrix `W[n_inputs][n_neurons]` is arranged with the weights for one neuron in one column of the matrix (an "input-major" layout) to allow the forward pass to use unit stride when accessing the weights. The weights and the input vector, `X[n_inputs]`, are represented as 16-bit values. The routine returns the index of the winning neuron in the location pointed to by the `c` argument, and a 32-bit representation of the squared Euclidean distance in the location pointed to by the `sum` argument. This routine ignores the neural map topology; 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.

The routine has two nested loops, an outer loop over groups of 32 weight matrix columns, and an inner loop that traverses the weights down a column. The last iteration of the outer loop can process less than 32 columns by setting the vector length register. In each iteration of the inner loop, up to 32 16-bit weights are loaded from one row of the matrix into a vector register using a unit stride load. A single 16-bit input vector element is read into a scalar register, then subtracted from the weights' vector register. The difference of two signed 16-bit numbers requires 17 bits to be represented completely, but the T0 multiplier takes only 16-bit inputs. The subtract operation configures the arithmetic pipeline to shift the result one bit to the right, rounding off the low order bit with jamming. The 16-bit rounded difference is then squared using the multiplier, yielding a 32-bit result. This multiplier result must be rounded down to leave headroom for the subsequent accumulation. The `rshift` parameter to the routine specifies how many bits to round off the multiplier result. This rounding is performed using round-to-nearest-even as part of the multiply instruction. The rounded multiplier results are accumulated using a 32-bit saturating addition. Usually, the `rshift` parameter will be determined by the input dimension, and is set to the smallest value that guarantees no overflow during accumulation. T0 has a saturation flag register with sticky bits that keep track of any saturations, and the `forward` routine uses these to return a boolean value indicating if saturation occurred. The inner loop requires one scalar load, one unit stride vector load, and three vector arithmetic operations, subtract, square and sum, to compute up to 32 connections. The squaring operation can only be performed with the multiplier in VP0, so the inner loop is unrolled to perform two weight matrix rows at a time to create an even number of vector arithmetic operations per loop iteration. This allows optimal scheduling of the loop across the two vector arithmetic units. The inner loop takes 12 cycles to compute up to 64 connections, giving an asymptotic peak of 213 MCPS.

To locate the winning neuron, the routine holds an index for each element in a separate vector register. These indices are initialized from memory at the start of the routine, and then incremented with 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. The inner loop completes with summed distances for up to 32 neurons in a vector register. These 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 using vector conditional move instructions.

At the end of the outer loop, the 32 global minimum sums and indices must be reduced down to a single minimum sum and index. Torrent provides a vector extract instruction that can move a sub-vector from the middle of one vector register to the start of another. Associative reductions are coded by extracting the last half of a vector, combining this with the first half using the appropriate associative operator, e.g., sum or minimum, then halving the vector length and repeating until the reduction is complete. On T0, the vector extract instruction executes in VMP to allow vector arithmetic instructions to be overlapped and the complete sum and index reduction operation takes 45 cycles.

Table 1 gives the measured performance of the forward pass routine on T0 for various sized networks taken from [3].

4.2 Weight Update

The second routine implements (2) with the following C interface:

```
int update(size_t n_inputs, size_t n_neurons, size_t stride,
           const short* X, const short* F,
           int rshift, short* W)
```

The routine modifies weights for `n_neurons` neurons located in contiguous columns of the weight matrix starting at the location pointed to by `W`. The `stride` argument gives the total number of neurons in the array, and hence the

Table 1. Spert-II KSOFM forward pass performance on real-world networks taken from [3].

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

element stride between rows in the matrix. The input vector is `X[n_inputs]` as before. The `F[n_neurons]` array holds 16-bit coefficients representing the value of the update factor $\alpha(t) \cdot \Lambda(t, d(i, c(t)))$ for each neuron. Depending on the topology of the neural map, the `update` routine might be called multiple times to update all neurons in the neighborhood of the winner. For example, with a 1-D grid it need only be called once, but with a 2-D grid it would be called once for each row in the neighborhood.

There are two nested loops in the routine. The outer loop processes the input vector 32 elements at a time, loading up to 32 16-bit elements of the input into a vector register. The inner loop operates on one neuron at a time, updating all weights connected to the inputs loaded in the outer loop. The 16-bit neuron weights are accessed using a strided vector memory load instruction. The weights are subtracted from the input vector elements to find the element distances, with jamming performed as in the forward pass to round the difference back to a 16-bit value. The 16-bit update factor for this neuron is loaded into a scalar register and multiplied by the difference to give a 32-bit result. This result is rounded down by the number of bits given in `rshift` using round-to-nearest-even rounding before being added in to the original weight vector using a 16-bit saturating addition. The weight vector is finally written back to memory with a strided vector store.

For each group of up to 32 weights, the inner loop performs 1 scalar load, a strided vector load and store, a vector subtract, a vector multiply, and a vector add. The time is dominated by the strided memory operations which require one cycle to transfer each element, but all other operations can be overlapped with the memory transfers, giving a peak rate of $2N + 1$ cycles to update N weights. This rate represents only 9.2% of peak arithmetic performance, but fortunately training neighborhoods shrink rapidly in practice, and so only a small fraction of the neurons need to be updated.

4.3 Speech Coding Benchmark

To measure training performance, we used a benchmark supplied by EPFL, Switzerland [4], which uses a KSOFM to implement speech coding by vector quantization. This benchmark has 12-dimensional input vectors mapped to a 2-dimensional neuron grid of varying size. A set of 30,000 training patterns were supplied.

The global adaption rate function is:

$$\alpha(t) = \frac{\alpha_0}{1 + K_\alpha \cdot t} \tag{3}$$

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:

$$R(t) = 1 + \frac{R_0}{1 + K_R \cdot t} \tag{4}$$

where R_0 defines the initial radius, and K_R is a time constant that controls the rate at which the radius shrinks. The neighborhood function, Λ, drops linearly, from 1 at the winning neuron to 0 for neurons outside radius $R(t)$. The distance function, $d(t, c(t))$, is the Manhattan distance on the 2-D grid.

The Spert-II implementation of this benchmark is written in C++, with calls to the two assembler functions. The adaption rate parameter calculations are performed in software-emulated floating-point, converted to 16-bit fixed-point, and then cached in a 2-D array of neighborhood adaption factors. The conversion keeps track of the fixed-point exponent of the maximum adaption factor to set the `rshift` parameter to the `update` routine appropriately. Computing new adaption rate parameters for every pattern is time consuming. Fortunately, updating the parameters more slowly seems to have little effect on convergence. For the timings below, the adaption rate parameters were updated after every 100 patterns.

For each pattern that is presented, it is necessary to map from the winning neuron index to 2-D neural map coordinates. This mapping is pre-computed and held in lookup tables. The C++ code manages the clipping of the neighborhood at the edges of the 2-D map, and calls the `update` routine with the appropriate pointers into the weight and cached factor matrices.

EPFL supplies the training data already converted to a 16-bit fixed-point representation. We have found no significant difference between floating-point and fixed-point trained nets in either convergence rate or final quantization distortion across a range of training parameters.

Table 2 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. Table 3 compares Spert-II performance with other reported implementations of this task [3, 9]. On larger networks and with larger training sets, the performance will asymptotically approach the peak forward pass performance, yielding up to 213 MCUPS.

Table 2. 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 initial radius, R_0, set to half the grid dimensions. The other training parameters are $\alpha_0 = 0.1$, $K_\alpha = 0.0025$, and $K_R = 0.02$.

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. Reported performance numbers for KSOFM [3, 9]. [a]Mantra I runs a smaller problem (6×10) due to memory limitations. [b]RENNS numbers are estimated based on [3], with the 100 neuron curve used for the benchmark figure, and the 10,000 neuron curve for the peak.

System	Clock	Measured 10x10 benchmark		Estimated peak	
	(MHz)	Number PEs	(MCUPS)	Number PEs	(MCUPS)
Sparcstation-20/51	50	1	3.46	-	-
Cray T3D	150	25	4.75	256	435
CNAPS (EPFL)	20	100	4.50	512	23
CNAPS (ASI)	20	-	-	256	54
Mantra I[a]	8	400	0.47	400	20
RENNES[b]	32	15	< 4.00	15	< 15
Spert-II	40	1	45.20	1	213

5 Summary

We have presented a Kohonen net implementation for the Spert-II vector microprocessor system. The implementation is built around two vectorized routines that can be used with arbitrary neural maps and neighborhood functions. The vectorization technique described here should provide good performance on other vector machines, including traditional vector supercomputers.

The two vectorized routines were used to build a complete training application used as a benchmark. This implementation is significantly faster than other systems for this small benchmark task, which represents the small networks in current use. Unlike competing neuro-computers, Spert-II achieves this high efficiency without sacrificing generality or a convenient programming environment.

Performance on larger networks is also important, as faster hardware makes it more practical to simulate larger networks and could encourage new applications. Spert-II's predicted peak performance on large networks is surpassed only by a large multi-million dollar MPP system.

6 Acknowledgements

Thanks to Paolo Ienne and Thierry Cornu for providing the speech coding benchmark, and to Jeff Bilmes for comments on an earlier draft of the paper. 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.

References

1. Wawrzynek, J., Asanović, K., Kingsbury, B. E. D., Beck, J., Johnson, D., Morgan, N.: Spert-II: A vector microprocessor system, *IEEE Computer*, **29**(3):79–86, March 1996.
2. Kohonen, T.: Self-organizing formation of topologically correct feature maps, *Biological Cybernetics*, **43**(1):59–69, 1982.
3. Myklebust, G., Solheim, J. G.: Parallel self-organizing maps for actual applications, *Proceedings of the IEEE International Conference on Neural Networks*, Perth, 1995.
4. Cornu, T., Ienne, P.: Performance of digital neuro-computers, *Proceedings Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, September 1994, 87–93.
5. Asanović, K., Johnson, D.: Torrent architecture manual, Technical Report, Computer Science Division, University of California at Berkeley, CSD-97-930, 1997.
6. Kane, G.: MIPS RISC Architecture (R2000/R3000), Prentice Hall, 1989.
7. Russel, R. M.: The CRAY-1 computer system, *Communications of the ACM*, **21**(1):63–72, January 1978.
8. Asanović, K., Beck, J.: T0 engineering data, Technical Report, Computer Science Division, University of California at Berkeley, CSD-97-931, 1997.
9. Ienne, P., Cornu, T., Kuhn, G.: Special-purpose digital hardware for neural networks: An architectural survey, *Journal of VLSI Signal Processing*, **13**(1):5–25, 1996.

This article was processed using the LaTeX macro package with LLNCS style