INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

A Performance Analysis of CNS-1* on Sparse Connectionist Networks

Silvia M. Müller† and Benedict Gomes‡

TR-94-009

February 1994

Abstract

This report deals with the efficient mapping of sparse neural networks on CNS-1. We develop parallel vector code for an idealized sparse network and determine its performance under three memory systems. We use the code to evaluate the memory systems (one of which will be implemented in the prototype), and to pinpoint bottlenecks in the current CNS-1 design.

*The CNS-1 project is a collaboration of the University of California at Berkeley and the International Computer Science Institute

†ICSI and CS Department, University of Saarland, Germany. E-mail: smueller@cs.uni-sb.de

‡ICSI and CS Division, U.C. Berkeley. E-mail: gomes@icsi.berkeley.edu

Contents

Chapter 1

Introduction

The Connectionist Network Supercomputer (CNS-1) [ABC+93], currently under development at ICSI, is a massively parallel multicomputer, with custom-designed fixed-point vector processor nodes. In [Mül93b], the performance of CNS-1 on densely connected multi-layer backpropagation networks has been analyzed in detail. Though dense backpropagation networks are by far the most popular neural networks today, the field of connectionism is extremely diverse and models with sparse connectivity are gaining in popularity. In fact, the typical target problem [ABC+93] for CNS is the simulation of a network of a million units, each with a thousand connections (a connectivity of 0.001). In this report, we will describe ways to represent such networks on the CNS-1 and analyze the resultant performance.

The network models we have chosen to analyze are simplified and do not correspond precisely to any particular sparse model. However, we hope to have captured important features of the computation of several different types of sparse models. We first analyze the performance of CNS-1 on a basic sparse network and then examine the impact of pipelining multiple examples through the network. Each network unit computes the dot product of its input activations with its 2 byte weights to produce a 1 byte output activation. An iteration consists of a single update of all the units on the machine. In a variant on the basic model, we show how the performance may be improved by pipelining multiple patterns through the machine. The models use different representations in order to achieve optimal performance.

The CNS-1 design has not yet been finalized. Several crucial decisions regarding the memory system (SRAM, SDRAM or RDRAM) and network will be made shortly. A second goal of this report is to provide information regarding the effect of these design choices on the performance of sparse connectionist models.

In Section 2 we briefly describe the basic characteristics of the CNS-1 in terms of its processor, memory and network performance, including the current design choices. We also introduce the benchmark workload and briefly discuss its relation to real world applications.

The following sections describe the basic sparse network (Section 3) and a variant on it - the pipelined sparse network (Section 4). The analysis of each of these networks is organized as follows. We first describe the parameters of the target problem and a reference instance. The reference instance will be used throughout the analysis to provide concrete performance figures. Then, under the assumption of an SRAM memory system we describe the representation, memory requirements, communication analysis, computation analysis and overall performance. These analyses and the results are then adapted to SDRAM and

RDRAM. Section 5 summarizes the performance of the CNS-1 on sparse networks.

Chapter 2

Machine Model

CNS-1 is a distributed memory multicomputer, consisting of fixed point vector processors embedded in a 2D mesh with wrap around in one dimension. The design can be scaled up to 1024 nodes. This report only considers the 128 node machine, because that will be the size of the prototype. Three aspects of the CNS-1 machine are relevant to the performance analysis: vector-processor speed, memory speed, and inter-processor communication speed. The following subsections summarize the design and performance of these components, and may be omitted by those familiar with the design or with [Mül93b].

2.1 Memory System

As it turns out, the performance of sparse networks on CNS is largely determined by the performance of the memory system. Three different memory systems are currently under consideration: a Static RAM (SRAM) system, a Synchronous DRAM (SDRAM) system, and an RDRAM system.

All three systems have an on-chip instruction cache, but our algorithms spend most of their time in loops. That keeps the run time impact of instruction cache misses very small. The hardware to support data accesses varies a lot in the three systems. This results in big differences in the data access time. For the performance of the memory systems, we therefore focus on data accesses but not an instruction cache misses.

2.1.1 SRAM System

The SRAM version of the CNS-1 would have no data cache. The scalar and the vector unit would both directly access the SRAM, and so the memory system would be similar to the T0 design [Asa93].

The instruction cache delivers one instruction per cycle but a cache miss takes 2 cycles. A busy memory (due to other cache traffic, for instance) causes an additional one cycle delay.

Transferring contiguous vectors of byte or halfword elements between the memory and the processor requires $\lceil VLR^1/8 \rceil$ cycles. Word transfers take twice as long, due to the SRAM

[1] Each vector register holds *VLR* elements; *VLR* is usually 32

speed. The link between the processor and the memory could actually transfer 16 bytes per cycle. Vector memory instructions with indexing or a stride only transfer one vector element per cycle regardless of whether the element is a byte, a half word or a word.

2.1.2 SDRAM System

The SDRAM version of the CNS would have an instruction cache and a $128 \times 32\text{B}$[2] data cache. The scalar unit would control the data cache but the vector unit could also access the cache. The SDRAM [SAM93, Bre93] itself would have a single active page of 8KB.

A page break requires an extra $t_p = 2$ cycle latency and there is a 2 to 4 cycle penalty for unaligned writes. An instruction cache miss takes 4 cycles. Additional memory traffic and page faults may increase it to 8 cycles. The data cache supports byte, halfword and word mode and delivers up to 8 words per cycle. Cache misses take 8 cycles on a clean line. For a dirty line, between 4 to 6 extra cycles are needed.

Vector memory operations have an extra one cycle latency, but otherwise take $\lceil VLR/8 \rceil$ cycles to read or write byte or halfword contiguous vectors. Word accesses still take twice as long. Vector memory instructions with indexing only transfer two vector elements in three cycles.

2.1.3 RDRAM System

The RDRAM [Ram92, Tos92] memory hierarchy would at least have three levels: separate instruction and data cache (4KB) on the processor chip, a second level cache of $(2 \times 2\text{KB})$ in each RDRAM chip, and the RDRAM itself. All the caches are direct mapped.

The RDRAM memory consists of four banks, each connected to its own memory interface. Each interface controls four RDRAM chips. The ports are interleaved every 32 bytes. Therefore, neighboring 32 byte blocks are always addressed via different ports. Every fourth block is stored in the same port but only every sixteenth block is stored in the same memory chip. The four ports work in parallel.

It is possible for the RDRAM system to run at a multiple of the processor speed, which allows the designer to slow down the CPU while maintaining the same memory bandwidth. Since the run time analysis for RDRAM is very complex, we only consider an RDRAM system running at twice the CPU speed.

The data access time depends on where the data is located in the memory hierarchy (Table 2.1). Vector memory accesses with indexing basically behave like *VLR* scalar accesses. An instruction cache miss takes between 20 and 45 cycles.

2.1.4 Comparing the Memory Systems

The final choice of memory system will depend on many factors such as availability, cost and physical considerations. The DRAM systems would be faster, cheaper and would have a bigger memory capacity, but the design would also be more complex. The SRAM system could probably run at 50 MHz, while SDRAM could go up to 75 MHz and RDRAM even up to 125 MHz.

[2] In this paper B designates bytes, and b designates bits

Vector Length	Size [Byte]	Read			Write		
		Hit	Miss		Hit	Miss	
			clean	dirty		clean	dirty
VLR	1, 2, 4	14	22	28	10	16	22
1	1, 2	6	15	20	2	9	14
1	4	7	15	21	3	9	15

Table 2.1: Time required for accessing contiguous vectors and scalars in the RDRAM memory system [cycles]

2.2 Processor Performance

Individual processors consist of a scalar MIPS core and a fixed-point vector coprocessor. The actual processor speed will depend on the VLSI process available and on the memory structure, but should be in the neighborhood of 50-75 MHz.

All scalar instructions are assumed to take a single cycle. [ABC+93] contains details of the vector coprocessor architecture. We merely note that there are two vector arithmetic pipes, permitting an addition and a multiplication to proceed in parallel. In general, vector arithmetic operations take $\lceil VLR/8 \rceil$ cycles, and may be fully overlapped with memory operations. In most cases arithmetic may be completely hidden behind memory accesses, particularly with indexed memory accesses which take at least 32 cycles for a 32 element vector.

2.3 Network Performance

The CNS-1 network design has not yet been finalized but it will be based on active messages [CSS+91, vCGS92]. [ABC+93, Mül93b, AC93] describe some basics of the network interface and give a timing model. We only consider data exchanges between nearest neighbors, resulting in a slightly simpler model.

2.3.1 Nearest Neighbor Communication

The network will have a bandwidth of about 125 MB/s per link, though the design team is currently shooting for a bandwidth of 250 MB/s. Each node can send and receive b bytes per cycle. The concrete value of b depends on the system cycle time and has to be specified for all our systems. A message can contain the contents of one vector register some scalars and nine header bytes. We omit the scalars for simplicity's sake, so a message transfers up to $4VLR$ data bytes.

The transfer time occurs partially in the CPU and partially in the network. The CPU executes some overhead at the beginning and end of a transfer. During the remaining time, the network transfers the message and the CPU processes further instructions or idles.

Exchanging $n \leq 4VLR$ bytes keeps the network busy for

$$T_{net}(n) = \left(\frac{n+9}{b} + 1\right)$$

cycles. In addition to 21 cycles overhead, the CPU might also have to access and store the message data in main memory. These memory accesses take $T_{save}(n)$ cycles and the CPU would be busy for

$$T_{CPU}(n) = 21 + T_{save}(n)$$

cycles. The whole transfer time then adds up to $T^{trans}(n)$ cycles:

$$T^{trans}(n) = T_{net}(n) + T_{CPU}(n) = \left(\frac{n+9}{b} + 1\right) + 21 + T_{save}(n).$$

To exchange $n > 4VLR$ data bytes at once, the processors send $k(n)$ messages in a row, with $m(n)$ data bytes each:

$$k(n) = \left\lceil \frac{n}{4VLR} \right\rceil \quad , \quad m(n) = \left\lceil \frac{n}{k} \right\rceil .$$

CPU and network activities can largely be overlapped, and so transfer only requires the following time:

$$\begin{aligned} T^{trans}(n) &= (k(n)-1) * \max\{T_{net}(m(n)), T_{CPU}(m(n))\} \\ &\quad + T_{net}(m(n)) + T_{CPU}(m(n)) \\ &= \begin{cases} k(n)\,T_{net}(m(n)) + T_{CPU}(m(n)) & , \quad T_{net}(m(n)) \geq T_{CPU}(m(n)) \\ T_{net}(m(n)) + k(n)\,T_{CPU}(m(n)) & , \quad T_{net}(m(n)) < T_{CPU}(m(n)). \end{cases} \end{aligned}$$

2.3.2 Broadcast

Paper [Mül93a] analyzes all-to-all broadcast on the CNS-1 in all detail. The results show, that such a broadcast with n data bytes per processor requires the same time as a nearest neighbor communication with $(P-1)n$ bytes, at least for $n \geq 4VLR$:

$$T^{bcast}(n, P) = T^{trans}(n \cdot (P-1)).$$

2.4 General Workload Model

In order to analyze the performance of the CNS on a sparse network, we have chosen a simple workload model that captures the essential computation. In general we will bypass the top level issue of unit placement - since we assume uniform random connectivity, there is not much we can say about locality or placement. The units will be evenly divided among all the processors.

2.4.1 Connectionist Models

In general, connectionist networks are inspired by the neural circuitry of the brain. Connectionist networks vary in detail from complex models of dendritic trees to simplified mathematical abstractions that attempt to capture the structure of the computation without mimicking its details. We are interested in the latter class of models, which consist of networks of simple processing units with output activations, connected by weighted links. Neural models differ widely in how they compute their output activation, the nature of their connectivity structure and their overall computational model.

The output activation is a simple function of the local state of the unit, the activations on the input links to the unit and the weights on those links. Most commonly, the output activation is the dot product of the input activations and the weights, followed by a thresholding function. The computation of the dot product is the most expensive portion of the computation, since its execution time is proportional to the total number of connections. Applying the threshold function is usually much faster, because its time is only proportional to the number of units. In our model, each unit just computes the dot product of the output. No thresholding function is considered.

Connectivity structures vary from highly structured (sparse) networks with user defined weights (e.g. [Sha88]) to multi-layer backpropagation networks, which usually have complete connectivity between layers. Even in densely connected multi-layer networks, weight pruning techniques like Optimal Brain Damage ([LCDS90]) may be applied to yield multi-layered sparsely connected networks.

The overall computational model of a neural network is strongly related to the structure of the network. Simulating a neural network may be broken down into iterations. During each iteration some portion of the units in the network are stepped in some order. With multi-layer feedforward networks, each layer of the network is computed in turn, with activations flowing from the input of the network to the output. With nets with feedback connections (recurrent nets), a common method of simulation is to synchronously update all the units. Each unit sees as its inputs the output activations of the previous iteration. A variant on this method is asynchronous updating where a randomly chosen fraction of the units are updated at each iteration. In both cases, it is possible to send a whole set of activations through the network in pipelined fashion. This opens up an extra degree of parallelism that might be exploited.

In our model, the connections are uniformly randomly distributed over a set of input activations. This model can be interpreted as either a recurrent network with synchronous updating (where the input activations are the outputs from the previous iteration), or as a single layer of a sparsely connected multi-layer network (where the input activations are the

outputs from the previous layer). Random connectivity might be quite an appropriate model for pruned backprop networks. Structured sparse networks might exhibit more clustering structure, as in the model developed by ([GH88]). However, such a clustered model would be even harder to analyze.

Chapter 3

Basic Sparse Model

Our basic network consists of U units distributed over P processors, with $u = U/P$ units per processor. The probability that a unit has a connection from any other unit is p_c. Assuming a uniform distribution for p_c, the average number of connections per unit will be $c = p_c \cdot U$. The weights are 16 bits wide. However, with the exception of a small impact on the paging overhead, 8 bit weights only affect the memory requirements and not the computation time. The unit output activation is the dot product of the input activations and the weights. Each unit updates its 1-byte output every cycle.

3.1 Reference Case

Quantitative performance comparisons require concrete values for our parameters. For this purpose, we use a reference model consisting of half a million units ($N = 512K$), with a connectivity $p_c = 0.001$, running on a 128 processor machine. Each processor holds 4096 units, with an average of 512 input connections per unit. The problem size is chosen to fit in 16 MB of memory; the million unit problem mentioned in [ABC+93] would not. If the node memory turns out to be 8MB, the reference problems must be further reduced in size.

3.2 Representation

For each unit we need to represent its *output* activation and all its incoming connections. A connection consists of a *weight* and an *input pointer* to a input activation. All inputs are stored in the *input table* (see Figure 3.1), a U byte array of the output activations from the preceding iteration. Each processor has a copy of the whole table, so that all the inputs are locally available. Note that since the connectivity is random, the input pointers will be randomly distributed over the input table. Almost all the activations will be needed on all the processors.

The units are equally distributed among all the processors, with each processor computing $u = U/P$ outputs. Each processor stores its u outputs in an local array, the output table. Since the input activations are just the output activations from the previous iteration, updating the input tables on all the processors involves an all-to-all broadcast, as described in Section 3.5.

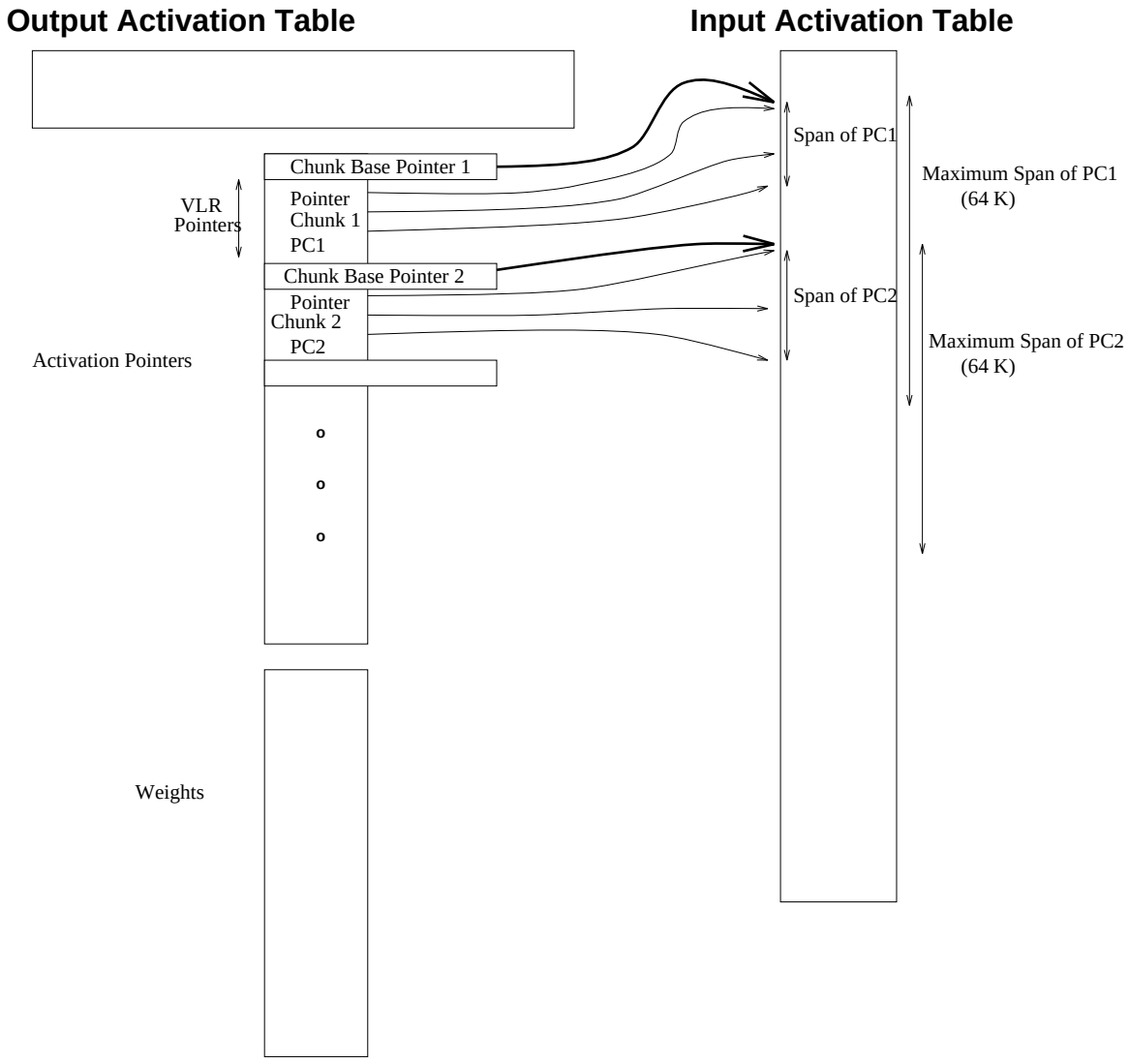

Figure 3.1: Basic Sparse Case: Pointer Chunk Representation. The representation of the input pointers and the weights for a single unit are shown

Since the neural network is sparse, each unit gets inputs from a small, random subset of the input table. Pointers into this table represent these inputs. The order of the inputs is irrelevant to the computation, and so the input pointers can be sorted according to the locations they point to. The weights corresponding to a unit's inputs are stored in a weight array.

Pointers into the usually large input table (0.5MB in the reference case), would need to be 4 bytes wide. A base/offset addressing scheme reduces the pointer size to two bytes and allow us to store 50% more connections in a given memory system. We can associate base addresses either with sections of the input table or with sections of a unit's input pointers. The *segmented input table* scheme associates base addresses with various segments of the input table. The *pointer chunks* scheme associates base addresses with 32 element chunks of each unit's connection pointers.

3.2.1 Segmented Input Table

We divide the input table into $U/64K$ segments (8 segments in the reference case) of $64K$ inputs, each with a different segment base address. A segmented input table requires that each unit has a *segment size table*. These tables indicate how many input activations reside in each of the $U/64K$ segments of the input table. When loading in a unit's input activations, the segment size table determines when to change the segment base address. For each unit, the segment size table with 2-byte entries will occupy $2 * U/64K$ bytes. For u units that adds up to $uU/32K$ bytes (64K in the reference case).

The problem with this approach is that a unit's input pointers are divided into irregular sized chunks. So, setting the vector length register for each segment and determining when to switch the segment base address will be quite complex. It is hard to analyze the time required for complex conditional code without actually writing the assembly. We therefore choose to analyze a simpler scheme that is only slightly more expensive.

3.2.2 Pointer Chunks

This scheme is based on a suggestion by A. Formella. The processors store a 4 byte offset with each set of VLR input pointers. A unit's connection pointers are partitioned into sets of VLR consecutive pointers, which we refer to as pointer chunks. A 4 byte chunk base address is associated with each pointer chunk; the pointers are offsets from that base address. The *span* of a pointer chunk is the difference between the first and last addresses pointed to by the given chunk. Using 2 byte pointers, a single pointer chunk can span upto 64K of the input table.

If a particular pointer chunk needs to span more than 64K, it may be split into two (or more) pointer chunks, padded with random pointers. The padding weights corresponding to these pointers must be zero, otherwise they would affect the result of the computation. In the reference case, such splitting should not occur often. The expected distance between consecutive input pointers is U/c and a pointer chunk of VLR pointers will span about $(U/c) * VLR$ bytes of the input table, that is 32KB in the reference case.

We use pointer chunks since the resulting algorithm is simpler and the computation is easier to analyze. The difference in performance and memory between the two schemes is not significant. In the reference case, the segmented input table approach stores 8 segment

sizes per unit, while the pointer chunk approach requires $c/VLR = 16$ base addresses. So the pointer chunk approach requires about 128KB more memory per processor than the other approach. That is less than 1% of the whole memory required.

3.3 Memory Requirements

A connections occupies 4 bytes, 2 for the weight and 2 for the connection pointer. The base address of a pointer chunk also occupies 4 bytes. On average, each network unit has c connections and requires $c/32$ base addresses. With $u = U/P$ units per processor, that adds up to $u(4c/32 + 4c)$ bytes. In addition, each processor holds the whole input table of U bytes. So, the memory requirement per processor is

$$M_{basic} = u(4c/32 + 4c) + U$$

bytes per processor. In the reference case, that equals $M_{basic} = 9MB$.

3.4 Per-processor Computation

The computation of a single units consist of the following actions:

1. For each of the unit's pointer segments:
   (a) Load in the base address for the pointer segment (scalar load, 1 cycle).
   (b) Load in the pointer segment (vector load, $\lceil VLR/8 \rceil$cycles)
   (c) Add the current chunk base pointer (see Figure 1) to all the offsets in the input pointer chunk (vector + scalar, $\lceil VLR/8 \rceil$cycles).
   (d) Load in a vector of weights for the currently available input activations (vector load, $\lceil VLR/8 \rceil$cycles).
   (e) Use the input pointers to load in the currently available inputs (indirect vector load, VLR cycles).
   (f) Compute the product of the input and weight vectors (vector * vector, $\lceil VLR/8 \rceil$cycles).
   (g) Accumulate the product of the input and weight vectors into an accumulation vector (vector + vector, $\lceil VLR/8 \rceil$cycles)
2. Reduce the accumulation vector (20 cycles, memory unusable)
3. Store the unit output (scalar write, 1 cycle)

We assume that arithmetic and memory operations can be overlapped appropriately. The total time taken is the maximum of the times required for arithmetic, for memory operations, and for scheduling the instructions. For one iteration of the inner loop, scheduling the instructions takes 7 cycles and memory operations $1 + 2\lceil VLR/8 \rceil + VLR = 41$ cycles. Since the vector unit can execute an multiplication and an addition in parallel, arithmetic operations only take $2\lceil VLR/8 \rceil = 8$ cycles, The memory operations clearly determine the overall speed. Processing the c connections of a single unit takes c/VLR iterations of the

inner loop. The final vector reduction[1] requires 20 cycles and storing the output 1 cycle. A single output activation can therefore be updated in $(c/VLR)(1+2\lceil VLR/8\rceil + VLR)+20+1$ cycles. Updating all u output activations on a processor will take

$$T_{basic}^{comp} = u\left(\frac{c}{VLR}\left(1+2\left\lceil\frac{VLR}{8}\right\rceil + VLR\right)+21\right) = u\left(\frac{41}{32}c+21\right) \text{ cycles.}$$

And so, in the reference case, the per-processor computation time requires 2.8 million cycles per iteration.

3.5 Communication

At the end of an iteration, each processor has to communicate its output activations to all processors. There are two ways to update the activations on each processor:

Selective Communication Each unit sends its activation to only those processors where it is required i.e. to those processors that have at least one connection from this unit. Since we have a sparsely connected net, it is possible that an activation may be needed on only a small number of other processors.

All-to-All Broadcast Each processor broadcasts all its output activations to all other processors.

The all-to-all broadcast is much simpler and enables dense data packaging. Selective communication eliminates any unnecessary transfers of activations but requires extra packaging and therefore higher overhead. To decide on the appropriate communication scheme, we first determine how many outputs need not be communicated between any pair of processors, P_A and P_B.

A single unit A on P_A need not be communicated to P_B if A is not connected to any unit on P_B. The probability that A is not connected to a given unit on P_B is $(1-p_c)$. So the probability that A is not connected to *any* of the u units on P_B is $(1-p_c)^u$. Taking the expectation of this probability over all the u units on P_A, we get the number of activations on P_A whose output need not be communicated to P_B, which is

$$u * (1-p_c)^u.$$

In the reference case, we can save less than 2% of the transfer between any pair of nodes. Such a low benefit would clearly not outweight the costs of packaging. Hence, we choose the simpler all-to-all broadcast. However, if the connectivity p_c is an order of magnitude lower, we could save the communication of about 66% of the activations. More complex issues, such as the time to package the required activations, will need to be analyzed in order to determine whether this makes selective communication preferable.

With all-to-all broadcast, every processor communicates its own set of u activations to every other processor. Since u is a multiple of $4VLR = 128$, the all-to-all broadcast takes

[1] Vector reduction makes use of the vext.v instruction which is implemented using the memory pipe hardware and hence cannot be hidden behind other memory operations.

time (section 2.3.2):

$$T^{bcast}(u,P) = \begin{cases} \frac{u(p-1)}{128} T_{net}(128) + T_{CPU}(128) & , \quad T_{net}(128) \geq T_{CPU}(128) \\ T_{net}(128) + \frac{u(p-1)}{128} T_{CPU}(128) & , \quad T_{net}(128) < T_{CPU}(128). \end{cases}$$

3.6 Overall SRAM Performance

The SRAM version of the CNS-1 will probably run at 50 MHz. A network link with a bandwidth of 125 MB/s therefore transfers $b = 2$ bytes per cycle, and transferring $n = 128$ bytes keeps the network busy for

$$T_{net}(128) = \left(\left\lceil \frac{128+9}{2} \right\rceil + 1 \right) = 70 \text{ cycles.}$$

The processor can load and store 128 bytes in 16 cycles. Consequently, a $n = 128B$ transfer spends only

$$T_{CPU}(128) = T_{save}(128) + 21 = 37 \text{ cycles}$$

in the CPU, much less than in the network. On the SRAM-system, the all-to-all broadcast therefore requires only

$$T^{comm}_{basic} = T^{bcast}(u,P) = (P-1) \left\lceil \frac{u}{128} \right\rceil 70 + 37 \approx 69.5u \text{ cycles.}$$

In our reference case this takes less than 285 thousand cycles. This is about an order of magnitude less than the 2.8 million cycles taken for computation.

The total time taken for the communication and computation of a single iteration is

$$\begin{aligned} T^{SRAM}_{basic} &= T^{comp}_{basic} + T^{comm}_{basic} \\ &= u(41c/32 + 21) + 69.5u \text{ cycles.} \end{aligned}$$

That is about $3.06M$ cycles in the reference case, which has 268 M connections i.e. about 88 connections per cycle.

3.7 Performance with SDRAM

The SDRAM performance is identical to that of the SRAM, except for three facts. First, vector memory instructions with indexing now take 1.5 cycles per element on average. Second, page faults in the SDRAM memory slow the execution down, and third, the clock speed might be slightly higher, around $50 - 75$ MHz. The network link still transfers $b = 2$ bytes per cycle.

3.7.1 Indexed Memory Accesses

During one iteration there occur cu/VLR vector memory loads with indexing. They require cu cycles on the SRAM system but $1.5cu$ cycles on the SDRAM system. The computation time therefore increases by

$$\Delta^{index}_{basic} = 0.5cu \text{ cycles.}$$

3.7.2 Page Faults

Recall that SDRAM has an active page of 8KB, with a page fault penalty of $t_p = 2$ cycles. Page breaks occur when shifting between a unit's memory and the input table and also during the indirect loading of inputs.

Without Data Cache

We assume that the unit's input pointers and weights (2K in the reference case) may all reside on a single page and no page break occurs in moving from a unit's weights to its input pointers. A page break occurs every time we switch from the input table to the unit's pointers and weights i.e. at the beginning of each pointer chunk, or c/VLR times per unit.

Accessing a unit's inputs from the input table causes additional page breaks, since the input table is spread over $U/8KB$ pages. There occurs at least one fault per page, but switching between the input table and the weights can cause up to c/VLR pages to be loaded twice. However, there occurs at most one page break per input.

Each processor computes u units. Page brakes consequently increase the computation time by

$$T_{basic}^{page} \leq t_p \frac{cu}{VLR} + t_p u \min\left\{c, \frac{U}{8K} + \frac{c}{VLR}\right\}$$

cycles per processor per iteration. This is only an upper bound and includes the faults due to switching between unit's memory and input table. In the reference case, a processor spends at most 786 thousand cycles in page breaks.

With Data Cache

The 4K data cache is large enough to store 1024 connections (weights and input pointers) at a time. Loading the data cache at the rate of 16 bytes per cycle takes $4K/16 = 256$ cycles. Each processor holds cu connections, and therefore spends $256/1024cu = cu/4$ cycles moving the connections into the data cache.

Since the processors now load the connections from main memory in blocks of 1024 elements, there occur only $cu/1024$ page breaks during these loads instead of c/VLR. While accessing the inputs, the data cache decreases the number of pages which are loaded twice. That saves at most cu/VLR page faults. Altogether, the data cache saves less than $cu/16$ page breaks or $t_p cu/16$ cycles, but requires twice as much cycles for loading the cache. We therefore choose the version without data cache.

Without using a data cache, the computation time per iteration then adds up to

$$T_{basic}^{comp} = u\left(\frac{41}{32}c + 21\right) + \Delta_{basic}^{index} + T_{basic}^{page}$$

cycles. In the reference case T_{basic}^{comp} is about 4.6 million cycles.

3.7.3 Transfer Time

Transferring $n = 128$ bytes still spends $T_{net} = 70$ cycles in the network. However, SDRAM increases the memory access time. Loading and storing $n = 128$ bytes basically requires 16

cycles, but the SDRAM adds one cycle delay per memory access, and a page fault occurs after every $8K/128 = 64$ transfers. So, each 128 byte transfer is responsible for 1/64 of a page fault and therefore spends

$$T_{CPU}(n) = T_{save}(128) + 21 = 16 + 2 + \frac{t_p}{64} + 21 \approx 39.03 \text{ cycles}$$

in the CPU, still less than the time spent in the network. The all-to-all broadcast therefore requires

$$T_{basic}^{comm} = (P-1)\lceil u/128 \rceil 70 + 39.03 \approx 69.5u \text{ cycles.}$$

3.7.4 Overall Performance

The total time for one iteration of a basic sparse network on SDRAM is the sum of communication and computation time:

$$T_{basic}^{SDRAM} = T_{basic}^{comm} + T_{basic}^{comp}$$

In the reference case, this is about 4.76M cycles for one iteration of 268M connections, or about 56 connections per cycle. The SDRAM system would have to run 1.66 times faster than the SRAM system to achieve the same performance.

3.8 Performance with RDRAM

If CNS-1 uses RDRAM, the neural network would still be represented in the same manner with the same memory requirements. However, caching effects will have a bigger impact on the machine performance.

3.8.1 Per-processor Computation

Each processor loads 1024 weights and input pointers in the 4K data cache and the $1024/VLR = 32$ corresponding pointer base addresses in a vector register. In the reference case, this corresponds to storing 8 units in the cache. The processor loads the inputs from the input table in main memory and computes the dot product as before. The output activations are computed and stored in a vector register. The processor writes the outputs back to main memory in 32 element blocks. We first consider the time taken to load the weights and input pointers and then the time taken to load the inputs from the input table.

32 vector loads are necessary to move a 4KB block in the data cache. The first 4 loads, each going to a different memory bank, will be clean RDRAM cache misses. They take 22 cycles each. The remaining 28 loads are hits and take only 14 cycles each. Loading the base address vector requires 22 cycles. The CPU spends a further 32 cycles extracting the 32 base addresses from the base address vector register. Accessing weights and pointers in the data cache requires additional $2 \cdot 1024/VLR \cdot 4 = 256$ cycles. Consequently, each processor spends

$$T_{weight} = \frac{uc}{1024}(4 \cdot 22 + 28 \cdot 14 + 22 + 32 + 256) = \frac{790}{1024}uc$$

cycles in loading weights and input pointers per iteration.

Indexed vector memory accesses basically behave like VLR scalar accesses. A processor could overlap up to 4 scalar accesses (one per memory port), but on average the processor can only keep 3 memory ports busy. The RDRAM chips have a 2KB cache line. In a 2K cache line holding a segment of the input table, we expect a unit to access $2K \cdot p_c = 2$ activations; the first access is a clean miss, the other a hit. Since the processor may overlap misses and hits, we assume that all indexed vector loads are clean misses and therefore take $15\lceil VLR/3 \rceil = 165$ cycles. Loading the activations then requires

$$T_{index} = 165\frac{uc}{VLR} = \frac{165}{32}uc$$

cycles per processor and iteration.

Reducing the output activation requires 20 cycles per unit and saving the output to main memory adds a further 16 cycles (clean write miss) per 32 units. The writes actually cause subsequent load misses to be dirty, but this is only a minor affect which we omit in our analysis.

Since memory accesses require even more cycles than in the SRAM system, it is even easier to hide arithmetic computation and instruction scheduling. The per-processor computation time therefore adds up to

$$T^{comp}_{basic} = T_{weight} + T_{index} + 20.5u \approx (5.93c + 20.5)u \text{ cycles}$$

or about 12.5 million cycles in the reference case.

3.8.2 Transfer Time

The RDRAM version of the CNS-1 was originally [ABC+93] expected to run at 125MHz, but $50 - 75$ MHz seems to be more realistic. A network link then transfers $b = 2$ bytes per cycle, but with the faster clock only $b = 1$ byte per cycle is possible, assuming the same network bandwidth. A 128-byte transfer keeps the network busy for $T_{net}(128)$ cycles:

$$T_{net}(128) = \begin{cases} 137/1 + 1 = 138 & , \quad b = 1 \\ \lceil 137/2 \rceil + 1 = 70 & , \quad b = 2. \end{cases}$$

Loading and storing 128 bytes requires 44 cycles, 16 cycles for a write with a clean RDRAM cache miss, and 28 cycles for a read with dirty miss. With the usual overhead of 21 cycles, a 128B transfer spends $T_{CPU} = 44 + 21 = 65$ cycles in the CPU. The all-to-all broadcast therefore requires

$$T^{comm}_{basic} = (P-1)\lceil u/128 \rceil \, T_{net}(128) + 65 \text{ cycles}.$$

3.8.3 Overall Performance

The total time for one iteration of a basic sparse network on RDRAM is

$$T^{RDRAM}_{basic} = T^{comm}_{basic} + T^{comp}_{basic} \approx \begin{cases} (5.93c + 20.5)u + 136.9u & , \quad b = 1 \\ (5.93c + 20.5)u + 69.5u & , \quad b = 2 \end{cases}$$

cycles. In the reference case, this is between 12.8 and 13.1 million cycles for one iteration of 268 million connections, or 20.5 and 21 connections per cycle.

Memory System	SRAM	SDRAM	RDRAM		
Cycle Time	20	20	20	8	ns
T_{basic}^{comp}	2772992	4608000	12515328	12515328	cycles
T_{basic}^{comm}	284517	284540	284545	560897	
T_{basic}	3057509	4892520	12799873	13076225	
Absolute Performance	4.37	2.74	1.05	2.57	GCPS
Relative to ALU Peak	19.8	12.4	4.7	4.6	%
Relative to System Peak	100.0	62.5	23.9	25.5	%
Relative to Indexed Peak	40.0	25.0	9.6	11.5	%

Table 3.1: Run time and performance overview of the reference case ($u = 4096$, $c = 512$, $p = 128$)

3.9 Performance of CNS-1 on the Basic Sparse Case

Table 3.1 summarizes the run time and performance of the CNS-1 under various memory systems. The values are for the basic sparse case with reference problem size. Comparing the absolute performance of the three CNS-1 versions – all at the same cycle time – shows that the SRAM version is best suited for this application, but the absolute performances are not sufficient to determine bottlenecks in the different CNS designs. We therefore also determine their performance relative to the ALU, System and Indexed peak performance.

3.9.1 Peak Performance

The *ALU peak performance* only takes arithmetic operations into account, and ignores all memory effects. This performance can only be achieved when working on-chip. The *System peak performance* is more realistic. It counts arithmetic instructions, memory accesses and transfer, but it assumes an ideal memory system. The ideal memory system delivers to the bandwidth of the memory interface; indexed accesses still take one cycle per element. The *Indexed peak performance* is similar to the System peak performance but the ideal memory system then supports indexed memory accesses. Indexed vector loads are then executed as fast as simple vector loads.

ALU Peak Performance

During each iteration, a processor performs two additions and one multiplication for each of its connections. Since the vector unit has two arithmetic pipelines, that only requires

$2\lceil VLR/8 \rceil uc/VLR = uc/4$ cycles. Reducing its output vector keeps the arithmetic unit busy for addition 20 cycles. When only considering arithmetic operations, one iteration therefore requires

$$T_{basic}^{ALU} = \frac{uc}{4} + 20u$$

cycles. In the reference case, that adds up to 606.2 thousand cycles. With a cycle time of $tc = 20ns$ that equal 22 GCPS.

System Peak Performance

The SRAM behaves like an ideal memory, except when accessing 4 byte data. That only happens during the communication. Then, $T_{CPU}(128)$ takes 29 cycles instead of 37; 21 cycles for communication overhead and $2 * 4$ cycles for loading and storing the messages. That reduces the SRAM communication time

$$T_{basic}^{comm} = (p-1)\frac{u}{128}T_{net}(128) + T_{CPU}(128)$$

by 8 cycles. A faster cycle time t_c (8 ns instead of 20 ns) increases the time $T_{net}(128)$ from 70 cycles to 138 cycles. Consequently:

$$T_{basic}^{System} = T_{basic}^{SRAM} + \begin{cases} -8 & , \quad t_c = 20ns \\ \frac{138-70}{128}(p-1)u - 8 & , \quad t_c = 8ns \end{cases}$$

The CNS-1 would have a System peak performance of 4.39 (10) GCPS on the basic sparse model with a cycle time of $tc = 20ns$ $(8ns)$.

Indexed Peak Performance

In contradiction to the previous case, we assume a memory system which supports indexed vector loads. Indexed vector loads only require 4 cycles instead of VLR cycles. This feature saves $(VLR-4)cu/VLR$ cycles per iteration. The Indexed peak execution time can therefore be expresses by:

$$T_{basic}^{Indexed} = T_{basic}^{System} - \frac{VLR-4}{VLR}cu.$$

That equals an Indexed peak performance of 10.98 to 22.4 GCPS, depending on the cycle time.

3.9.2 Results

The SRAM system almost achieves the performance of the ideal memory system. To achieve the same performance, the SDRAM system would have to run 1.66 times faster than the SRAM version, that is around 83MHz. The fastest RDRAM system running at 125MHz — that is 2.5 times faster than SRAM — does not even come close to the SRAM performance. DRAMs are designed for fast block accesses to the disadvantage of the random access time. This reduces the performance of the DRAM systems, especially on an application with many indexed vector loads. These performance results definitely speak for the SRAM

system, which is also simpler to implement and to program. So far, bigger memory capacity is the only advantage of the DRAM systems.

The SRAM system reaches the System peak performance, but there is a 80% performance loss when compared to the ALU peak performance. Only 60% of this loss is due to indexed memory accesses, as the performance figures relative to the indexed peak performance indicate. This indicates that the CNS has a bottleneck in the memory interface. The CPU consumes and produces more data than the memory interface can load and store, especially with indexed vector accesses. A single iteration requires 1.4 (2.2, 6.1) cycles per connection on the SRAM (SDRAM, RDRAM) system. The indirect load already takes 1 (1.5, 5.2) cycle per connection.

It is hard to circumvent this memory bottleneck in hardware. In the next section, we therefore analyze a modification of our algorithm which is less sensitive to this bottleneck.

Chapter 4

Sparse Pipelined Model

Pipelining the execution of d sets of inputs allows us to load in vectors of inputs, thus reducing the impact of the indirect load on the performance (pointed out by S. Omohundro). We are *not* pipelining multiple iterations of the execution of the same input set. Rather, we are pipelining the execution of d different input sets. Recurrence is still permitted: every dth iteration corresponds to the same input set. Another way of looking at this problem is that it is identical to the original sparse problem, except that the input and output activations are now vectors rather than scalars.

4.1 Reference Pipelined Model

For the reference case we assume the same parameter values as we had in the reference basic case, along with a pipelining depth of $d = 32$. In fact, this value of d makes most sense: with a larger pipelining depth we would run out of memory and a smaller depth would result in the same run time but for fewer connections (see Section 4.5).

4.2 Representation

Pipelining to a depth of $d \leq VLR$ increases the size of the input table by the same factor. An input table of half a million inputs pipelined to a depth of 32 requires 16MB, i.e. all the available memory. We solve this problem by distributing the input table over the P processors instead of duplicating it. The representation is then similar to the segmented input table approach that we described for the basic sparse case (see Section 3.2.1).

The input table is split in P chunks, the *input table blocks*. At any given time, each processor only holds x of these blocks. They are stored in the *input table segment*. An appropriate value for x is determined later on. The current input segment of two neighboring processors only differs by one block. That results in different segmentations of the input table (see Figure 4.1).

A processor still represents its outputs in an output table, but now each output is a d element *output vector* (see Figure 4.2). On a processor, each unit's connections – weights and input pointers – are divided in P/x *connection segments*, corresponding to the input table segment. All the pointers of one connection segment point to a single segment of

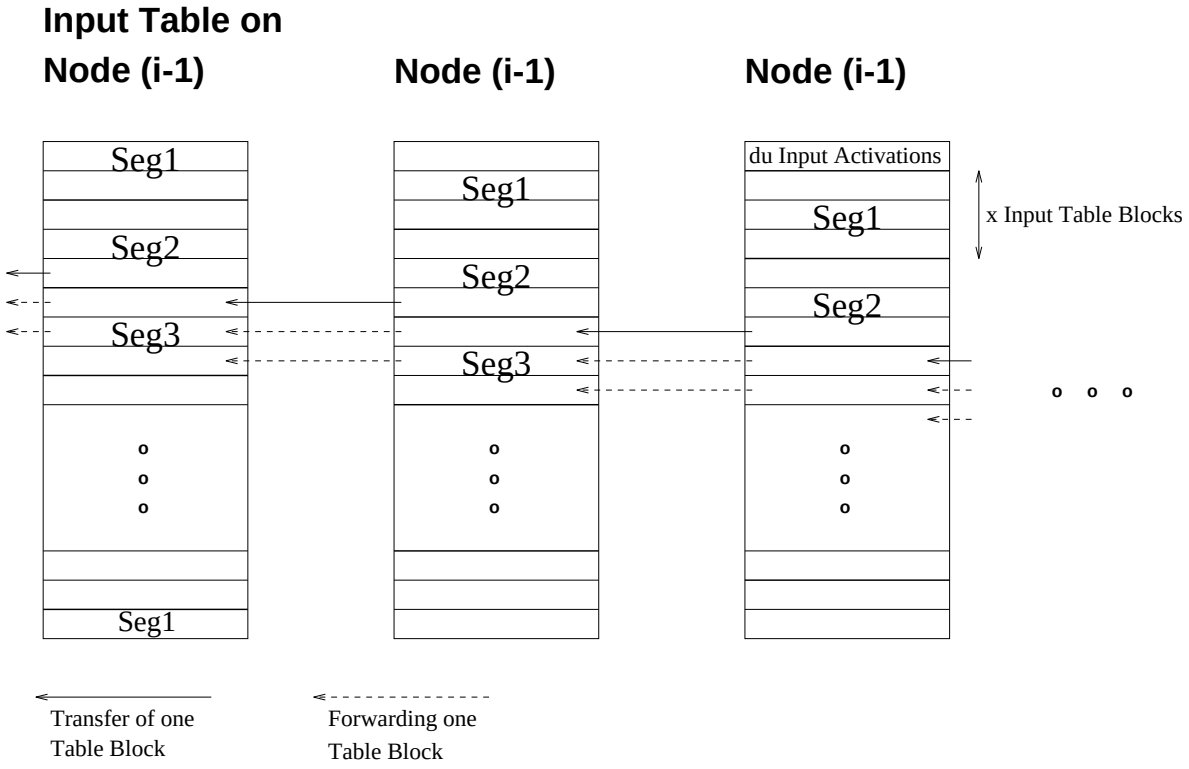

Figure 4.1: Segmentation of the input table for three neighbored nodes, $x = 3$. The arrows indicate transfer after processing the second table segment (Seg2)

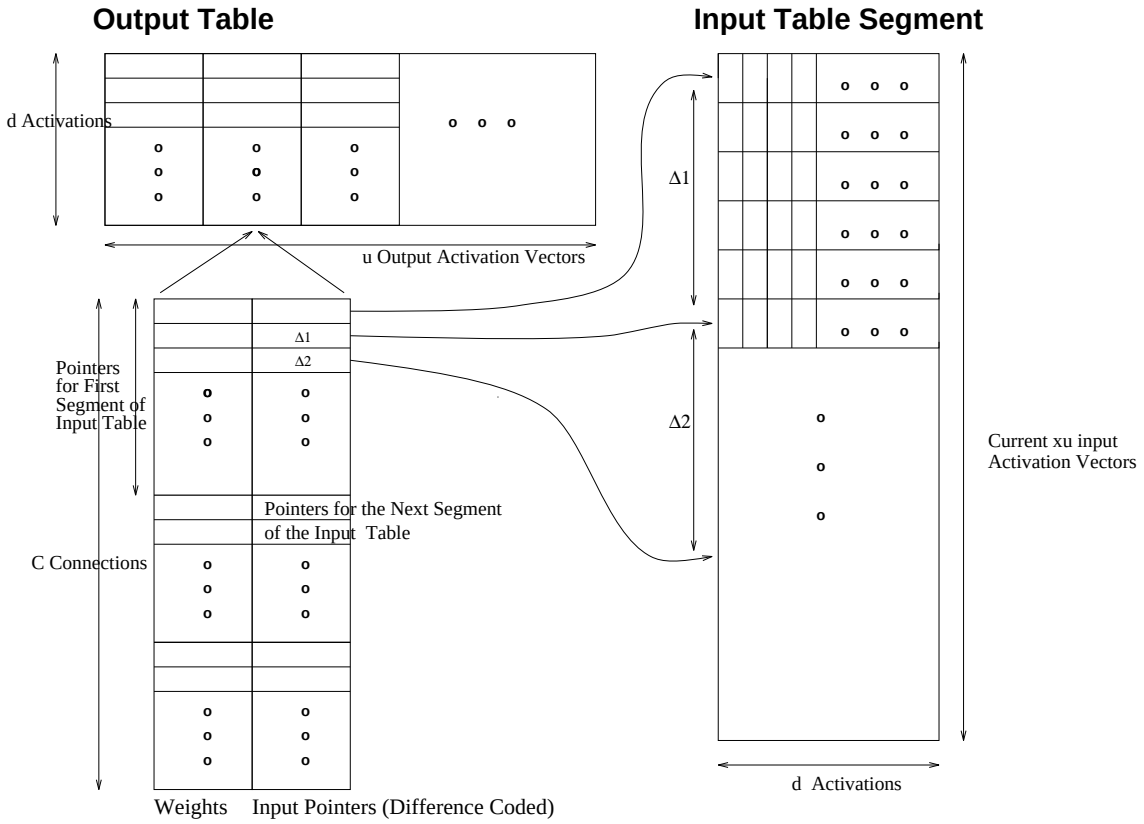

Figure 4.2: Representation for Pipelined Sparse Nets

the input table. The unit's connection segments are ordered according to the sequence in which the the input table segments arrive at the processor. Within a connection segment, we employ a *difference code* so that the pointers can be represented with 2 bytes. Instead of storing the ith pointer t_i directly, we store the difference, $t_i - t_{i-1}$. With a 2 byte difference code, successive input pointers may be separated by upto 64KB.

In case the jump between two successive pointers is more than 64K, a dummy pointer may be used as an intermediate "step" with a zero weight. We now analyze how many dummy weights are necessary. The probability that the next connection occurs after a gap of exactly g inputs is

$$p_{next}(g) = p_c(1 - p_c)^g.$$

The probability that there is *no* connection to the next g inputs is

$$p_{nogap}(g) = 1 - \sum_{k=0}^{g} p_c(1 - p_c)^k = (1 - p_c)^{g+1}.$$

Since each input is a d-byte vector, 64KB only holds 2K input vectors. In the reference case, this probability works out to $p_{nogap}(2K) = 0.14$, i.e. 14% of the time, the gap between two successive inputs will be greater than 64K.

To simplify matters, using the reference case as a guide, we assume that the number of dummy pointers and weights is $0.5c$. **In effect, we have $C = 1.5c$ unit connections.**

A single iteration of the neural network algorithm is now divided into P/x phases, due to the segmentation of the input table. During one of these phases, a processor multiplies its current input table segment with the corresponding connection segment and updates its output vectors. Until the final phase, the output vectors only hold intermediate values. These data are 4 bytes wide instead of 2 to avoid a precision loss. After each phase, the processors receive x new input table blocks from their neighbors. The details of the per-processor computation and of the communication are described later on.

4.3 Memory Requirements

A unit still requires 4 bytes for each of its C connections, $4d$ bytes for its 4 byte intermediate results pipelined to a depth of d and d bytes for its final output. For a processor's u units this adds up to $(4C+5d)u$ bytes. Each segment of the input table occupies $(x/P)dU$ bytes. In order to overlap communication with computation, space is required for two segments of the input table. One segment is used for computation while the next segment is being received. Altogether, the connections, inputs and outputs require M_{pipe} bytes of memory space on each processor:

$$M_{pipe} = (4C + 5d)u + 2(x/P)dU = (6c + 5d + 2dx)u.$$

A larger value of x is preferable in order to improve caching and to reduce the reading and writing of intermediate results. For the reference case, we derive the maximum value of x that the memory can support, based on the formula for M_{pipe}. We assume that the total memory is 16MB including 1MB for the program code.

$$(6c + 5d + 2dx)u \leq 15MB$$

In the reference case, with $u = 4K$ units per processor, $d = 32$ and $c = 512$ connections, a processor can maximally hold

$$x_{max} \leq 8 \quad \text{, for } d = 32$$

input table blocks. Conversely, with $x = 1$, the maximum pipelining depth is

$$d_{max} \leq 95 \quad \text{, for } x = 1.$$

We choose a value of $x = 8$ for our analysis. Even with more memory and consequently a larger value of x_{max}, we would not obtain a significant performance improvement.

4.4 Communication

Every phase, each processor sends the last block of its current input table segment to the succeeding node on the ring and then forwards the next $x - 1$ input table blocks, just received from its neighbor. Altogether, each node sends and receives xdu data bytes per

phase. Since $du > 128$, this communication has the run time $T^{trans}(xdu)$ as shown in paper [Mül93a] and section 2.3. A whole iteration requires P/x of these communications. That takes

$$T^{comm}_{basic} = \frac{P}{x} T^{trans}(xdu) \text{ cycles.}$$

The CNS-1 can overlap additional computation with the transfer, if more time is spent in the network than in the communication handlers.

A communication phase overlapped with t cycles of computation can then be completed in

$$\begin{aligned} &\max(kT_{net}(128), t + (k-1)T_{CPU}(128)) + T_{CPU}(128) \\ &\qquad \approx \max(kT_{net}(128), t + kT_{CPU}(128)) \end{aligned}$$

cycles with $k = \lceil xdu/128 \rceil$. This formula is optimistic, because switching between computation and communication also costs time, especially on systems with data caches. Unless mentioned otherwise, we do not assume any overlap of communication phases with ongoing computation.

4.5 Per-processor Computation

At the beginning of an iteration, each processor first writes its previously calculated output activations into its input table segment, and collects further results from the $x-1$ previous nodes on the ring. These data fill the remaining $x-1$ input table blocks. The output activation table is zeroed out. Each computation phase executing the following steps for each unit U_i on the processor:

1. Set the input pointer register to the base address of the input table segment (register load, 1 cycle).
2. Load the current output activation vector for U_i (vector load of d 4 byte values, $\lceil d/4 \rceil$ cycles[1]).
3. For all the weights and pointers in U_i that correspond to the currently available inputs:
   (a) Load the next input pointer difference (scalar load, 1 cycle).
   (b) Add the input pointer difference to the input pointer register (scalar add, 1 cycle).
   (c) Load the next weight (scalar load, 1 cycle).
   (d) Load the input activation vector (vector load,$\lceil d/8 \rceil$ cycles).
   (e) Multiply the input activations by the weight to yield a scaled input activation vector (vector * scalar, $\lceil d/8 \rceil$ cycles).
   (f) Add the scaled input activations to the output activation vector (vector + vector, $\lceil d/8 \rceil$ cycles).

[1] The memory interface actually supports twice this bandwidth, but the SRAM speed is the limiting factor

4. Store the output activation vector back in the output activation table (vector store of 4 byte values, $\lceil d/4 \rceil$ cycles).

After each phase, each processor passes on x input table blocks to the next processor on the ring, while it receives the next chunk of input activations from the previous processor on the ring. Only in the last phase, this transfer is dropped.

We assume that memory and arithmetic operations are overlapped appropriately through the use of techniques like loop unrolling. Each iteration of the inner loop then requires the maximum of the memory access time ($2 + \lceil d/8 \rceil$ cycles), the computation time ($\lceil d/8 \rceil$ using both arithmetic pipes) and the number of instructions (6 instructions):

$$T_{pipe}^{loop} = max(\lceil d/8 \rceil + 2, \lceil d/8 \rceil, 6) \text{ cycles.}$$

The memory access time for one inner loop iteration is 6 cycles, which equals the number of instruction cycles needed. Since the limiting factor, is the instruction bandwidth, pipelining to a shorter depth than 32 will have no impact on the time.

On average, each unit has C connections, due to padding. Accessing these connections takes $6C$ cycles per unit. Loading and storing the output activation vector (steps (2) and (4)) requires additional $2\lceil d/4 \rceil u = 16u$ cycles for each of the P/x phases. The total computation time for U units on P processors, pipelined to a depth of 32 or less is therefore

$$T_{pipe}^{comp} = u\left(6C + 16\frac{P}{x}\right) = u\left(9c + 16\frac{P}{x}\right)$$

cycles. In the reference case, that works out to 19.9 million cycles. 5% of this time is spent reading and writing intermediate results, but a smaller value of x would make it worse. For $x = 1$ that percentage would be 30%.

4.6 Overall Performance with SRAM

The SRAM version of the CNS can transfer $b = 2$ bytes per cycle, as shown in section 3.6. Each communication phase therefore spends more time in the network ($T_{net}(128) = 70$ cycles) than in the CPU ($T_{CPU}(128) = 37$ cycles). During a whole iteration, the transfer time adds up to

$$T_{pipe}^{comm} = \frac{P}{x}\left(\left\lceil \frac{xdu}{128} \right\rceil T_{net}(128) + T_{CPU}(128)\right) = \frac{P}{x}\left(\left\lceil \frac{xu}{4} \right\rceil 70 + 37\right)$$

cycles. In the reference case with $x = 8$, T_{pipe}^{comm} equals 9.2 million cycles.

The total time for one iteration of the pipelined sparse net is the sum of the computation and communication times:

$$\begin{aligned} T_{pipe}^{SRAM} &= T_{pipe}^{comp} + T_{pipe}^{comm}(du) \\ &= u(9c + 16P/x) + 17.5\, uP + 37\frac{P}{x} \text{ cycles.} \end{aligned}$$

In the reference case, one iteration with 8Giga connections therefore takes about 29.1 million cycles, that is about 295 connections per cycle. Overlapping communication and computation reduces the execution time, allowing CNS-1 to achieve 347 connections per cycle.

4.7 Performance with SDRAM

As in the non-pipelined case, page faults are the only additional delays we have to consider. We also have to analyze whether a data cache improves the run time or not.

4.7.1 Without Data Cache

Without data cache, we use exactly the same algorithm as for SRAM. In the course of the inner loop, a page fault occurs every time we switch between the unit's memory and the input table, i.e. two page faults per connection. This adds up to $2t_p uC$ cycles per processor. Reading and writing the output activations causes $2u$ additional faults per phase. Over one iteration with P/x phases this adds up to:

$$2t_p u \left(C + \frac{P}{x} \right) = 4u \left(1.5c + \frac{P}{x} \right)$$

cycles or 12.8 million cycles in the reference case.

4.7.2 With Data Cache

Some of the page breaks can be saved if the unit input pointers and weights are in the data cache, but pre-loading the data cache cost extra time. Loading a single weight and input pointer takes only 2 cycles, while a page break takes 4 cycles.

Weights and pointers are pre-loaded into the data cache, in blocks of 4KB (cache size). During the computation, these data are then accessed in the cache without changing the memory page.

Over the course of an iteration, each processor pre-loads all its Cu connections with four bytes each. At a rate of 16 bytes per cycle, that takes $4/16 \cdot Cu$ cycles. The connections are processed in 4KB blocks. There occur $2Cu/4K$ page breaks when moving between the unit's memory and the input table. Altogether, a processor spends T_{pipe}^{cache} cycles loading the connections into the data cache:

$$T_{pipe}^{cache} = \frac{4Cu}{16} + p_f \frac{2Cu}{4K} = Cu \left(\frac{1}{4} + \frac{1}{1024} \right).$$

Loading inputs from the input table now causes most of the page breaks. The average distance between the inputs pointed to by adjacent input pointers in a unit is duP/C i.e. the total size of the input table divided by the number of connections. This assumes that the input pointers are uniformly distributed over the input table. In the reference case, the gap between successive inputs that must be loaded is 32K. Hence, it is reasonable to assume that *every* load of an input vector causes a page break. That adds up to a total of $t_p Cu$ cycles per processor.

The processors read and write the intermediate output activations during each phase. That causes additional 2 page faults per d-element vector. With data cache, a processor spends altogether

$$T_{pipe}^{page} = T_{pipe}^{cache} + t_p Cu + 2t_p \frac{P}{x} \frac{ud}{d} = Cu \left(\frac{1}{4} + \frac{1}{1024} + 2 \right) + 2u \frac{P}{x}$$

cycles with page faults. In the reference case, this equals 7.2 million cycles, with the lion's share for loading of the inputs. In contradiction to the basic model, using a data cache improves the run time of the pipelined model.

On the SDRAM system, the total computation time taken for one pipelined iteration therefore amounts to

$$T_{pipe}^{comp} = u(9c + 16P/x) + T_{pipe}^{page} \approx u(12.38c + 20P/x) \text{ cycles.}$$

4.7.3 Transfer Time

On a CNS with SDRAM, the network takes $T_{net}(128) = 70$ cycles to transfer a 128 byte message, and the CPU takes $T_{CPU}(128) = 39 + 1/32$ cycles to deal with it (section 3.7.3). So, the network time is still longer than the CPU time. During a whole iteration, the transfer time adds up to

$$T_{pipe}^{comm} = \frac{P}{x}\left(\left\lceil\frac{xu}{4}\right\rceil 70 + 39.03\right) \approx 17.5uP + 39.03\frac{P}{x}$$

cycles. In the reference case with $x = 8$ that equals 9.2 million cycles.

4.7.4 Overall Performance

The total time taken for one iteration of the pipelined case with SDRAM is the sum of the computation and communication times and can be expressed by the following formula:

$$\begin{aligned} T_{pipe}^{SDRAM} &= T_{pipe}^{comp} + T_{pipe}^{comm} \\ &= u(12.38c + 20P/x) + 17.5\,up + 39.03\frac{p}{x} \text{ cycles.} \end{aligned}$$

In the reference case, this works out to 36 million cycles for 8G connections, or about 236 connections per cycle.

4.8 Performance with RDRAM

If CNS-1 uses RDRAM, an efficient use of the data caches is essential for high performance. This makes the run time analysis more difficult.

4.8.1 Per-processor Computation

Half the on-chip data cache (2KB) is used for weights and pointers, with the other 2KB used for intermediate results. Aside from that, we basically use the same algorithm as before. Computation and memory accesses are completely overlapped, and so the run time only depends on the memory access times. The processors pre-load weights and pointers in blocks of $\alpha = \lceil Cx/P \rceil \cdot 2^{\beta}$ elements:

$$\beta = \left\lfloor \log_2\left(\frac{2K}{4\lceil Cx/P\rceil}\right)\right\rfloor.$$

Both vectors are only two bytes wide, and so the memory system can overlap two consecutive accesses. Only half the loads are visible. The first four vector loads per block are RDRAM cache misses; they require at least 22 cycles each. The remaining loads are cache hits, with a 14 cycle access time. Later on, the processors access the weights and pointers in the cache, to perform the computation. This takes additional 2 cycles per connection, and so handling weights and pointers adds up to T^{weight} cycles per iteration:

$$T^{weight} = \frac{Cu}{\alpha}\left(\frac{\alpha}{32}14 + 4(22-14) + 2\alpha\right) \approx 3.66\,cu + 32\frac{uP}{2^\beta x}.$$

Half the data cache (2KB) is reserved for intermediate results. Since they are 4 byte wide, the cache can hold 16 vectors of length $d = 32$. The intermediate results are a word wide. The processors pre-load 16 result vectors of length $d = 32$ into the data cache, update them, and write the whole cache block back. 16 blocks fit in one RDRAM cache line, which is spread over four memory banks. When loading 16 blocks, only four loads are dirty RDRAM cache misses, the remaining loads are hits. A dirty load requires 28 cycles, a hit 14 cycles. When starting a new phase, the dirty misses actually occur during loading the weights and pointers. This does not change the run time, because the four clean misses then occur when loading the result vectors. Writing the vector back to the RDRAM is always a clean cache miss, and takes 16 cycles per write. Moving the result vectors between cache and CPU requires an additional $2 \cdot VLR/8 = 8$ cycles per vector. That adds up to T^{result} cycles per iteration:

$$T^{result} = \frac{P}{x}\frac{ud}{16d}\left(16(14+16+8) + \frac{4}{16}(28-14)\right) \approx 38.22\,u\frac{P}{x}.$$

The input vectors are directly accessed in the RDRAM. All accesses are clean misses, due to indexed accesses. As shown in section 4.7.2, the gap between input vectors accessed successively is 32KB in the reference case. That is also the length of a RDRAM cache line, and so every load will be a cache miss.

Writing back the result vectors actually causes $uP/(16x)$ dirty RDRAM cache lines; 15/16 of them influence the load time of the input vectors. Since byte vectors only update one memory bank, 16 loads are necessary to update a whole RDRAM cache line. That extends the execution time by $15/16 \cdot uP/(16x) \cdot 16(28-22)$ cycles. During one iteration, each processor therefore spends T^{input} cycles loading inputs:

$$T^{input} = \frac{Cud}{32}22 + \frac{15uP}{256x}16(28-22) = 33uc + \frac{45}{8}u\frac{P}{x}.$$

The total computation time adds up to

$$T^{comp}_{pipe} = T^{weight} + T^{result} + T^{input} \approx 36.66\,uc + \left(43.8 + \frac{32}{2^\beta}\right)u\frac{P}{x}$$

cycles per iteration. In the reference case, it will take 80 million cycles to perform the computation of one iteration.

4.8.2 Transfer Time

On a CNS with RDRAM, a 128-byte transfer spends more time in the network than in the CPU (section 3.8.2). It keeps the CPU busy for $T_{CPU}(128) = 65$ cycles. Depending on the

cycle time, the message spends $T_{net} = 70$ or 138 cycles in the network. Per phase, each processor sends xud data bytes to its neighbor. During a whole iteration, the transfer time therefore adds up to

$$T_{pipe}^{comm} = \frac{P}{x}\left(\left\lceil\frac{xdu}{128}\right\rceil T_{net}(128) + T_{CPU}(128)\right) = \frac{P}{x}\left(\left\lceil\frac{xu}{4}\right\rceil 138 + 65\right)$$

cycles. In the reference case with $x = 8$ that equals 18 million cycles.

4.8.3 Overall Performance

The total run time is the sum of the computation and communication times.

$$\begin{aligned} T_{pipe}^{RDRAM} &= T_{pipe}^{comp} + T_{pipe}^{comm} \\ &= 36.66\,uc + \left(43.8 + \frac{32}{2^{\beta}}\right) u\frac{P}{x} + \frac{P}{x}\left(\left\lceil\frac{xu}{4}\right\rceil 138 + 59\right) \end{aligned}$$

In the reference case, one iteration with 8G connections therefore takes about 98 million cycles, that is about 88 connections per cycle.

4.9 Performance of CNS-1 on the Pipelined Sparse Case

Table 4.1 summarizes the run time and performance of the CNS-1 under various memory systems. The values are taken on the pipelined sparse model with reference problem size. As in the basic case, we compare the absolute performance number to the ALU peak performance and System peak performance of a CNS with the same clock rate.

4.9.1 Peak Performance

Since there are no indexed vector load/stores in the pipelined sparse network algorithm, the System peak performance and the Indexed peak performance are the same.

ALU Peak Performance

In this case, we only count the arithmetic operations. Each processor has two arithmetic vector pipeline and one scalar pipeline. A arithmetic vector operation takes 4 cycles for 32 vector elements. A processor computes dCu connections per iteration. The calculation requires a scalar addition, a vector multiplication and a vector addition per d connections. Consequently, the arithmetic operations require

$$T_{pipe}^{ALU} = \left\lceil\frac{d}{8}\right\rceil Cu$$

cycles per iteration. In the reference case, that adds up to 12.6 million cycles, and with a cycle time of $t_c = 20$ns (8ns) it equals 34.1 (85.3) GCPS.

Memory System	SRAM	SDRAM	RDRAM		
Cycle Time	20	20	20	8	ns
T_{pipe}^{comp}	19922944	27134976	80009216	80009216	
T_{pipe}^{comm}	9175632	9175665	9176080	18088976	cycles
T_{pipe}	29098576	36310641	89185296	98098192	
Absolute Performance	14.8	11.8	4.8	11.0	GCPS
Relative to ALU Peak	43.2	34.7	14.1	12.8	%
Relative to System Peak	98.2	78.7	32.0	38.2	%

Table 4.1: Run time and performance overview of the reference case ($u = 4096$, $c = 512$, $p = 128$, $x = 8$)

System Peak Performance

For this peak performance, we assume an ideal memory. The only difference to the SRAM system is the access time for word vectors. The SRAM executes these accesses only at half the optimal speed. The ideal memory system only requires 4 cycles per 32 word vector.

That reduces the computation time T_{pipe}^{comp} (page 26) by $8uP/x$ cycles and the transfer time by $8P/x$ cycles. We assume, that the SRAM system runs at 20ns. A faster cycle time only changes the time spent in the network, because the link bandwidth would not be scaled up. For a 128 byte transfer T_{net} would rise from 70 to 138 cycles. That results in the following run time formula for the system peak performance:

$$T_{basic}^{System} = T_{basic}^{SRAM} - 512K + \begin{cases} -8P/x & , \quad t_c = 20ns \\ (138-70)xu/4 - 8P/x & , \quad t_c = 8ns \end{cases}$$

In the reference case, that equals between 28.6 and 37.5 million cycles or between 15 and 28.6 GCPS.

4.9.2 Results

The SRAM system still has the best absolute performance. To achieve the same run time, the SDRAM system would have to run 1.36 faster, at more than 68MHz. The RDRAM version would have to run four times faster than the SRAM version, at about 200MHz. It is therefore most realistic, to achieve a performance of 14 GCPS with the simpler SRAM system.

Compared with the performance results on the basic sparse case, all three systems do much better with pipelining. The vector unit can now be used moe efficiently. The

processors only load one weight and input pointer per vector and not per vector element as in the basic case. That reduces considerably the amount of loads and indirect accesses and therefore increases the performance by a factor of 3.4 to 4.6. The biggest advantage is on the RDRAM system, because the pipelined algorithm uses more block accesses.

Chapter 5

Conclusion

We developed efficient parallel vector code of two sparse neural network algorithms and analyzed their performance on the CNS-1. The analysis was performed for the three memory systems that are currently being considered for the design, SRAM, SDRAM and RDRAM. Even after taking into account the higher clock speed of the DRAM systems, they are outperformed by SRAM. Between the DRAM systems, SDRAM performs far better than RDRAM. DRAM systems are designed for fast block accesses, but random accesses are their weak point. A low random access time is important especially for the sparse basic model. On all three memory systems, pipelining improved performance considerably - between 3.4 to 4.6 times - because it reduces the amount of memory accesses and indexed loads per connection. However, the naive use of pipelining requires more memory than is available, but distributing the input table over all the processors instead of replicating it solves this problem.

5.1 Results

In the course of designing the data-structures and doing the performance analysis, some important design issues became clear:

- Contiguous vector accesses are crucial. Since the CNS and its memory system only have minimal support for scatter and gather operations, indexed vector load/stores cause a big performance loss.

- Careful representation of sparse connections is essential for good performance and in order to meet the storage requirements. Without careful attention, pointers easily take double the space required for weights and activations.

- Nearest neighbor transfers are sufficient for this application. We implemented all the communication, including the all-to-all broadcast, by nearest neighbor transfers. Since nearest neighbor transfers saturate the bandwidth of the link between a processor and the corresponding network node, more complex communication schemes would not reduce the transfer time (which is 10 to 30% of the total run time).

- RDRAM was very hard to design for and to analyze, due to its multi-level caching[1]. Even if RDRAM does yield good performance in certain cases, obtaining that performance takes a considerable effort. With limited software development resources, an RDRAM system would probably end up with far from optimal performance on all but a small suite of very carefully coded programs.

- The main issue with an SDRAM system is what should be placed in the data cache. There is a trade-off between how long it takes to load the data cache and how many page faults are saved through its use.

From our analysis, it seems quite clear that an SRAM memory system is the memory system of choice. However, if using SRAM would imply a much smaller node memory, then SDRAM might be preferable. SDRAM is still quite easy to design for and yields good performance, at least on our idealized network model.

Backpropagation on large dense layered networks is the second benchmark application of the CNS-1. The results of the report [Mül93b] showed, that a CNS-1 with RDRAM memory system could simulate a multi-layer backprop network at close to (ALU) peak performance. Using a SRAM or SDRAM system which runs at a slower system clock consequently results in a big performance loss.

Our algorithms for sparse neural networks used only a ring topology of the processors, but that should not be taken to mean that the barrel topology is unnecessarily complex. If the CNS were to adopt a ring topology, the performance of backpropagation would be severely impacted. In addition, it will frequently be necessary to run different sub-networks on different parts of CNS-1, and a single ring would be quite inappropriate for this purpose.

5.2 Future Work

Actual network algorithms differ from our idealization in one way or the other, but we tried to capture the most time consuming part of those algorithms.

Omitting the threshold function should have no big performance impact, because the processors simulate cu units but compute the threshold function for only u values.

We chose a random connectivity structure, that fits well with some applications, but on structured networks we might find clusters of higher connectivity. Many of our arguments (for instance, the fact that an all-to-all broadcast is more efficient than selective communication - section 3.5) may need to be rethought. Perhaps a model similar to that used by ([GH88]) may be more appropriate.

One important case that we are currently considering is the sparse activation case i.e. the case in which most of the activations are zero. We will show that a considerably different representation scheme yields much better performance in this case. Another case that is worth considering is the backprop case. The data structures we have designed are ideal for forward propagation, but they also work well for the backward propagation step, as a brief analysis showed. The second author will also be working on a simulator that that aims at

[1] In fact, at one point, additional levels of caching were being proposed to further improve the performance of RDRAM.

helping connectionist researches achieve good performance on CNS-1 without the intimate knowledge of the hardware details that was necessary for this analysis.

Bibliography

[ABC+93] K. Asanović, J. Beck, T. Callahan, J. Feldman, B. Irissou, B. Kingsbury, P. Kohn, J. Lazzaro, N. Morgan, D. Stoutamire, and J. Wawrzynek. CNS-1 architecture specification. Technical Report TR-93-021, International Computer Science Institute and UC Berkeley, 1993.

[AC93] K. Asanović and T. Callahan. *Torrent Architecture Manual.* International Computer Science Institute and UC Berkeley, 1993. Internal document, revisions 1.5/1.9.

[Asa93] K. Asanović. *T0 Reference Manual.* International Computer Science Institute and UC Berkeley, 1993. Internal document, revision 1.5.

[Bre93] Williams Brett. Synchronous DRAMs: Designing to theJEDEC standard. *MICRON: Design Line*, 2(2), 1993.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eichen, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[GH88] Joydeep Ghosh and Kai Hwang. Mapping neural networks onto message passing multicomputers. *Journal of Parallel and Distributed Processing*, 6:291–330, 1988.

[LCDS90] Y. Le Cun, J.S. Denker, and S.A. Solla. Optimal brain damage. In D.S. Touretzky, editor, *Proceedings of NIPS 1989*, volume 2, pages 598–605. Morgan Kaufmann, 1990.

[Mül93a] S. M. Müller. All-to-all Broadcast on the CNS-1. Technical Report TR-93-082, International Computer Science Institute, Berkeley, 1993.

[Mül93b] S. M. Müller. A performance analysis of the CNS-1 on large, dense backpropagation networks. Technical Report TR-93-046, International Computer Science Institute, Berkeley, 1993.

[Ram92] Rambus Inc., Mountain View, California. *Rambus Technology Guide*, 0.90 - preliminary edition, May 1992.

[SAM93] SAMSUNG Electronics. *SAMSUNG Synchronous DRAM*, revision 1 edition, March 1993.

[Sha88] Lokendra Shastri. *Semantic Nets: Evidential Formalization and its Connectionist Implementation.* Morgan Kaufmann, 1988.

[Tos92] Toshiba. *Advanced Information:* $2M \times 9$ *RDRAM*, 1992.

[vCGS92] T. von Eichen, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th Int. Symposium on Computer Architecture.* ACM Press, 1992.