INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

A Performance Analysis of the CNS-1* on Large, Dense Backpropagation Networks

Connectionist Network Supercomputer

Silvia M. Müller
smueller@icsi.berkeley.edu
TR-93-046
September 1993

Abstract

We determine in this study the sustained performance of the CNS-1 during training and evaluation of large multilayered feedforward neural networks. Using a sophisticated coding, the 128-node machine would achieve up to 111 Giga connections per second (GCPS) and 22 Giga connection updates per second (GCUPS). During recall the machine would archieve 87% of the peak multiply-accumulate performance. The training of large nets is less efficient than the recall but only by a factor of 1.5 to 2.

The benchmark is parallelized and the machine code is optimized before analyzing the performance. Starting from an optimal parallel algorithm, CNS specific optimizations still reduce the run time by a factor of 4 for recall and by a factor of 3 for training. Our analysis also yields some strategies for code optimization.

The CNS-1 is still in design, and therefore we have to model the run time behavior of the memory system and the interconnection network. This gives us the option of changing some parameters of the CNS-1 system in order to analyze their performance impact.

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

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure of the analysis . . . 1

2 Modelling the Timing **3**
2.1 Memory system . . . 3
2.1.1 Memory hierarchy . . . 3
2.1.2 On-chip caches . . . 4
2.1.3 Main memory . . . 4
2.1.4 Refresh . . . 4
2.2 Data network . . . 5
2.2.1 Single transfer . . . 5
2.2.2 Multicast . . . 6
2.2.3 Multiple transfers . . . 6
2.3 Datapath . . . 7
2.3.1 Location of scalars . . . 7
2.3.2 Multiply-accumulate instructions . . . 7

3 Benchmark Problem **8**
3.1 Description of the applications . . . 8
3.1.1 Recall . . . 9
3.1.2 Training . . . 9
3.2 Parallelization and data distribution . . . 10
3.2.1 Current approach . . . 11
3.2.2 Alternative approach . . . 11

4 Performance Analysis of Kernel 1 **13**
4.1 Step 1: Reading the activations . . . 13
4.1.1 I/O on the 128-node machine . . . 14
4.1.2 I/O on the 1024-node machine . . . 16
4.2 Steps 2 and 3: Computation of the next layer . . . 17
4.2.1 Matrix vector multiplication . . . 17
4.2.2 Global sum . . . 19
4.2.3 Sigmoid computation . . . 19
4.2.4 Interleaving the three main routines . . . 20

4.2.5 Penalty of I-cache misses 21
4.3 Step 4: Writing the results 22
4.3.1 Timing in the Torrent network 23
4.3.2 Timing in the Hydrant network 23
4.4 Problems with the D-cache size 23
4.4.1 Steps 1 and 4 24
4.4.2 Steps 2 and 3 24

5 Performance Results of Kernel 1 **28**
5.1 Performance results 28
5.1.1 Detailed results under fixed conditions 28
5.1.2 General results 29
5.1.3 Implications for the CNS design 30
5.2 Influence of the mapping and the parallelization scheme 31
5.2.1 Local optimizations 32
5.2.2 Global optimizations 33

6 Performance Analysis of Kernel 2 **34**
6.1 The forward pass 34
6.1.1 I/O during training 35
6.1.2 Computation of the next layers 35
6.2 The backward pass 37
6.2.1 Step 5: Error computation 37
6.2.2 Error backpropagation 40
6.2.3 Weight updates 44
6.3 Cache problems 47
6.3.1 Step 1 47
6.3.2 Steps 2 and 3 47
6.3.3 Step 5 48
6.3.4 Steps 6 and 7 48
6.3.5 Steps 8 and 9 49

7 Performance Results of Kernel 2 **51**
7.1 Performance results 51
7.1.1 General results 51
7.1.2 Detailed results under model conditions 52
7.2 Influence of the CNS specific optimizations 55
7.2.1 Description of the optimizations 55
7.2.2 Performance impact of the optimizations 56
7.3 Comparision of recall and training 57
7.3.1 Scaling of the performance 57
7.3.2 Limiting memory bandwidth 58
7.3.3 Efficiency of the assembler code 58

8 Conclusion **59**
8.1 General results 59
8.2 Implication for the CNS design 59
8.3 Optimization strategies 60

A Assembler Code **61**
A.1 Kernel 1: Recall 61
A.2 Kernel 2: Training 65
A.2.1 Forward pass 65
A.2.2 Backward pass 68

Chapter 1

Introduction

1.1 Motivation

In the architecture specification of the CNS-1, Asanović et al sketch a benchmark application for this multicomputer and demand a performance of 10^{11} connections per second:

> "Extensive work in the area of connectionist speech understanding points to the need for a machine two to three orders of magnitude faster than the best machines available today. In condensed form, a statement of the requirements for the one abstract problem, representative of potential applications is: *Evaluate the activations in a network with one million units having an average of a thousand connections per unit for a total of a billion connections. This should be done one hundred times per second.*" ([ABC+93], p. 5)

The question arises whether the CNS-1 can really execute such a benchmark with the required performance or not. As one result of the report, we will show that this is possible.

Since the CNS-1 is still in design, performance results have now the biggest impact for the following two reasons: First, detailed performance analysis can localize design bottlenecks, which at this stage it is fairly cheap to work out. Second, there are still some open aspects in the design. Our analysis should help the design team solving these problems.

This report is also motivated by the fact that programming the CNS-1 efficiently is not a trivial task. That is because the CNS-1 is a special purpose multicomputer with three levels of parallelism: the Torrent nodes work in a MIMD manner, each Torrent can execute up to four instructions at a time, and three of these instructions are vector operations. At the moment, there exists no efficient compiler for the machine, and therefore most of the code has to be hand optimized.

1.2 Structure of the analysis

The design team describes the design and the functionality of the CNS-1 very detailed in the technical reports [ABC+93, Asa93, AC93], but until now they have only sketched out the timing behavior of the memory system and the interconnection network. Since for our analysis we also need detailed information on these two components, we model them in

chapter 2. We try to keep the model as consitent as possible, but the design will definitely change over the next time, partially caused by the result of this report.

Chapter 3 deals with the benchmark, its specification and data distribution. Originally the recall of large nets was considered to be the only part of the benchmark application. We use the training as a second benchmark kernel. In the following four chapters we parallelize the two kernels, optimize their machine code, determine the run times and analyze the resulting performance. The last chapter summarizes the results and their possible impact on the machine design. We also sketch some strategies for CNS specific code optimization.

General Notation

In this document, **B** designates *bytes*, and **b** designates *bits*.

Acknowledgments

This work has benefitted a lot from discussions with various members of the realization and CNS groups at ICSI. I would particularly like to thank Krste Asanović, Jim Beck, Tim Callahan, Jerry Feldman, David Johnson, Brian Kingsbury, Phil Kohn, Nelson Morgan and John Wawrzynek.

Chapter 2

Modelling the Timing

The **C**onnectionist **N**etwork **S**upercomputer (CNS-1) has been designed for neurocomputation and should reach high performance even on sparsely connected neural networks; this at least is one goal of the project [ABC+93]. The CNS is a multicomputer with distributed memory, based on a super-scalar design. The nodes are connected via a mesh with wraparound in one dimension. Each processor has a scalar unit and three coprocessors: a network interface, a memory unit, and a small SIMD array. The scalar unit is a MIPS-based design extended by a few instructions to communicate with its coprocessors. The SIMD unit is accessed via vector operations. The details of the hardware and the ISA are described in [ABC+93, Asa93, AC93].

2.1 Memory system

2.1.1 Memory hierarchy

The CNS-1 has a three level memory hierarchy: separate instruction and data cache on the processor chip, a second level cache in each RDRAM chip, and the RDRAM itself. All the caches are direct mapped, and their capacity is given in Table 2.1.

	Cache Size	Line Size	No. of Lines
I-Cache	4KB	(32×4B) 128B	32
D-Cache	4KB	32B	128
RDRAM-Cache			
4.5 Mb chip	2KB	1KB	2
18 Mb chip	4KB	2KB	2

Table 2.1: Cache capacity

RDRAM	Read			Write		
	Hit	Miss		Hit	Miss	
		/dirty	dirty		/dirty	dirty
4.5 Mb	$48+2x$	$152+2x$	$152+2x$	$16+2x$	$120+2x$	$120+2x$
18 Mb	$44+2x$	$112+2x$	$156+2x$	$12+2x$	$64+2x$	$108+2x$

Table 2.2: Main memory access time for x Byte [ns]

2.1.2 On-chip caches

The I-cache delivers one instruction per CPU cycle; a cache miss takes between 20 and 45 cycles. This time includes the memory access and the update of the I-cache. In this analysis we always assume the worst case for the delay of an I-cache miss.

The D-cache supports byte, half word and word mode. Both, scalar and vector unit, can access the D-cache, but only one at a time. A scalar access takes one cycle; a vector access needs $\lceil \text{vlength}/32 \rceil$ cycles to be completed. In the case of a cache miss, the delay of the main memory has to be added to the access times.

The D-cache is not large enough to store all the data. Less frequently used data should only be stored in the main memory. We assume that there are special load and store instructions to access these data, so that they can bypass the cache, and that the CPU still can access the D-cache during the RDRAM ports are executing some requests.

2.1.3 Main memory

The memory consists of four banks, each connected to its own memory interface. Each interface controls four RDRAM chips. The ports are interleaved every 32B and the banks every 128B. Therefore, neighboring 32B 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.

The memory access time mainly depends on three parameters: the capacity of the RDRAM chip, the type of the access (read/write), and whether the data are in the RDRAM cache or not. In the case of a cache miss, the access time also depends on the dirty flag; the write back of a dirty cache line takes some additional cycles (Table 2.2). The data sheets [Ram92, Tos92b, Tos92a] specify the memory access times in nanoseconds, but all our analysis is done in cycles. We assume that one CPU cycle of the CNS takes 8 ns.

2.1.4 Refresh

The refresh of the RDRAMs has to be started at least every 1024 μs (128000 cycles) and takes about 1 μs (125 cycles). In the best case all refreshes occur when the main memory is idle and so they do not affect the run time of the benchmark. However, it is much more likely that some refreshes will contend with memory accesses and therefore delay the execution of the kernels. In the worst case the run time increases by a factor of $\tau_{ref} = 1024/(1024-1)$, that is less than 0.098 %.

2.2 Data network

There are only three types of transfers in our benchmark: single transfer, multicast and multiple transfers.

The processor overhead for sending and receiving messages is fairly small, because of active messages [CSS+91]. In our analysis, we assume that they are $T_{p,s} = T_{p,r} = 10cc$ (CPU cycles). This includes the overhead of moving the data to the transfer registers. We also assume that the overhead in the network interface for sending and receiving is of the same magnitude: $T_{n,s} = T_{n,r} = 10cc$. The message length m_{length} is counted in bytes; it is the amount of data plus nine header bytes. ([Cal93, AC93] gives a detailed description of the network interface.)

2.2.1 Single transfer

Under a single transfer we understand, that a processor p sends a message to a processor q, and that there is no further activity going on which could block the transfer. The message has to performe h_{op} network hops. Figure 2.1 illustrates the timing of such a transfer; the message is 20 bytes long and passes two intermediate nodes, $p+1$ and $p+2$. In general, a single transfer works as follows:

- Node p switches to the handler and prepares the message. This takes $T_{p,s}$ units of time. After that p is free for other work and the network deals with the message.
- After time $T_{p,s}$ the network interface of node p deals with the message. $T_{n,s} - 1$ cycles later the global part of the interface gives the message to the direction-dependant part of the hardware which is then busy for m_{length} cycles.
- A package leaves the buffer one cycle after it arrived, and spends one cycle in the link. Each hop in the network therefore delays the message by two cycles.
- The interface of node q receives the first byte after $T_{p,s} + T_{n,s} + 2h_{op}$ cycles and has to deal with this message for $m_{length} - 1 + T_{n,r}$ cycles.
- Now processor q gets an interrupt and switches to the handler. Reading the message out of its network interface takes additional $T_{p,r}$ cycles.

In the best case the whole transfer takes time

$$T_{ST}(h_{op}) = T_{p,s} + T_{n,s} + 2h_{op} - 1 + m_{length} + T_{n,r} + T_{p,r}.$$

Node p can send the next message to node q without any network conflicts $m_{length} + 1$ cycles after it started the previous transfer. The network buffers are the critical hardware elements. They deal $m_{length} + 1$ cycles with a transfer. After that time, they are able to receive a new package from the same direction.

Under real circumstances, it is possible that an interrupt can not be executed immediately or that a package is blocked for a while. This delay increases the transfer time T_{ST}.

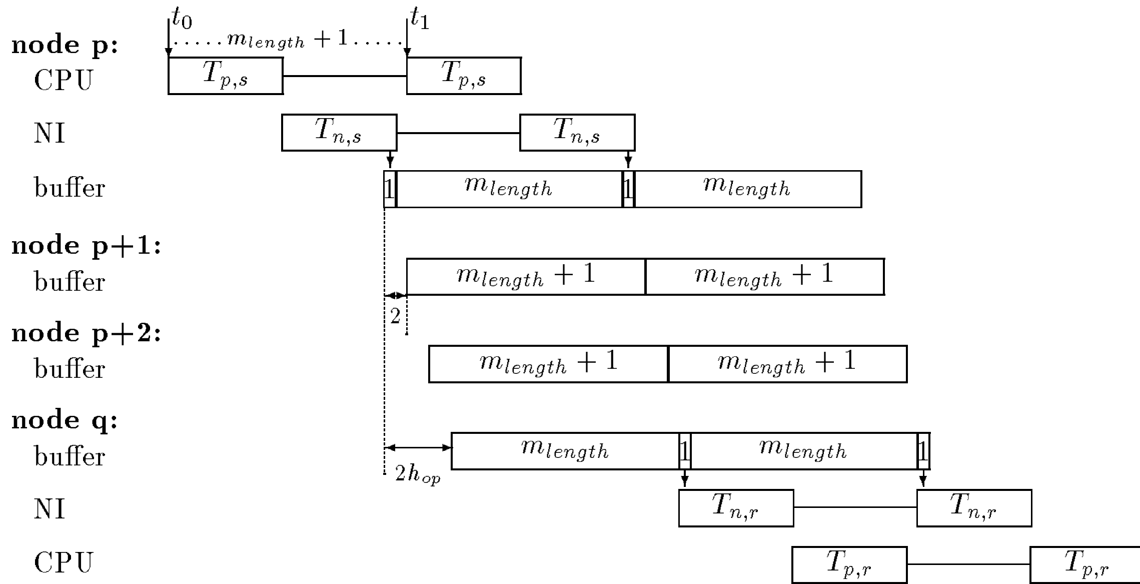


Figure 2.1: Timing diagram for sending a 20B message from p to q via nodes p+1 and p+2.

2.2.2 Multicast

The CNS-1 has a simple but quite effective mechanism that allows a multicast to be executed as fast as a single one-to-one transfer. A special flag in the message header gives the option to a sender to drop off a copy of the message at every node along the path from the source to the destination.

When receiving a message with an active multicast flag, the router forwards it to a network link and to the CPU. The router is a crossbar, and therefore routing the message to one or two buffers requires the same amount of time.

This is not a mechanism for general multicast or broadcast, but it is sufficient for distributing data along columns or rows of the processor network. It is the fastest mechanism for that type of multicast.

2.2.3 Multiple transfers

This type of transfer involves $2k$ adjacent nodes ($1 \leq k \leq 16$). The first k nodes $p_1, \cdots, p_k$ send messages to the remaining k nodes and vice versa.

transfer 1:	p_i sends to p_{i+k}	$1 \leq i \leq k$
transfer 2:	p_{i+k} sends to p_i	

Both transfers occur at the same time and will not interfere with one another because they use different parts of the network. For a ring of 32 nodes it is a bit different. The direction of the message alternates from node to node, but this makes the transfer only faster.

The transfer from node p_k to p_{2k} takes $T_{ST}(k) = 40 + 2k - 1 + m_{length}$ cycles. The other $k - 1$ messages have to use the same part of the network, therefore they are blocked for a while. The message of node p_{k-i} is delayed for $i \cdot m_{length}$ cycles. Without further problems the whole transfers complete in $T_{MT}(k) = T_{ST}(k) + (k-1) \cdot m_{length} = 39 + 2k + k \cdot m_{length}$ cycles.

2.3 Datapath

We make two assumptions for the datapath which might not be consistent with the CNS-1 design. One is the location of the scalars for vector operations, and the other is the execution of a multiply-accumulate instruction. At the moment the implementation of these aspects is still open. We will therfore analyze the performance impact of different solutions but in general we assume the following:

2.3.1 Location of scalars

Some vector operations require scalars. These variables can only be changed by the scalar unit, but the vector unit can read them. We assume that they are stored in a special register file in the vector unit. The scalar unit reads and writes this register file via special move instructions.

It is also possible, to store them in the register file of the scalar unit. This solution requires less hardware, but makes the register file control more complicate.

2.3.2 Multiply-accumulate instructions

We assume that the arithmetic vector unit VP0 has a special multiply-accumulate instruction, and that this instruction takes as long as a multiply instruction. This is possible in the current design of the CNS-1 datapaph but it requires a change in the machine language and the control logic.

Chapter 3

Benchmark Problem

Detailed performance analyses require a specification of the workload. In the field of neurocomputing the workload is usually defined as the recall step of a connectionist network. On the other hand, the CNS-1 should be used for studying large connectionist networks and designing new neuroalgorithms. This implies that a huge amount of computing time will be spent training the networks. For these reasons, both training and recall should be represented in the benchmark.

3.1 Description of the applications

Our benchmark focuses on dense three-layer networks with about 10^9 connections. The first kernel deals with recall and the second with training. We use a backpropagation algorithm described in [MR88, Gre92]. The underlying neural network has one hidden layer which is completely connected to the other two layers. The network has n_i inputs, n_h hidden nodes, and n_o outputs, and its size varies between 201.3 million and 1.34 billion connections. The input patterns are mapped into several classes with 64 subclasses each. The size of the layers is listed in Table 3.1.

When presenting a pattern to the input layer, the information stored in the weight matrices and bias vectors are used to compute an output vector. We call this step *recall*.

Connections	Global Size				Local Size			Machine Size		
	n_i	n_h	n_o		n_{pi}	n_{ph}	n_{po}	p_i	p_h	# Nodes
192M	8K	16K	4K	(64×64)	1024	1024	512	8	16	128
768M	16K	32K	8K	(128×64)	2048	2048	1024	8	16	128
1536M	16K	64K	8K	(128×64)	2048	4096	1024	8	16	128
768M	16K	32K	8K	(128×64)	1024	512	512	16	64	1024
1536M	16K	64K	8K	(128×64)	1024	1024	512	16	64	1024

Table 3.1: Size of the neural network and dimension of the local connection matrices, when parallelized as defined in section 3.2.1

When presenting an input and a result pattern at the same time, both vectors are used to update the information stored in the neural network. This step is called *training*.

3.1.1 Recall

During recall, the machine reads an activation vector. This input vector $\mathbf{i}$ specifies the values of the input layer of the neural network. The values of the hidden vector $\mathbf{h}$ are then obtained combining the input vector with the weight matrix $\mathbf{A}$ and the bias vector $\mathbf{a}$. $\mathbf{A}$ and $\mathbf{a}$ are related to the first two network layers. In the following step the values of the output vector $\mathbf{o}$ are computed by combining the hidden vector with the weight matrix $\mathbf{B}$ and the bias vector $\mathbf{b}$, both related to the second and third layer. Mathematically the recall or *forward pass* for one pattern can be described as follows:

$$
\begin{aligned}
&\mathrm{read}(\mathbf{i})\\
&\mathbf{h} = \mathbf{A}\cdot\mathbf{i} + \mathbf{a}\\
&\forall x : \mathbf{h}[x] = 1/(1+\exp(\mathbf{h}[x]))\\
&\mathbf{o} = \mathbf{B}\cdot\mathbf{h} + \mathbf{b}\\
&\forall x : \mathbf{o}[x] = 1/(1+\exp(\mathbf{o}[x]))\\
&\mathrm{write}(\mathbf{o})
\end{aligned}
$$

In the following text, we refer to step 3 (5) of this code as the sigmoid computation of vector $\mathbf{h}$ ($\mathbf{o}$). The sigmoid computation is often implemented as table lookup. In our implentation, the elements of the input, hidden and output vector are one byte wide, the elements of the weight matrices and bias vectors are two bytes wide, and intermediate results like partial sums are 4 bytes wide.

3.1.2 Training

During training the machine also computes an output vector for each input vector, but in a second phase it compares the output with the required result ($\mathbf{r}$), computes the error vectors of the output and hidden layers ($\mathbf{eo}$, $\mathbf{eh}$), and updates the weight matrices and bias vectors. For one pattern, the formal description of the training step is given below; fac is the learning rate and $< x, y^T >$ the outer product of the vectors x and y.

Forward pass:

$$
\begin{aligned}
&\mathrm{read}(\mathbf{i},\mathbf{r})\\
&\mathbf{h} = \mathbf{A}\cdot\mathbf{i} + \mathbf{a}\\
&\forall x : \mathbf{h}[x] = 1/(1+\exp(\mathbf{h}[x]))\\
&\mathbf{o} = \mathbf{B}\cdot\mathbf{h} + \mathbf{b}\\
&\forall x : \mathbf{o}[x] = 1/(1+\exp(\mathbf{o}[x]))
\end{aligned}
$$

Error in the output layer:

$$\mathbf{eo} = \mathbf{r} - \mathbf{o}$$

Error backpropagation:

$$
\begin{aligned}
&\mathbf{eh} = \mathbf{B^T}\cdot\mathbf{eo}\\
&\forall x : \mathbf{eh}[x] = \mathbf{h}[x](1-\mathbf{h}[x])\,\mathbf{eh}[x]
\end{aligned}
$$

Weight update:

$$
\begin{aligned}
&\mathbf{B} = \mathbf{B} - fac\cdot < \mathbf{eo}, \mathbf{h^T} >\\
&\mathbf{b} = \mathbf{b} - fac\ \cdot\mathbf{eo}\\
&\mathbf{A} = \mathbf{A} - fac\cdot < \mathbf{eh}, \mathbf{i^T} >\\
&\mathbf{a} = \mathbf{a} - fac\ \cdot\mathbf{eh}
\end{aligned}
$$

In this report, we refer to the training step without the forward pass as the *backward pass*. In our implementation, the elements of the error vectors are two bytes wide, the width of the other data is as described before.

3.2 Parallelization and data distribution

Both kernels mainly consist of matrix-vector operations, and it is well known how to parallelize these. Additional parallelism could come from the fact that these algorithms have to be executed for a huge number of patterns. The data and control flow graphs in figure 3.1 can show whether it is possible to pass several patterns at a time through the network.

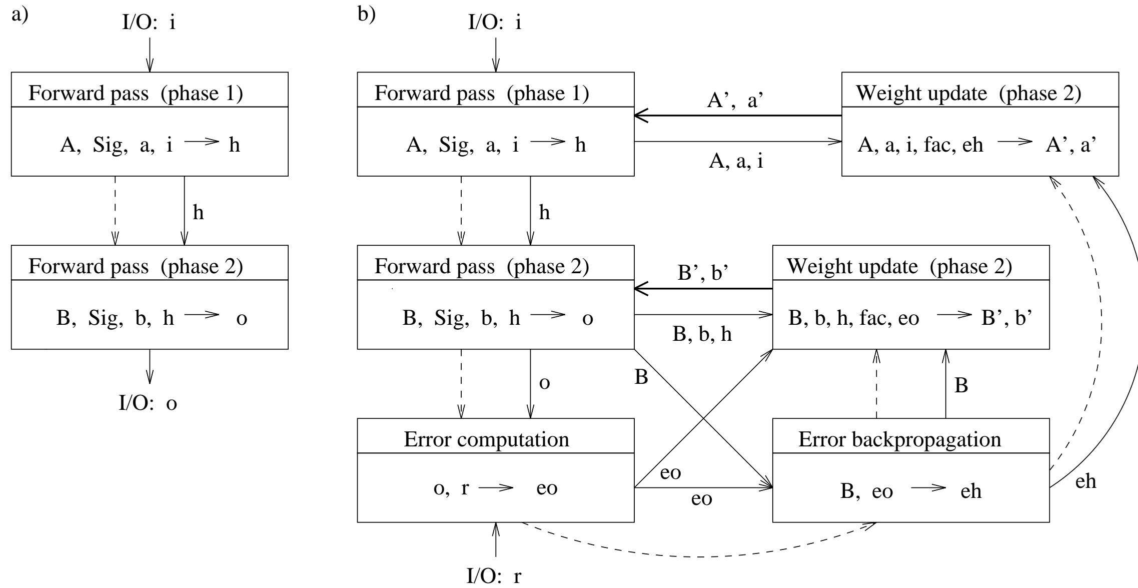


Figure 3.1: Data flow and high level control flow of recall (a) and training (b). Sig stands for the sigmoid lookup table. Dashed arrows indicate the control flow, solid arrows indicate the data flow. The weight updates change the matrices of the next iteration, these dependencies are indicated by fat arrows.

Recall There are only a few data dependencies in the recall code, all of them are based on input or result vectors. The flow graphs are acyclic and several patterns can therefore be executed in parallel. This makes recall a highly parallel problem and gives us mainly three options for parallelization:

- Both weight matrices are spread over the whole multicomputer but the processors execute each procedure for several patterns at a time.
- The multicomputer is divided into two groups of processors. One group executes phase 1 in parallel the other group executes phase 2. The execution of both phases is pipelined.
- The processing nodes are divided in several groups. The neural network is duplicated, and one copy is assigned to each group of processors. The groups work in parallel but on different input patterns.

The third version stores several copies of the network data, and therefore needs much more storage space then the other two versions. That would definitely be a problem, especially for large neural networks. In the following, we will therefore only consider the first two options.

Training The weight updates change the information stored in the neural network, that causes cycles in the data flow graph. One cycle even includes the first and the last procedure of the training code. Consequently, the processors first have to finish one pass through the network before they can start the next pass. Training therefore has less parallelism than recall. The parallelism comes from the matrix vector operations or from the branches in the control flow.

3.2.1 Current approach

The bias vectors are mapped into the last column of the connection matrices. For performance reasons we store the extended matrices in column-major order. They are distributed blockwise over all processors. The local matrices have the dimensions $n_{pi} \times n_{ph}$ and $n_{ph} \times n_{po}$, with $n_{pi} = n_i/p_i$, $n_{ph} = n_h/p_h$ and $n_{po} = n_o/p_h$; $p_h \geq p_i$. Besides these matrices, the nodes also store the corresponding parts of the inputs, hidden units and outputs; p_i processors share the same inputs and outputs, and p_h processors deal with the same hidden units. Nodes sharing the same hidden units are located in a ring of the processor network. Nodes sharing inputs are arranged in neighbored columns. Figure 3.2 illustrates the distribution for the 128-node machine.

The dimensions of the local matrices are chosen so that most of the activations and results can be stored in the on-chip caches. This implies that n_{pi} and n_{ph} nearly have the same size. We also make the restriction that n_{pi} is a multiple of four, and the other two dimensions are multiples of 64. For our analysis we assume that the dimensions of the matrices are powers of two.

This distribution can be used for recall and training, makes a good use of the on-chip data cache, and keeps the amount of transfer fairly small. There occur p_h global sums when computing a hidden vector. The global sums involve p_i nodes each, and can be executed in parallel. p_i global sums have to be executed in parallel when computing a vector of the input or output layer. There are p_h nodes involved in each of them. This transfer can widely be overlapped with computation, because of active messages.

Kumar et al. analyzed in [KSA93] several parallel formulations of the backpropagation algorithm for multilayered feedforward networks. They showed that the blockdistribution we are using is optimal for hypercubes and related architectures when using per-pattern training. We still have to addapt their formulation, because the CNS-1 has a barrel topology, uses active messages and has a powerful vector unit.

3.2.2 Alternative approach

During recall, it is possible to compute the hidden and output vectors in a pipelined manner. One part of the machine deals with the first three steps of recall, the other part only deals with the remaining three steps. The first group of processors reads the input vector,

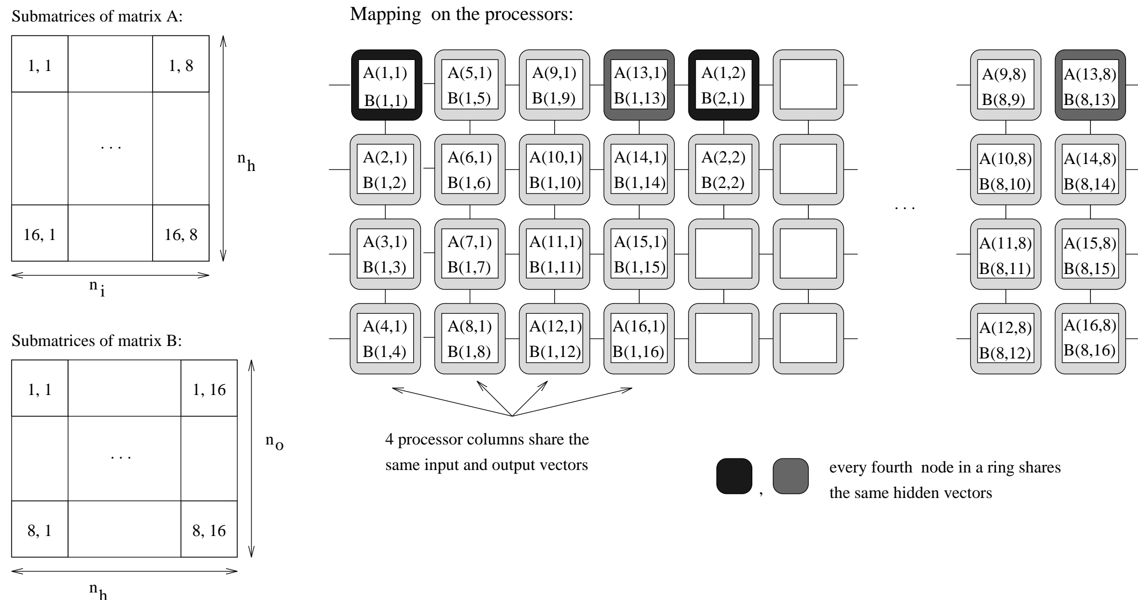


Figure 3.2: Mapping the submatrices on the 128-node machine. Four neighbored columns share the same input and output vectors. In a fixed ring, every fourth node belongs to the same group. Nodes of such a group share the same hidden vectors.

broadcasts it to each node in the group, computes the hidden vector and send it to the next group of processors. The second group receives the hidden vector and broadcasts it to each node in the group, computes the output vector and writes the result to the outside world. After sending the hidden vector to the other group, the first processors can already deal with the next input pattern.

This data distribution also requires only a small amount of transfer, and most of that transfer will not be visible, because it can be overlapped with computation. But compared with the previous distribution, this version has two major drawbacks. That is the reason why we chose the first distribution.

First, this distribution can not efficiently be used in the training of networks. Half of the processors would be idle because of data dependencies. Training is much more timeconsuming than recall, and therefore a performance loss of 50% really hurts.

Second, each node has to see the whole input or hidden vector. These vectors are usually too large for the on-chip data cache, therefore they have to be accessed in the main memory. The main memory also handeling two matrix accesses already is a bottleneck. Additional memory requests therefore slow down the execution.

Chapter 4

Performance Analysis of Kernel 1

Kernel 1, the recall of a dense, three-layer neural network, has to be executed for several patterns; two of them are passed through the network at the same time. That makes it much easier to hide the latency of the main memory. The kernel works in four steps.

The I/O nodes get two activations from the outside world and distribute them over the whole machine. In the second step, the processing nodes compute the values of the hidden units. Third, the nodes combine these values with the second connection matrix and produce two result vectors. During the fourth step these vectors are stored in main memory, and the processors in the bottom row send the results to the I/O nodes. They transfer the data to the outside world. The second and the third step use the same procedures, but with different data.

Each of the four steps has two types of procedures, one with network activity and one with local computation or I/O. Procedures of different types can be interleaved. That makes it possible to hide the latency of the processor network.

The I/O nodes and the processing nodes run different programs. The following pseudocode (Table 4.1) shows the structure of both programs. Code optimization is considered later on. Variables starting with R or r refer to the abstract use of vector and scalar registers.

4.1 Step 1: Reading the activations

This step runs on several levels: outside world, I/O nodes, and processing nodes. The I/O nodes get the data from the outside world and distribute them over the processing nodes. The processors receive their part of the activations and store them in the data cache and the main memory.

We assume that the outside world communicates with the CNS only via one fourth of the I/O nodes; the data have to be transferred over $32/4 = 8$ H-ports, each with a bandwidth of 64 bit per 20 ns (400MB/s).

One Hydrant has to read the activations for four columns of processing nodes. In the large machine (1024 nodes) all these Torrents share the same activations, but in the 128-node machine only two columns of processors use the same activations.

Hydrant program	**Torrent program**
for i=1, # patterns, 2 { for j=1, npi, 128 read(Vi1[j:j+127]) for j=1, npi, 128 read(Vi2[j:j+127]) for j=1, npo, 128 read(Vo2[j:j+127]) for j=1, npi, 128 read(Vo2[j:j+127]) }	for i=1, # patterns, 2 { get-activ(Vi1, Vi2, npi) layer(1, npi, nph, M1, Vi1, Vi2, Vh1, Vh2) layer(2, nph, npo, M2, Vh1, Vh2, Vo1, Vo2) out-res(Vo1, Vo2, npo) }

Table 4.1: Coarse structure of the first step

4.1.1 I/O on the 128-node machine

The nodes $H_a, \cdots, H_d$ are four neighboring Hydrants. H_b and H_a broadcast the same vectors to the Torrents and so do H_c and H_d. An H-port is connected to H_b which has to read the data for both groups of processors. The different pseudocodes are shown in Table 4.2. The first argument in a send instruction codes how far the message has to go up a column and the second argument codes the number of hops in the ring.

The run time of this step mainly depends on three quantities: the overhead in the Hydrant network, the time for distributing 128 bytes to the Torrents, and the time between two of these broadcasts. Transfers are either single transfers or multicasts; both take $T_{ST}(h_{op})$ cycles. All messages are 137 bytes long.

Timing in the Hydrant network

Node H_b reads two vectors of 128 bytes each via the H-port. That takes 6 cycles per vector. There also occurs some overhead based on protocols. Therefore we assume that the reading of 128 bytes takes 20 cycles; resulting in 40 cycles for reading two vectors. The CPU then starts the broadcast to the nodes H_c and H_d, and the transfer to node H_a. That takes $2 \cdot T_{p,s} = 20$ cycles. Afterwards H_b broadcasts the vector of the previous iteration along the network column. That keeps the CPU busy for 10 cycles. In the next $1 + l_{oop}$ cycles the CPU saves the current vector in a vector register and executes the loop overhead.

Two transfers with the same sender and receiver must be separated by $m_{length}+1$ cycles. That implies that one pass of the loop takes

$$max(40 + 3T_{s,p} + 1 + l_{oop}, m_{lenght} + 1) = max(71 + l_{oop}, 138) = 138$$

cycles. The first part of the H_b code is almost the same as the loop body, only the broadcast to the Torrents is missing. The node H_b therefore starts the first transfer to the Torrents after $m_{length} + 1 + 40 + 2T_{s,p} = m_{length} + 61$ cycles.

Node H_a distributes the first 128 bytes after $40 + T_{p,s} + T_{ST}(1) = 91 + m_{length}$ cycles, H_c after $40 + T_{ST}(1) = 81 + m_{length}$ cycles, and H_d two cycles later than H_c. Therefore the

Code for Hydrant H_b

```
get-byte(128, Ri1[0:31]);
get-byte(128, Ri2[0:31]);
bcast(0, 2, Ri1[0:31]);
send(0, -1, Ri2[0:31]);
Ri3[0:31] = Ri2[0:31];
for j=129, 2*npi, 128
{ get-byte(128, Ri1[0:31]);
  get-byte(128, Ri2[0:31]);
  bcast(0, 2, Ri1[0:31]);
  send(0, -1, Ri2[0:31]);
  bcast(32, 0, Ri3[0:31]);
  Ri3[0:31] = Ri2[0:31];
}
bcast(32, 0, Ri3[0:31]);
```

Code for Hydrant H_a, H_c, H_d

```
for j=1, 2*npi, 128
{ receive(128, Ri1[0:31]);
  bcast(32, 0, Ri1[0:31]);
}
```

Code for Torrent

```
for j=1, npi, 128
{ receive(128, Ri1[0:31]);
  Vi1[i:i+127] = Ri1[0:31].b;
}
for j=1, npi, 128
{ receive(128, Ri1[0:31]);
  Vi2[i:i+127] = Ri1[0:31].b;
}
```

Table 4.2: Pseudocode for reading and distributing the activations

startup in the Hydrant network takes between $61 + m_{length}$ and $92 + m_{length}$ cycles. Further broadcasts to the Torrents can be started every 138 cycles. This time is also sufficient for reading new data and sending them to the other Hydrants.

Timing in the Torrent network

The Torrents in the top row of the machine receive the message $T_{ST}(32) = 103 + m_{length} = 240$ cycles after the corresponding Hydrants started the transfer. New data arrive every $m_{length} + 1 = 138$ cycles. The CPU needs ten cycles to receive the data from the network interface. The remaining 127 cycles can be used for storing the data in the cache and the main memory and executing loop overhead. The run time of these operations is visible only when storing the last vector. We can assume that the last write is a hit, so storing a vector in the cache and main memory takes $4 + 10 = 14$ cycles.

In the worst case, code and data are not in the RDRAM cache, and the cache lines are dirty. Loading a block of 32 instructions takes 45 cycles. The data can be stored in $(120 + 2 \cdot 128)/8 = 47$ cycles (4.5 Mb memory chip). Even with loop overhead that still fits into the 127-cycle slot without additional time penalty.

Penalty for I-cache misses

Node H_b has the longest program. The code of its main loop should still fit into two blocks. H_b also executes the code of three message handlers with at most 32 instructions each. The processor loads five blocks for the main procedure and three for the handlers. At worst, all these codes use the same space in the I-cache. Then the processor has to load four blocks

```
get-byte(128, Ri1[0:31]);                 bcast(0, 3, Ri1[0:31]);
bcast(0, 3, Ri1[0:31]);                   bcast(32, 0, Ri3[0:31]);
Ri3[0:31] = Ri1[0:31];                    Ri3[0:31] = Ri1[0:31];
for j=129, 2*npi, 128                   }
{ get-byte(128, Ri1[0:31]);             bcast(32, 0, Ri3[0:31]);
```

Table 4.3: Pseudocode of kernel 1 for the Hydrants H_a

per loop iteration. That makes $2 + 5n_{pi}/64$ blocks. Each block prolongs the execution time by 44 cycles.

The following formulas show the kernel run time and the penalty caused by I-cache misses. Each node handles one procedure per pass of kernel 1; that requires T_{proc} cycles. RDRAM refreshes can increase the run times at most by a factor of τ_{ref}.

$$\begin{aligned} T_{step1} &= H_{startup} + \frac{n_{pi}}{64}(m_{length} + 1) + 14 + 102 + T_{proc} \\ H_{startup} &\leq 92 + m_{length} = 229 \\ P_{enalty} &\leq 44\left(2 + \frac{5n_{pi}}{64}\right). \end{aligned}$$

There are two ways to hide the startup time of the Hydrant network. First, the Hydrants know whether they have to deliver further patterns or not. Therefore they can read the next two patterns and distribute them in their local network during the computation steps 2 and 3. The broadcast to the Torrents is then triggered by the end of step 4. Second, the Hydrants also have D-caches. During step 2 they read as many data of the next two patterns as they can store in the cache.

4.1.2 I/O on the 1024-node machine

On the large machine, four columns of Torrents share the same activations. That makes the I/O-step a bit simpler. The code for the Torrents and Hydrants with no H-port are the same; only the nodes with H-port execute different code, and the partitioning of the Hydrants has changed. Now H_a has the H-port; it reads the activations and distributes them to the three neighboring Hydrants H_b, H_c and H_d via a multicast. Afterwards all four nodes broadcast the vector to the Torrents (Table 4.3).

These changes only influence the overhead in the Hydrant network. The network bandwidth still limits the time between two broadcasts to the Torrents to $m_{length} + 1$ cycles. Node H_a delivers the first vector to the Torrents after $m_{length} + 1 + 20 + T_{p,s} = m_{length} + 31$ cycles. H_b starts the broadcast after time $20 + T_{ST}(1) = 61 + m_{length}$; H_c starts it two cycles later and H_d four cycles later. Now the startup varies between $31 + m_{length}$ and $65 + m_{length}$ cycles.

```
for i=0, np2 - 1, 64;                                /* global sum */
{ /* matrix vector multiplication */                 gsum(Rt1[0:63],st);
  Rt1[0:63] = Rt2[0:63] = 0;                         gsum(Rt2[0:63],st);
  for j=0, np1 - 1, 4
  { rd1[0:3] = Vi1[j,j+3];                           /* sigmoid table lookup */
    rd2[0:3] = Vi2[j,j+3];                           Rt1[0:63] = clip(Rt1[0:63]);
    for k=0, 3                                       Rt2[0:63] = clip(Rt2[0:63]);
    { Rt1[0:63] += rd1[k] * M[j+k][i;i+63];          Vo1[0:63] = Sigm[Rt1[0:63]];
      Rt2[0:63] += rd2[k] * M[j+k][i;i+63];          Vo2[0:63] = Sigm[Rt2[0:63]];
    }                                              }
  }
```

Table 4.4: Pseudocode for the forward routine of kernel 1

4.2 Steps 2 and 3: Computation of the next layer

Steps 2 and 3 execute the same code but on different data. The code only runs on the processing nodes. Their pseudocode is shown in Table 4.4; n_{p1} and n_{p2} are the dimensions of the local activation and result vectors. The parameter st specifies whether the second or third layer should be computed; st only influences the code for global sum. For better performance, the assembler code uses some optimizations which are not shown in the pseudocode. The inner loop of the matrix vector multiplication is unrolled, and the outer loop is software pipelined. The execution of the three routines of this kernel – matrix vector multiplication, global sum and sigmoid computation – are interleaved.

The local data can be stored at different levels of the memory hierarchy; this influences the run time of the kernel. There are mainly three versions:

- D-cache stores all local activations and results of one layer,
- D-cache only stores the local activations.
- D-cache only stores parts of the local activations and results.

The first case is assumed to be the default, but that is not feasible for large local vectors and a limited cache size. In a separate section we analyze those cache problems.

At the end of step 1 the two local input vectors are stored in the on-chip cache. During step 2 the processors combine the inputs with the first weight matrix and produce values for the hidden units. They store these results in the cache. The results of the second step are the inputs of third. Therefore the inputs of step 3 are also in the D-cache when step 3 starts its execution.

4.2.1 Matrix vector multiplication

The activation are one byte wide; and the CPU can load four activations from the D-cache with one word access. Later on the CPU extracts the data byte by byte. The weights are

RDRAM	Hit	Miss		Hit	Miss	
		/dirty	dirty		/dirty	dirty
	r_h	$r_{m,/d}$	$r_{m,d}$	r_h	$r_{m,/d}$	$r_{m,d}$
4.5 Mb	$48+2\cdot 32$	$152+2\cdot 32$	$152+2\cdot 32$	14	27	27
18 Mb	$44+2\cdot 32$	$112+2\cdot 32$	$156+2\cdot 32$	14	22	28
	[ns]			[cycles]		

Table 4.5: Reading 32B via one Rambus port

directly loaded from the main memory, bypassing the D-cache. The processor accesses 32 elements with two bytes each. The data are 32B aligned, so two memory ports will execute the request in parallel. This access time depends on the actual status of the main memory and the size of the RDRAM chips (Table 4.5).

Cache analysis

There are 16 memory chips, each has two cache lines of size l_{RDC}. For small memory chips l_{RDC} is 1KB, and 2KB for large chips (Table 2.1). The weights are two bytes wide, so a whole memory cache line can hold $16l_{RDC}/2$ weights or $\alpha = 8l_{RDC}/n_{p2}$ rows of weights.

Loading a matrix stripe of 64 columns, the memory cache line has to be updated n_{p1}/α times; and there are β banks involved per update. The banks are interleaved every 128B respective 64 weights. In the best case, $n_{p1}/64$ is a multiple of four and all matrix rows start in the same bank ($\beta = 1$). When that number is only a multiple of two, then $\beta = 2$ banks are involved per cache update, and in every other case even all four banks are involved.

Therefore, there occur $2\beta n_{p1}/\alpha$ memory cache misses, when loading 64 matrix columns. Only half of these misses can be seen, because one vector load uses two memory ports and two loads can be overlapped. All the other weight accesses are hits. The memory has only two cache lines, and all the memory accesses during the forward step are reads. For these reasons only $\max(2\beta, \beta n_{p1}/\alpha)$ cache misses can have a dirty cache line.

Run time of the matrix vector multiplication

Some weight vectors can be loaded from the RDRAM caches, others are directly loaded from the RDRAM. A load hit takes only 14 cycles and its run time is dominated by the 16 cycle computation time. A load miss, however, takes between 22 and 28 cycles and dominates the computation time. A detailed analysis of the assembler code (Appendix A.1) therefore yields the following run time for one pass of the matrix vector multiplication code:

$$\begin{aligned} T_{MVM} &= r_{m,d} + (n_{p1}-2)\max(16, r_h) + 20 \\ &\quad + \beta\frac{n_{p1}}{\alpha}(r_{m,/d} - 16) + \min\left(2\beta - 1, \beta\frac{n_{p1}}{\alpha} - 1\right)(r_{m,d} - r_{m,/d}) \\ &= r_{m,d} + 16n_{p1} - 12 \\ &\quad + \frac{\beta n_{p1} n_{p2}}{8l_{RDC}}(r_{m,/d} - 16) + \min\left(2\beta - 1, \frac{\beta n_{p1} n_{p2}}{8l_{RDC}} - 1\right)(r_{m,d} - r_{m,/d}) \end{aligned}$$

	128 Nodes		1024 Nodes	
p	$p_i = 8$	$p_h = 16$	$p_i = 16$	$p_h = 64$
Processor Subnet	every 4th node per row	4 columns, 4 nodes each	every 2nd node per row	2 columns, 32 nodes each
T_{gsum}	$\sum_{i=2}^{4} T_{MT}(2^i)$ $+20 \log p$	$\sum_{i=0}^{1} 2\, T_{MT}(2^i)$ $+20 \log p$	$\sum_{i=1}^{4} T_{MT}(2^i)$ $+20 \log p$	$\sum_{i=0}^{4} T_{MT}(2^i)$ $+T_{MT}(1)$ $+20 \log p$
	$28\, m_{length}+$ $56 + 59 \log p$	$6\, m_{length}+$ $12 + 59 \log p$	$30\, m_{length}+$ $60 + 59 \log p$	$32\, m_{length}+$ $64 + 59 \log p$
	Step 2	Step 3	Step 2	Step 3

Table 4.6: Run time for global sum [cc]

In our investigation n_{p1} is a power of two and at least 512, therefore $\beta = 1$.

4.2.2 Global sum

The four global sums are executed by messages handlers. One handler computes the direction of the message and sends the vector Rt; the other handler receives a vector and adds it to the register Rt. We assume that this administrative overhead takes about 20 cycles.

Let p be the number of processors which share the same result vector in steps 2 or 3; then a global sum is executed in $\log p$ iterations. The run time of one iteration depends on the size of the CNS and the distribution of the matrices. The different cases are listed in Table 4.6.

4.2.3 Sigmoid computation

The four sum vectors are still in the vector registers, but the data have the wrong format. That is corrected by shift and clip instructions. The sigmoid table is stored in main memory, but after the preload step the whole table is present in the RDRAM caches. Nevertheless, a table access still needs more time than an arithmetical operation and a D-cache access. For that reason the sigmoid computation of four vectors takes time $T_{sig} = T_{preload} + 4T_{table} + 4$; 4 cycles are necessary for storing the last result vector in the D-cache. The preload and table access times depend on the size of the sigmoid table and the access patterns. The table has 16K or 32K elements. The access patterns depend on the inputs, and even under the assumption that all accesses are spread equally over the whole table, we have to analyze the best and the worst case.

Memory analysis

4.5Mb memory chip Each of the 16 memory chips has two 1KB cache lines. The 16K table therefore fits in one cache line, while the 32K table occupies two lines. One vector word access is sufficient to load a cache line of a memory bank. So a preload of 16K data

takes $4(152+64)$ ns or $T_{preload} = 4 \cdot 27 = 108$ cycles. Preloading the larger sigmoid table takes 216 cycles, twice the time $T_{preload}$.

After the preload, each table access is a hit, and the table access time only depends on the access pattern. In the best case all 32 requests are equally distributed over the memory ports, then four request can be overlapped. At worst, all requests use the same memory port and have to be executed sequentially. It takes $(48+2)$ ns or 7 cycles to load one byte from the memory cache. The table access for four vectors takes in that case:

$$4\,T_{table} = \begin{cases} 4 \cdot \frac{32}{4} \cdot 7cc + 3cc &= 227cc \qquad \text{best case} \\ 4 \cdot 32 \cdot 7cc &= 896cc \qquad \text{worst case.} \end{cases}$$

18Mb memory chip The analysis is almost the same as in the previous case, but now the cache line is twice as large. Even a table with 32K elements fits in a single cache line and can be loaded in four steps. The memory is also slightly faster. A preload takes only $4(112+64)$ ns or 88 cycles. This time is based on a clean cache miss. That is realistic because the processors do not write into the main memory during steps 2 and 3. It takes $(44+2)$ ns or 6 cycles to load a byte from the memory cache. The table lookups of four vectors can now be done in the time:

$$4\,T_{table} = \begin{cases} 4 \cdot \frac{32}{4} \cdot 6cc + 3cc &= 195cc \qquad \text{best case} \\ 4 \cdot 32 \cdot 6cc &= 768cc \qquad \text{worst case.} \end{cases}$$

4.2.4 Interleaving the three main routines

Global sum is the only routine with network traffic, and its code is executed in message handlers. For these reasons the four global sums can run in parallel with the next matrix vector multiplication. From time to time the handlers interrupt the multiplication, but only $8 \log p$ times.

It is better not to interrupt the sigmoid computation, because an interrupt could cause an I-cache miss which swaps the sigmoid table out of the memory cache. Afterwards it takes up to four memory accesses to load the table again. It should be possible to avoid these undesirable interrupts by scheduling the sigmoid computation as a high priority thread. Figure 4.1 shows the execution scheme of steps 2 and 3.

There occurs some overhead in the forward step. The processors handle two procedures, and three threads per pass of the outer loop. One procedure handling takes time T_{proc} and the handling of a thread time T_{thr}. The nodes also have to load five parameters per procedure. Only the first load instruction is a memory cache miss. In the small memory system it takes $t_{par} = (20 + 4 \cdot 7) = 48$ cycles to load five words; the large system needs $t_{par} = (21 + 4 \cdot 7) = 49$ cycles. The following formula therefore describes the run time of steps 2 and 3:

$$\begin{aligned} T_{step2,3} &= T_{MVM}(n_{pi}, n_{ph}) + \left(\frac{n_{ph}}{64} - 1\right) \max\left(T_{MVM}(n_{pi}, n_{ph}) + 40 \log p_i,\, 4T_{Gsum}(p_i)\right) \\ &+ \frac{n_{ph}}{64} T_{Sig} + \max\left(T_{MVM}(n_{ph}, n_{po}) + 40 \log p_i,\, 4T_{Gsum}(p_i)\right) \\ &+ \left(\frac{n_{po}}{64} - 1\right) \max\left(T_{MVM}(n_{ph}, n_{po}) + 40 \log p_h,\, 4T_{Gsum}(p_h)\right) + \frac{n_{po}}{64} T_{Sig} \end{aligned}$$

$$+ \quad 4T_{Gsum}(p_h) + 2\,T_{proc} + 2 \cdot t_{par} + 3\frac{n_{ph}+n_{po}}{64}T_{thr}.$$

4.2.5 Penalty of I-cache misses

The I-cache is organized in 32 blocks with 32 instructions each. The sigmoid computation needs less than 32 instructions, and the matrix vector multiplication fits in five blocks or fewer. The send handler of global sum is coded in two blocks and the receive handler in one block. Steps 2 and 3 have almost the same code, only the send handlers are different. All together the code needs twelve blocks or fewer and should fit in the I-cache, but a bad mapping can avoid that.

At worst, all routines have to share the same cache line. That causes a lot of I-cache misses. One miss stalls the machine up to 44 cycles and can also influence later data accesses.

Amount of I-cache misses

The processors load the code of the matrix vector multiplication during the first iteration of step 2, that causes five I-cache misses. Only one of them is dirty because the code fits in one memory cache line and there are no memory writes in our version of the forward step.

In later iterations, the global sum message handlers interrupt the matrix vector multiplication. Both handlers together have three blocks of code; that means three misses per two interrupts. The two blocks sigmoid code cause two misses per iteration. At worst, the multiplication code has to be restored after each iterupt. That doubles the misses, caused by message handlers and sigmoid computation. The last iteration of step 3 is different. The message handlers and the sigmoid code do not share the cache with the multiplication, so there is no restoring and the message handlers are resident in the cache during the whole global sum computation. The following formula describes the amounts of I-cache misses N_{Im} during steps 2 and 3, each miss stalls the CPU up to 44 cycles:

$$\begin{aligned} N_{Im} &= 5 + \frac{n_{ph}}{64} \cdot 2 \cdot (12 \log p_i + 2) + \left(\frac{n_{po}}{64} - 1\right) \cdot 2 \cdot (12 \log p_h + 2) + 3 + 2 \\ &= 6 + \frac{n_{ph}}{64} \cdot (24 \log p_i + 4) + \frac{n_{po}}{64} \cdot (24 \log p_h + 4) - 24 \log p_h. \end{aligned}$$

Influence on data accesses

Loading a block of instructions destroys a cache line of a whole memory bank, the old data are gone. The next access to these data therefore causes a clean memory cache miss. Those misses occur during the matrix vector multiplication and the sigmoid computation.

Matrix vector multiplication During the matrix vector multiplication the vectors are loaded in half word mode; the execution of two vector loads is overlapped. It therefore takes two aceesses to restore the RDRAM cache again, but only one of the misses can be seen.

The amount of swapped cache lines influencing the data accesses of the matrix vector multplications can be derived from the total amount of I-cache misses N_{IM}. Misses occuring in the message handlers and in the sigmoid computation have no direct influence, only

RDRAM Version	Table Access Pattern	
	Best Case	Worst Case
4.5 Mb	$1 \cdot (20-7) = 13$	$4 \cdot 13 = 52$
18 Mb	$1 \cdot (15-6) = 9$	$4 \cdot 9 = 36$

Table 4.7: Delay of the table lookup time of four vectors caused by I-cache misses [cc]

restoring the multiplication code causes significant misses. For these reasons the amount of cache lines containing weight vectors and being swapped by I-cache misses are not higher than

$$\begin{aligned} N_{Im}(MVM) &= 4 + \frac{n_{ph}}{64} \cdot (12 \log p_i + 2) + \left(\frac{n_{po}}{64} - 1 \right) \cdot (12 \log p_h + 2) \\ &= 2 + \frac{n_{ph}}{64} \cdot (12 \log p_i + 2) + \frac{n_{po}}{64} \cdot (12 \log p_h + 2) - 12 \log p_h. \end{aligned}$$

At worst, each of the swapped weight vectors replaces a cache hit by a cache miss and extends the data access times by

$$\max(16, r_{m,/d}) - \max(16, r_h) = r_{m,/d} - 16$$

cycles; $r_h = 14cc, r_{m,/d} \geq 22cc$.

Sigmoid computation The processors load two blocks of sigmoid code during one iteration, but only the second access occurs after the preload of the sigmoid table. At worst, that I-cache miss swaps parts of the sigmoid table out of the RDRAM cache. Accessing and restoring the table then causes up to t_{res} visible data misses in the RDRAM caches. Each byte access only restores the cache line of a single memory chip.

The best access pattern enables the CPU to schedule requests to all four memory ports at a time; then the table can be restored with one table lookup ($t_{res} = 1$). At worst, no memory accesses can be overlapped and the reload of the table takes up to four requests ($t_{res} = 4$). Table 4.7 shows how these facts influence the table lookup time for different memory systems.

In the second and third step the penalty caused by I-cache misses can be at most P_{enalty} cycles;

$$\begin{aligned} P_{enalty} &= 44 \cdot N_{Im} + (r_{m,/d}(32B) - 16) \cdot N_{Im}(MVM) \\ &\quad + (r_{m,/d}(1B) - r_h(1B)) \cdot t_{res} \cdot \frac{n_{ph} + n_{po}}{64}. \end{aligned}$$

4.3 Step 4: Writing the results

This step runs on I/O nodes and processing nodes. All processing nodes save their results in the main memory; and in addition to that the Torrents in the bottom ring send the results

to the attached I/O nodes. Some Hydrants send the results to the outside world; in our case every fourth node. Before that transfer, they collect the data from their neighbors.

The timing of the Hydrants depends on the distribution of the result vectors. We therefore have to analyze the run time for 128 and 1024 nodes. In the Torrent network the run time of step 4 is independent of the machine size.

4.3.1 Timing in the Torrent network

We only analyze the timing of the processors in the bottom ring, because they have the longest run time. Besides saving the results in the main memory they also have to send the data to an I/O node. That transfer takes time $T_{ST}(1)$, and further transfers can be started every $m_{length}+1$ cycle. Each of the $2 \cdot n_{po}/128$ messages blocks the CPU only for 10 cycles. In the remaining time the CPU saves the results. The vectors are already in vector registers, and storing one vector register takes $44+23=67$ cycles at the most, even with a dirty data miss in the small RDRAM and an I-cache miss. The CPU has more than $m_{length}-9$ cycles per vector and therefore can hide the storing completly behind the transfer.

The message handler and the code of step 4 fit in one cache block each. At worst, they require the same block in the I-cache. That slows the execution down by $44/64 \cdot n_{po}$ cycles. That yields

$$\begin{aligned} T_{step4} &= \frac{n_{po}}{64}(m_{length}+1)+41 \\ P_{enalty} &\leq \frac{11 n_{po}}{8}. \end{aligned}$$

4.3.2 Timing in the Hydrant network

The distribution of the activations and results is the same, so we can take the same mapping of the Hydrants as for step 1.

The worst case occurs in the 128-machine, because only two columns share the same results there. This implies that H_b has to get some data from H_c and send both vectors to the outside world. In the 1024-node machine four columns share the same results, and no transfer occurs between H_c and H_c.

I/O on the 128-node machine

Messages from the Torrents arrive every $m_{length}+1$ cycle and block the Hydrant processor and network interface for 10 cycles each. Node H_c therefore has $m_{length}-9=128$ cycles to send a message to H_b, and H_b has the same amount of time for receiving these data and writing both vectors. Analog to reading, one vector can be written in 20 cycles. Even with an I-cache miss that still fits in 128 cycles. The Hydrant nodes therefore can consume the results at the same speed as the Torrents send them.

4.4 Problems with the D-cache size

Till now we assumed that the local input, hidden and output vectors have 1K elements at most, and so all the local vectors required during one step could be stored in the D-cache.

For some of the systems mentioned in table 3.1 this is not feasible. In this section we analyze how those cache conflicts influence the code and the run time.

4.4.1 Steps 1 and 4

The run time of step 1 mostly depends on the time required for broadcasting the inputs from the Hydrants to the Torrents, as shown in previous analyses (section 4.1). The Torrents receive data, store them in the cache and main memory, and execute some loop overhead. The time they spent in the CPU and the memory system gets only visible when storing the last vector. In the case that the activation vectors are too large for the D-cache the cache accesses can be omitted. That speeds up the last iteration of step 1 by four cycles.

When entering step 4, the output vectors are already stored in the main memory, so only the processors in the bottom ring have to do some work. Originally these Torrents sent the data to the I/O nodes and stored them in the main memory. The memory accesses could completely be hidden behind the transfer. Now they first have to load the vectors before sending them to the Hydrants. Exept for the first load, all memory accesses can still be hidden behind transfer. The first load of 32 words is very likely to be a dirty RDRAM cache miss. The run time of step 4 therefore takes 27 or 28 cycles longer depending on the RDRAM type.

4.4.2 Steps 2 and 3

For step 2 there are two cases which can produce D-cache problems: the hidden vectors or the activation vectors are too large. We will analyze both cases.

Too large hidden vectors

The hidden vectors can not be stored in the D-cache, because that would swap the activations out of the cache, but in this step activations are accessed much more often than hidden elements. We therefore keep the activations in the D-cache and store the hidden vectors directly in the main memory.

The hidden vectors are stored at the end of the sigmoid computation, so the additional transfer will not disturb the table lookups. At worst, the store operations cause a drity RDRAM cache miss, but the hidden vectors are only one byte wide, and so the four writes can be overlapped. When using the small RDRAM, four vectors can be stored in $23+3*4 = 35$ cycles. The large RDRAM is one cycles faster. The sigmoid computation of four vectors takes now

$$T_{Sig} = T_{preload} + 4T_{table} + 4 + \begin{cases} 35 & ; \quad \text{4.5 Mbit RDRAM chips} \\ 34 & ; \quad \text{18 Mbit RDRAM chips.} \end{cases}$$

This D-cache problem only influences the sigmoid computation.

Too large activation vectors

We still keep as many activations in the D-cache as possible. This implies that even very small hidden vectors have to be stored directly in the main memory. We already analyzed in

```
for i=0, np2 - 1, 64;    /* matrix vector multiplication */
{ Rt1[0:63] = Rt2[0:63] = 0;
  for k=0, np1-1, 1024
  { C1[k:k+63] = Vi1[k:k+63];    /* loading a block of activations */
    C2[k:k+63] = Vi2[k:k+63];    /* in the cache */
    for j=k, k + 1023, 4
    { rd1[0:3] = C1[j,j+3];
      rd2[0:3] = C2[j,j+3];
      for l=0, 3
      { Rt1[0:63] += rd1[l] * M[j][i;i+63];
        Rt2[0:63] += rd2[l] * M[j][i;i+63];
      }
    }
  } ···
```

Table 4.8: Pseudocode of the matrix vector multiplication. The two input vectors with length np1 are too large for the D-cache, so they are loaded in blocks of 1KB.

the previous paragraph, how this influences the sigmoid execution time. The D-cache is too small to store the local parts of both activation vectors at the same time. The processors therefore only load chunks of one kilo byte in the cache. Table 4.8 shows the new pseudocode for the matrix vector multiplication. Variables starting with C, R or r indicate that these data will be stored in the on-chip D-cache, the vector registers or the scalar registers.

The mapping of the activation vectors requires some attention. The two activation vectors should be stored close together in the main memory to minimize the amount of RDAM cache misses. On the other hand, it should also be possible to keep korresponding parts of both vectors in the D-cache. The mapping therefore has to prevend, that corresponding parts allocate the same space in the D-cache. At worst, there are 1KB data between the two vectors.

The only difference between the new and the original code (table 4.4) is the blocktransfer of the activations. When transfering the data in word mode, the four memory ports can work in parallel. The four memory banks per port will only cause four misses, because both activation vectors completely fit into one memory cache line. The size of the memory cache line is $16l_{RDC}$, that is 16KB or 32KB. In our analysis, the local vectors will be at most 4096 elements. The transfer of a 1KB block of both vectors takes

$$4r_{m,d} + \left(\frac{2 \cdot 1024}{4 \cdot 32} - 4\right) r_h = 4r_{m,d} + 12r_h$$

cycles, using the notation of table 4.5. There occure $n_{p1}/1024 = n_{pi}/1024$ of these blocktransfers per pass through the outer loop of the matix vector multiplication

These transfers do not influence the matrix access time. One memory cache line can only keep less that eight matrix rows. So after 1024 rows the matrix access would be a

RDARM cache miss even without the blocktransfer. These cache problems slow the matix vector multiplication down by about 1.5%, as the new run time formular shows.

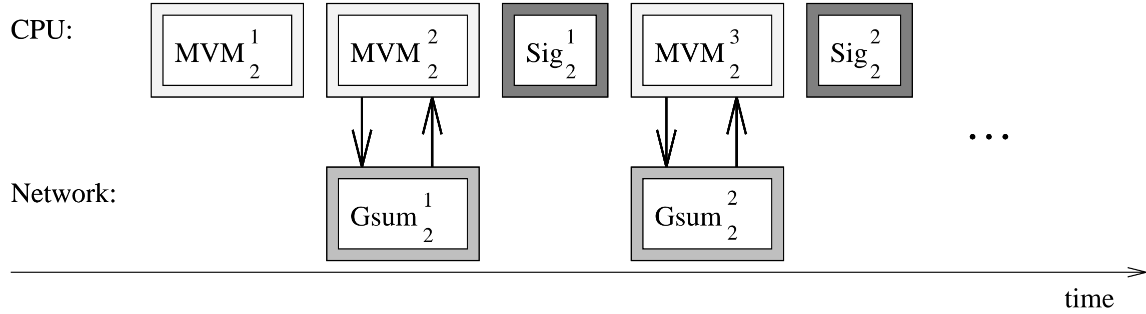
$$\begin{aligned}
T_{MVM}(n_{p1} \geq 2K) &= \frac{n_{p1}}{1024}(4r_{m,d} + 12r_h) + T_{MVM} \\
&= \frac{n_{p1}}{1024}(4r_{m,d} + 12r_h) + r_{m,d} + 16n_{p1} - 12 + \\
&\quad \frac{\beta n_{p1} n_{p2}}{8 l_{RDC}}(r_{m,/d} - 16) + \min\left(2\beta - 1, \frac{\beta n_{p1} n_{p2}}{8 l_{RDC}} - 1\right)(r_{m,d} - r_{m,/d}) \\
&\quad \text{where } \beta = 1,\ n_{p1} = n_{pi} \text{ and } n_{p2} = n_{ph}
\end{aligned}$$

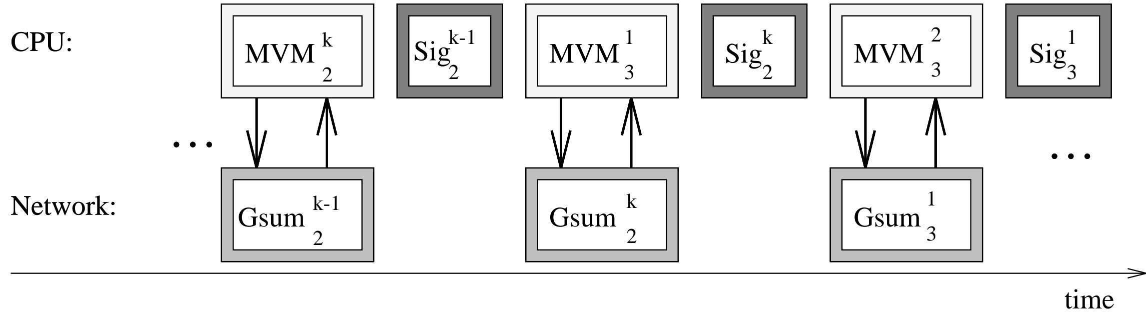
Step 3

In this step we have a similar situation than in step 2, so the same analysis can be used. The hidden vectors now play the role of the input vectors and the output vectors play the role of the result vectors. This implies that now the cache problems are caused by too large output or hidden vectors.

Besides these two cases there is also a new case. The hidden vector fits in the D-cache, but in step 2 it was written to the main memory because the activation vectors required the whole D-cache. At the beginning of step 3 all processors transfer the hidden vector from the main memory to the D-cache. This transfer is similar to the blocktransfer described earlier and therefore it takes time

$$4r_{m,d} + \left(\frac{2 \cdot n_{ph}}{4 \cdot 32} - 4\right) r_h = 4r_{m,d} + \left(\frac{n_{ph}}{64} - 4\right) r_h.$$





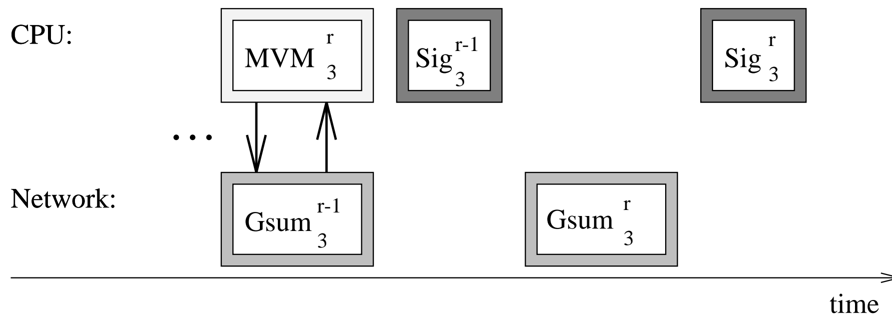


Figure 4.1: Interleaving scheme of steps 2 and 3. MVM stands for matrix vector multiplication, Sig for sigmoid computation and Gsum for global sum. The lower index indicates the step number, and the upper index counts the iterations. Step 2 is executed in $k = n_{ph}/64$ iterations and step 3 in $r = n_{po}/64$ iterations. Arrows between threads indicate that the network thread will interrupt the CPU thread from time to time, to get some computation done.

Chapter 5

Performance Results of Kernel 1

In the previous chapter we developed formulas for the run time of kernel 1 under different conditions. In this chapter we determine the CNS performance under those conditions and investigate the results. We also analyze the performance impact of the data distribution and of the parallelization scheme.

5.1 Performance results

To get performance results for different conditions, we vary the number of processing nodes, the RDRAM type, the dimensions of the neural network and the size of the sigmoid lookup table. For other parameters, like the access pattern of the lookup table and the models of memory refesh and of I-cache misses, we consider best and worst cases.

The absolute performance of the CNS running kernel 1 is measured in *connection per second* [CPS]. The amount of arithmetical operations during recall equals one multiply-accumulate instructions per connection. Each node has a peak performance of 1 GMAS (Giga multiply-accumulate per second). The peak performance for kernel 1 is therefore p GCUPS. Comparing the absolute performance with the peak performance yields the relative performance measured in %.

5.1.1 Detailed results under fixed conditions

We now investigate the performance of the 128-node machine with 4.5 Mb memory chips on a neural network comprising 192M connections. The net has 8K inputs, 16K hidden units and 4K outputs. Table 3.1 shows the corresponding values for the local problem size and the partitioning of the machine. Steps 1 and 2 are executed in 16 rounds, steps 3 and 4 in only 8 rounds. Inserting these parameters in the run time formula explains how the matrix vector multiplication, the global sum, the sigmoid computation and the I/O influence the run time of kernel 1.

The matrix vector multiplications and the global sums are the most timeconsuming routines, but their execution can partially be overlapped, because the global sums spend most of their time in the processor network. The results in table 5.1 show that the network latency can largely be hidden, at least under the given conditions.

Step	Matrix Vector Multiply	Global Sum			
		CPU	Network		
			visible	hidden	
2	17807	480	15796		per
3	17103	640	3640		Round
2	284912	7680	0	252736	total
3	136824	5120	3640	29120	

Table 5.1: Run time of the matrix vector multiplication and the global sums, based on the best case of I-cache misses ([cc]).

Sigmoid Computation					Matrix vector Multiply
Access Pattern	Best Case		Worst Case		
Table Size	16K	32K	16K	32K	
Step 2	5424	7152	16128	17856	284912
Step 3	2712	3576	8064	8928	136824

Table 5.2: Sigmoid computation time for the best case of I-cache misses ([cc]).

The sigmoid computation time depends on the size of the lookup table and the access patterns. The size of the lookup table has only a 29% impact on the run time, but the access patterns have a much stronger influence. Bad patterns can triple the sigmoid execution time of the best case (table 5.2).

The sigmoid computation contributes less than 7% to the total run time of kernel 1, and therefore changes in the size of the lookup table or in the access patterns become negligible phenomena for the run time of kernel 1. Changing the table size from 16K to 32K only results in a performance loss of less than 0.6% and bad access patterns in a performance loss of 3.5%, at most. I/O and administrative overhead add another 3% to the total run time.

Table 5.3 shows that a major performance penalty is related to I-cache misses. Actually, the code of kernel 1 is small enough to fit in the I-cache, but a bad mapping may cause that all routines have to share the same cache block. That results in a lot of cache misses, most of them related to message handler interrupts. Under the given conditions, such a bad mapping increases the total run time by about 22%.

5.1.2 General results

The previous results show that refresh and changes in the table lookup behavior have virtually no impact on the total performance. For the next analysis we therefore assume the worst case of both parameters.

As already seen before, the performance penalty for I-cache misses is quite high. At worst, these misses increase the run time by 30% (Table 5.4), and so they have a stronger

Model		Run Time [cc]				
I-Cache Misses	Table Lookup	Step 2	Step 3	I/O	Overhead	Total
Best Case	Best Case	298016	148296	3546	9378	459222
	Worst Case	310448	154512	3546	9378	477870
Worst Case	Best Case	358680	183314	7462	9378	558820
	Worst Case	371736	189842	7462	9378	578404

Table 5.3: Run time of kernel 1.

RDRAM	Connections				
Chips	192M	768M	1536M	768M	1536M
4.5 Mb	21.7	10.8	7.9	28.9	27.8
18 Mb	21.3	10.8	8.2	27.7	27.0
	128 Processors			1024 Processors	

Table 5.4: Worst case penalty of I-cache misses [%]

influence on the performance than I/O and administrative overhead. Compilers should take care of that, if possible; codes running interleaved should be stored in different cache blocks. Consecutive storing of the codes is the easiest way to reach that goal. From the hardware point of view, associative caches can also reduce the amount of I-cache misses.

In the memory model we considered two versions of RDRAM differing in capacity and speed. The CNS with the larger and faster RDRAM naturally has a better performance, but the improvement is very small (Table 5.5).

That table also shows that the 128-node machine can achieve the desired performance of $10^{11} CPS$, even under worst case assumptions for refresh and table lookup. The performance of the memory system and network are sufficient for kernel 1, only I-cache misses should be controlled carefully in software or hardware.

Kernel 1 scales quite well on the CNS. Scaling the machine from 128 to 1024 nodes reults in a speedup of over 5.5 (68%) for fixed global problem size, and in a speedup of 7.58 (94%) for fixed local problem size; that is virtually an ideal value.

5.1.3 Implications for the CNS design

The location of the scalars for vector operations and the implementation of a multiply accumulate instruction are two open aspects in the CNS design, but with the coding presented in this paper, both aspects have no influence on the performance of kernel 1.

Location of scalars for vector operations

So far, we assumed that the SIMD unit has a separate register file for scalars which are used in vector operations; the SIMD unit can not directly access the registerfile of the scalar

Node	Performance				Connections	
	Absolute [$10^9 CPS$]		Relative [%]		Total	per Node
128	105	109	82.2	85.2	192M	1536K
	104	111	81.4	86.4	768M	6144K
	97	106	75.4	82.9	1536M	12288K
1024	587	593	57.3	57.9	768M	768K
	796	819	77.7	79.9	1536M	1536K
	Size of RDRAM Chips [Mb]					
	4.5	18	4.5	18		

Table 5.5: Rounded performance values of kernel 1 on the CNS with worst case model of refresh and table lookup and best case of I-cache misses.

unit. That causes additional move instructions, because the scalar unit computes all the scalars. Anyway, there are some NOP instructions in the current assembler code of kernel 1, and so these additional moves do not influence the execution time.

On the other hand, storing the scalars of vector operations in the scalar unit can also cause problems. The requirement of scalars is very high, especially when the execution of some routines is interleaved. In such a situation it is rather likely that the scalar unit has not enough registers and some of the scalars have to be stored in vector registers or in the next memory hierarchy. For kernel 1 that would also cause no severe performance loss.

Implementation of a multiply accumulate instruction

The arithmetic vector pipeline VP0 can execute a multiply accumulate instruction, but with fixed point numbers that prohibits an independant scaling of the product and the sum, and the general performance impact of such a combined execution is still unclear. For kernel 1 there is virtually no performance penalty related to a separate execution of the multiply accumulate instructions.

With the combined execution, only one of the arithmetic vector pipelines is busy, the other pipeline idles all the time. There are also enough NOP instructions in the assembler code to schedule an add instruction for the second arithmetic without an run time impact, but when leaving the inner loop of the matrix vector multiplication the storing of the results is delayed for four cycles. In our code the matrix multiplications are executed in stripes of 64, so there are only $(n_{ph} + n_{po})/64$ outer loops. The penalty of a separate execution of the multiply accumulate instructions is therefore less than 192 cycles per two patterns or less than 0.1%.

5.2 Influence of the mapping and the parallelization scheme

We used some sophisticated coding to hide the memory and network latency and to make use of the three levels of parallelism which the CNS provides. Some of the optimizations deal with the data distribution and the coarse structure of the program; they try to reduce

the amount of visible network transfer. Other optimizations deal with local aspects like loop unrolling. They try to achieve a high memory bandwidth and to overcome other pipelining hazards. The following analysis shows the performance increase related to those code optimizations. Whenever performance numbers occur, we assume model conditions as described in section 5.1.1 and a best case model for I-cache misses.

5.2.1 Local optimizations

We mainly used three optimizations to hide the memory latency and to keep the vector pipelines busy.

Two patterns at a time

In a matrix vector multiplication there is only one multiply accumulate operation per matrix element. The matrix is usually stored in the main memory and not in the cache because of its size. Both facts imply that the amount of computation per element is very small and that it is very hard to hide the memory latency. To increase the amount of computation, we pass two patterns through the neural network at the same time. The computation is still accurate, because the weight matrix does not change during recall.

This optimization has the largest impact on the matrix vector multiplication. In the other routines, like I/O, global sums and sigmoid computation, it only increases the overhead, all of which are minor effects.

The analysis in chapter 4.2.1 shows that new weight vectors arrive at the SIMD unit every $m_h/2 = 7$ cycles, assuming an RDRAM cache hit. The vector pipeline VPO has to execute two multiply accumulate instructions per weight vector, these instructions are scheduled every 4 cycles. So the SIMD unit can consume two weight vectors every $\max(16, m_h) = 16$ cycles.

When passing only one pattern through the neural network at a time, only one multiply accumulate instruction is scheduled per weight vector. Then two weight vectors are consumed in $\max(8, m_h) = m_h = 14$ cycles, but there are twice as much passes through the neural network.

The following formula roughly estimates the performance penalty of passing single patterns. A more detailed analysis shows, that under model conditions this penalty lies between 70% and 73%.

$$\frac{2max(8, m_h)}{max(16, m_h)} = \frac{2m_h}{16} = 1.75$$

Larger matrix stripes

In our code, each of the processors consumes its matrices in stripes of 64 elements. That is double the vector length, and allows the processors to use all four memory ports.

The weights are two bytes wide, and so one vector access requires two memory ports. When loading only 32 columns, the processors always end up in the same two memory ports, wasting half the memory bandwidth. Accesses of 64 weights guarantee that all four memory ports are involved. This enables the RAMBUS controller to overlap two memory accesses; and consequently, the CPU sees only half the memory access time.

This means, using stripes of 32 elements reduces the effective memory bandwidth, and increases the run time of kernel 1 by 91%, assuming model conditions. The penalty of not using this optimization is even larger (95%), when the previous optimization is also omitted. With none of these optimizations kernel 1 is slowed down by $1.73 \cdot 1.95 = 3.37$.

Loop unrolling

The inner loop of the matrix vector multiplication is unrolled in our assembler code (Table 4.4). That increases the amount of independent operations, and therefore makes it much easier to overcome pipeline hazards.

An activation is one byte wide. Four of them are stored per memory word. After a load, the scalar unit extracts the activations byte by byte. This code depends on the position of the byte and can hardly be written as an assembler code loop. Loop unrolling is therefore a natural optimization.

5.2.2 Global optimizations

In our optimized code, we store the weight matrices in coulumn-major order, interleave the execution of the main routines, and distribute the matrices blockwise over all nodes.

Column-major ordering makes the code of kernel 1 simpler, because the reduction of a vector in a SIMD unit would cause some problems. The analysis of kernel 2 will show that the performance of the multiplication is slightly better with this ordering than with the row-major ordering.

We interleave the execution of the matrix vector multiplication, the global sum and the sigmoid computation as described in chapter 4.2.4. That allows us to hide most of the network latency. Under model conditions 97.5% of the transfer time is hidden. Without interleaving, all that transfer gets visible, and the run time of kernel 1 increases by a factor of 1.595.

We chose a blockwise distribution of the matrices for three reasons: i) it minimizes the amount of inputs, hidden units and outputs per node, ii) it, nevertheless, guarantees a decent number of columns per matrix block, and iii) it keeps the transfer requirement small.

The smaller the local vectors are, the more likely it is that they can be held in the on-chip cache. Smaller vectors therefore reduce the amount of main memory accesses. That is an important fact for the performance of kernel 1, because an earlier analysis showed, that the memory port is much busier than the other vector pipelines.

An efficient interleaving of the main routines requires a minimal number of elements per local matrix row. A blockwise distribution guarantees that for both weight matrices. It also restricts the transfer to some global sums. That makes it much simpler to hide the network latency.

Without our local optimizations, the run time of kernel 1 increases by a factor of at least 3.37, and without interleaving the main routines it increases by a factor of 1.6. Both optimizations are independant; one speeds up the matrix vector multiplication, the other hides the network transfer time. Therefore the total performance profit of our optimizations is at least a factor of 3.97.

Chapter 6

Performance Analysis of Kernel 2

Kernel 2 deals with the training of a three-layer neural network using the backpropagation algorithm described in chapter 3. In our code, the bias vectors are mapped into the last column of the weight matrices, the matrices are stored column wise, and the sigmoid computation is implemented as table lookup. We therefore get a slightly different description of the backpropagation algorithm.

Forward pass:

(1) $\text{read}(\mathbf{i}, \mathbf{r})$
(2a) $\mathbf{h} = \mathbf{A} \cdot \mathbf{i}$
(2b) $\forall x : \mathbf{h}[x] = \text{sigmoid}(\mathbf{h}[x])$
(3a) $\mathbf{o} = \mathbf{B} \cdot \mathbf{h}$
(3b) $\forall x : \mathbf{o}[x] = \text{sigmoid}(\mathbf{o}[x])$

Error in the output layer:

(5) $\mathbf{eo} = \mathbf{r} - \mathbf{o}$

Error backpropagation:

(6) $\mathbf{eh} = \mathbf{B}^{\mathbf{T}} \cdot \mathbf{eo}$
(7) $\forall x : \mathbf{eh}[x] = \mathbf{h}[x](1 - \mathbf{h}[x])\,\mathbf{eh}[x]$

Weight update:

(8) $\mathbf{B} = \mathbf{B} - fac \cdot < \mathbf{eo}, \mathbf{h}^{\mathbf{T}} >$
(9) $\mathbf{A} = \mathbf{A} - fac \cdot < \mathbf{eh}, \mathbf{i}^{\mathbf{T}} >$

In the following, we analyze seperately the forward pass, the error computation, the backpropagation and the weight update.

6.1 The forward pass

This part of the training is similar to the recall, but now only one pattern is passed through the network at a time. The processors therefore have to execute twice as much passes. These two aspects reduce the amount of parallelism and make it harder to get efficient code. Consequently, this causes major changes in the coding of the matrix vector operations.

The original recall code consits of four steps, reading activations, computing hidden and output vectors, and writing results. During training, the processors do not write results, and therefore the fourth step is obmitted. We now analyze how the run time of the steps has changed.

6.1.1 I/O during training

Originally, the processors read two patterns at a time in $2n_{pi}/128$ iterations. Now they read one input pattern and the corresponding result pattern per pass through the input procedure. That changes the amount of input data to $n_{pi}+n_{po}$ and the number of iterations per pattern to $(n_{pi}+n_{po})/128$. The amount of startup operations remains the same. Modifying the run time formulas from page 16 yields:

$$
\begin{aligned}
T_{step1} &= H_{startup} + \frac{n_{pi}+n_{po}}{128}(m_{length}+1) + 14 + 102 + T_{proc} \\
H_{startup} &\leq 92 + m_{length} = 229 \\
P_{enalty} &\leq 44\left(2 + \frac{5(n_{pi}+n_{po})}{128}\right).
\end{aligned}
$$

During training, the processors execute step 1 twice as often as during recall, assuming the same amount of patterns.

6.1.2 Computation of the next layers

The steps 2 and 3 are the same but they work on different data. There are three procedures per step – local matrix vector computation, global sums and sigmoid computation. They are interleaved in the same manner than during recall. Per pass through the neural network, the processors compute only half the amount of global sums, and per sigmoid call, they look up the values of two 32-element vectors instead of four. The preload time remains the same, but the lookup time is reduced from $4T_{table}$ to $2T_{table}$ with:

$$
2\,T_{table} = \begin{cases}
\left.\begin{array}{lcll} 2\cdot\frac{32}{4}\cdot 7cc + 3cc &=& 115cc & \text{best case} \\ 2\cdot 32\cdot 7cc &=& 448cc & \text{worst case}\end{array}\right\} \text{4.5 Mb RDRAM chips} \\
\left.\begin{array}{lcll} 2\cdot\frac{32}{4}\cdot 6cc + 3cc &=& 99cc & \text{best case} \\ 2\cdot 32\cdot 6cc &=& 384cc & \text{worst case}\end{array}\right\} \text{18 Mb RDRAM chips.}
\end{cases}
$$

Matrix vector multiplication

During one matix vector multiplication, the processors execute only half the amount of arithmetic vector operations. These operations are overlapped with the loads of the weight vectors. The number of loads remains the same. In the recall code there are scheduled 16 cycles of arithmetic operations during one vector load, now there are scheduled only 10 cycles of operations. Loading a weight vector takes between 14 and 28 cycles depending on the memory situation. Reducing the amount of arithmetical operations per weight vector therefore has only a slight impact on the run time of a matrix vector multiplication.

Some data have to be stored in the D-cache after leaving the loop. The last vector operation of the recall code can be obmitted because it only deals with the results of the second pattern. This reduces the loop overhead by four cycles. Modifying the run time formula from page 18 yields:

$$
\begin{aligned}
T_{MVM} = {} & r_{m,d} + (n_{p1}-2)\max(10, r_h) + 16 \\
& + \beta\frac{n_{p1}}{\alpha}(r_{m,/d} - r_h) + \min\left(2\beta - 1, \beta\frac{n_{p1}}{\alpha} - 1\right)(r_{m,d} - r_{m,/d})
\end{aligned}
$$

Combining the three main routines

There are only minor changes in how the run times of the three procedures combine to the total run time of the steps 2 and 3. During training these steps require two parameters less than during recall. Loading the three parameters passed in the main memory takes only $t_{par} = 20 + 2 \cdot 7 = 34$ cycles for the small memory chips and $t_{par} = 21 + 2 \cdot 7 = 35$ cycles for the large memory chips. The following formula described the new run time of the steps 2 and 3:

$$\begin{aligned} T_{step2,3} &= T_{MVM}(n_{pi}, n_{ph}) + \left(\frac{n_{ph}}{64} - 1\right) \max\left(T_{MVM}(n_{pi}, n_{ph}) + 40 \log p_i,\, 2T_{Gsum}(p_i)\right) \\ &+ \frac{n_{ph}}{64} T_{Sig} + \max\left(T_{MVM}(n_{ph}, n_{po}) + 40 \log p_i,\, 2T_{Gsum}(p_i)\right) \\ &+ \left(\frac{n_{po}}{64} - 1\right) \max\left(T_{MVM}(n_{ph}, n_{po}) + 40 \log p_h,\, 2T_{Gsum}(p_h)\right) + \frac{n_{po}}{64} T_{Sig} \\ &+ 2T_{Gsum}(p_h) + 2\, T_{proc} + 2 \cdot t_{par} + 3 \frac{n_{ph} + n_{po}}{64} T_{thr}. \end{aligned}$$

Penalty of I-cache misses

We already described in the recall analysis that I-cache misses have two run time impacts. First, loading the I-cache takes 45 cycles per block, and second, these loads allocate space in the RDRAM caches and therefore disturb succeeding data accesses.

Amount of I-cache misses The number N_{Im} of I-cache misses of the steps 2 and 3 is much smaller now than during recall, because the processors execute only half the amount of global sums. Each global sum causes $6 \log p$ I-cache misses. N_{Im} has the following value:

$$\begin{aligned} N_{Im} &= 5 + \frac{n_{ph}}{64} \cdot 2 \cdot (6 \log p_i + 2) + \left(\frac{n_{po}}{64} - 1\right) \cdot 2 \cdot (6 \log p_h + 2) + 3 + 2 \\ &= 6 + \frac{n_{ph}}{64} \cdot (12 \log p_i + 4) + \frac{n_{po}}{64} \cdot (12 \log p_h + 4) - 12 \log p_h. \end{aligned}$$

Influence on data accesses For the sigmoid computation, this impact depends on the number of preloads and not on the number of lookups. Global sums access no data in the main memory, so there occur only changes in the matrix vector multiplication code.

Message handlers and the sigmoid computation swap parts of the multiplication code out of the I-cache. Loading these code blocks then interferes with matrix accesses. Global sums are overlapped with the matrix vector multiplication, but in the training code there are only half as much global sums than in the recall code. The impact on data accesses during the multiplication therefore reduces to:

$$\begin{aligned} N_{Im}(MVM) &= 4 + \frac{n_{ph}}{64} \cdot (6 \log p_i + 2) + \left(\frac{n_{po}}{64} - 1\right) \cdot (6 \log p_h + 2) \\ &= 2 + \frac{n_{ph}}{64} \cdot (6 \log p_i + 2) + \frac{n_{po}}{64} \cdot (6 \log p_h + 2) - 6 \log p_h. \end{aligned}$$

The penalty caused by I-cache misses during the second and third step of the forward pass adds up to no more than:

$$
\begin{aligned}
P_{enalty} &= 44 \cdot N_{Im} + (r_{m,/d}(32B) - r_h(32B)) \cdot N_{Im}(MVM) \\
&\quad +(r_{m,/d}(1B) - r_h(1B)) \cdot t_{res} \cdot \frac{n_{ph} + n_{po}}{64}.
\end{aligned}
$$

6.2 The backward pass

The backward pass consits of three major parts: the error computation, the error backpropagation and the weight updates. The four instructions of the error backpropagation and the weight update can be arranged in three ways, as shown below. We chose the current arrangement to make best use of the D-cache. This cache will usually be too small to keep the input vector, the hidden vector and the two error vectors. We therefore combine the steps 6 and 7 and store the result **eh** in the main memory. We then schedule the update of matrix B and finally the update of matrix A. Before the last update the processors preload the second error vector. With this arrangement only one vector has to be shuffled around.

$$
\begin{array}{lll}
(6)\quad \mathbf{eh} = \mathbf{B}^{\mathbf{T}} \cdot \mathbf{eo} & & \\
 & \longleftarrow & \\
(7)\quad \forall x : \mathbf{eh}[x] = \mathbf{h}[x](1 - \mathbf{h}[x])\,\mathbf{eh}[x] & & \\
 & \longleftarrow & (8)\quad \mathbf{B} = \mathbf{B} - fac \cdot < \mathbf{eo}, \mathbf{h}^{\mathbf{T}} > \\
(9)\quad \mathbf{A} = \mathbf{A} - fac \cdot < \mathbf{eh}, \mathbf{i}^{\mathbf{T}} > & & \\
 & \longleftarrow &
\end{array}
$$

In our benchmark fac is constant and therefore we can do the following equivalent transformation. We only multiply the error vector **eo** with the learning rate fac instead of multiplying both weight matices. This transforamtion influences three instructions of the backward pass:

$$
(5)\quad \mathbf{eo} = fac\,(\mathbf{r} - \mathbf{o}) \quad ; \quad (8)\quad \mathbf{B} = \mathbf{B} - < \mathbf{eo}, \mathbf{h}^{\mathbf{T}} > \quad ; \quad (9)\quad \mathbf{A} = \mathbf{A} - < \mathbf{eh}, \mathbf{i}^{\mathbf{T}} >
$$

6.2.1 Step 5: Error computation

The fifth step of the training code computes the error in the output layer using the entropy error metric and scales the result by the learning rate fac. The entropy error is the difference between the output vector of the neural network and the result vector. Both arguments of the vector subtraction have n_{po} one-byte elements. The elements of the error vector are two bytes wide. All vectors are accessed in chunks of 32 elements. Table 6.1 shows the pseudocode of this step.

The error vector replaces the output vector in the D-cache. The result vectors are directly accessed in the main memory and bypass the on-chip D-cache. The memory interface can overlap four of these vector accesses. The inner loop of step 5 therefore consists of four vector subtractions. The run time of the inner loop depends on whether the result vectors already stand in the RDRAM cache or not. Only one fourth of the RDRAM cache misses can be seen because of the overlapped loads. The main memory constist of 16 RDRAM chips. The RDRAM cache has a line size of l_{RDC} per chip. We assume that n_{po} is a power

```
for i=0, npo - 1, 128;
{ RV1[0:31] = r[i:i+31];
  RV5[0:31] = o[i:1+31];
  RV2[0:31] = r[i+32:i+63];
  RV6[0:31] = o[i+32:i+63];
  RV3[0:31] = r[i+64:i+95];
  RV7[0:31] = o[i+64:i+95];
  RV4[0:31] = r[i+96:i+127];
  RV8[0:31] = o[i+96:i+127];
  RV1[0:31] = RV1[0:31] - RV5[0:31];
  RV2[0:31] = RV2[0:31] - RV6[0:31];
  RV1[0:31] = RV1[0:31] * rf;
  RV3[0:31] = RV3[0:31] - RV7[0:31];
  RV2[0:31] = RV2[0:31] * rf;
  eo[i:i+31] = RV1[0:31];
  RV4[0:31] = RV4[0:31] - RV8[0:31];
  RV3[0:31] = RV3[0:31] * rf;
  eo[i+32:i+63] = RV2[0:31];
  RV4[0:31] = RV4[0:31] * rf;
  eo[i+64:i+95] = RV3[0:31];
  eo[i+96:i+127] = RV4[0:31];
}
```

Table 6.1: Pseudocode for the error computation. The vectors **o** and **eo** are stored in the D-cache, the result vector **r** is stored in the main meory.

of two then the following formula describes the maximal number of memory cache misses for loading one result vector:

$$
\begin{aligned}
n_{miss} &= \begin{cases} \min\left(4, \frac{n_{po}}{128}\right) & ; \quad n_{po} \leq 16\, l_{RDC} \\ 4 \cdot \frac{n_{po}}{16\, l_{RDC}} & ; \quad n_{po} > 16\, l_{RDC} \end{cases} \\
&= \min\left(4, \frac{n_{po}}{128}\right) + \max\left(0, \frac{n_{po}}{4\, l_{RDC}} - 4\right).
\end{aligned}
$$

The vector units have only one load/store pipeline and so the run time of the inner loop mainly depends on the execution of the two vector loads and the one vector store. The subtraction and the multiplication can be hidden completely behind the memory transfers.

Each vector loaded from main memory or D-cache blocks the load/store pipeline for four cycles. That implies that memory accesses should not be started more often than every four cycles. Main memory accesses first check in the D-cache for the data, then they acccess the RDRAMs. When scheduling the memory loads every five cycles the remaining four cycles can be used for loading a vector from the D-cache.

Run time with RDRAM cache miss

Assuming the four vectors are not in the RDRAM cache, Then the first result data arrives after $r_{m,d} = 27$ or 28 cycles. That is enough time to start three further main memory accesses and to load four vectors from the D-cache. After time $r_{m,d}$ the next vectors arrive every five cycles. The processors combine these data with the vectors loaded from the cache and store the results in the D-cache. Diagram 6.1 shows the timing of the inner loop. VP0 and VP1 are the arithmetic vector pipelines, only VP0 has a multiplier. With an RDRAM cache miss, the processors need the time $t_{miss} = r_{m,d} + 35cc$ for one pass through the inner loop.

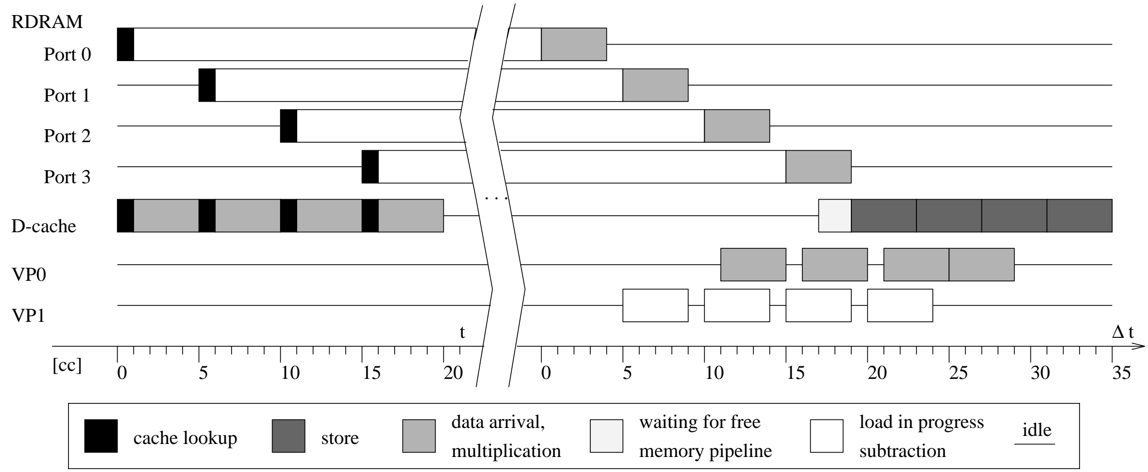


Figure 6.1: This shows how the load/store operations and the arithmetic opeartions work together in the code of the inner loop assuming an RDRAM cache miss. Δt is the time spent in the loop after the arrival of the first memory data.

Run time with RDRAM cache hit

The first memory data arrives much earlier now, already after $r_h = 14cc$. That makes the timing of the inner loop a bit more complicate. But in this case one pass through the loop only takes $t_{hit} = 53cc$. Diagram 6.2 shows how this time comes together.

Total run time of step 5

During step 5, the processors execute a procedure call, that takes time T_{proc}. This time includes the transfer of four parameters via parameter registers. Outside the loop the processors execute six instructions, four of them are overlapped with loading the fifth parameter from main memory. So the total run time of step 5 adds up to:

$$\begin{aligned} T_{step5} &= T_{proc} + r_{m,d}(4B) + 2 + t_{hit}\left(\frac{n_{po}}{4 \cdot 32} - n_{miss}\right) + n_{miss} \cdot t_{miss} \\ &= T_{proc} + r_{m,d}(4B) + 2 + 53\left(\frac{n_{po}}{4 \cdot 32} - n_{miss}\right) + n_{miss} \cdot (r_{m,d} + 35). \end{aligned}$$

Influence of I-cache misses

I-cache misses have virtually no impact on the run time of step 5. The code is very short and therefore fits into one cache block. During fetching the first instruction the whole block is transferred into the I-cache. That increases the execution of the first instruction by 44 cycles. All the other instruction fetches are I-cache hits. This step is not overlapped with transfer operation and so there occur no further I-cache misses. The only I-cache miss occurs before preloading data into the RDRAM cache, and therefore the miss does not disturb the succeeding data accesses.

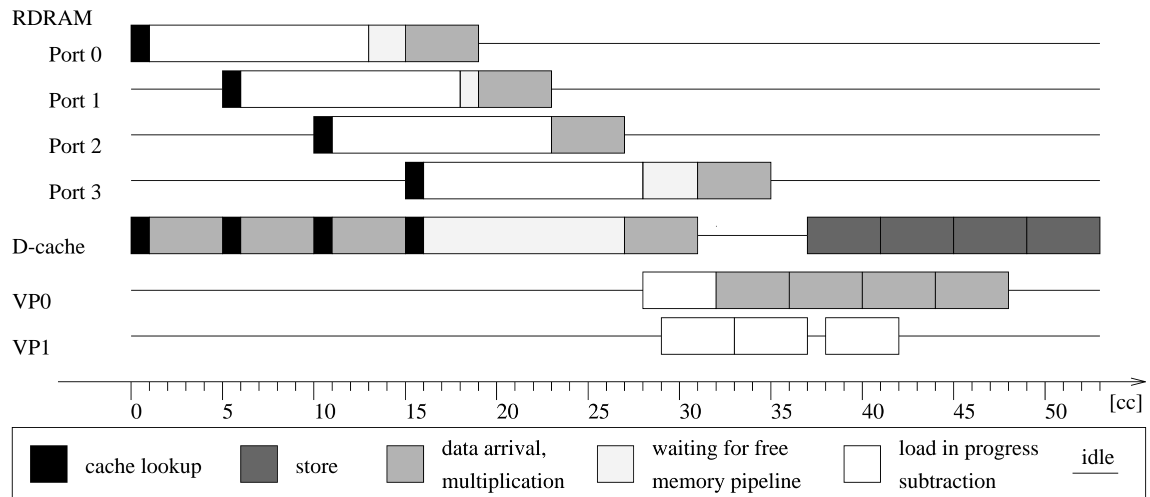


Figure 6.2: This shows how the load/store operations and the subtraction work together in the code of the inner loop assuming an RDRAM cache hit.

6.2.2 Error backpropagation

The error backpropagation involves the steps 6 and 7 which are executed together. In step 6, the processors execute a matrix vector multiplication with the transposed weight matrix, and in step 7, they scale the error vector and store it in the main memory.

Data access pattern

The local weight matrices are still stored in column-major order. Accessing the matrix per column reduces the amount of RDRAM cache misses and makes it much easier to use the vector features of the processors. This access patter has also a drawback. The processors have to reduce one vector per column of the local matrix to get an element of their result vector. They can speed up this plus-reduction when executing it for at least four vectors at a time. For the global sum it is even advantageous to execute it for whole 32-element vectors. That reduces the amount of overhead per data element.

To satisfy all three conditions, the processors access four columns of the matrix at once in chunks of 64 elements. Loading 64 elements guaranties that all four memory ports are involved in the transaction, because the ports are interleaved every 32 bytes and the matrix elements are 2 bytes wide. Table 6.2 shows the pseudocode of the error backpropagation. R and r refer to the symbolic use of vector and scalar registers.

The assembler code (appendix A.2.2) uses some optimizations which are not shown in the pseudocode to achieve a better performance. The inner loop (index k) is software pipelined, and in the outer loop the execution of the three main routines – local computation, the global sum and the scaling – is interleaved.

```
for i=0, nph - 1, 32;
{ for j=0, 7
  { RV3[] = RV4[] = RV5[] = RV6[] = 0;
    for k=0, npo -1, 64
    { RV1[] = eo[k:k+31];
      RV2[] = eo[k+32:k+63];
      RV3[] += RV1[] * B[i+j][k:k+31];
      RV3[] += RV2[] * B[i+j][k+32:k+63];
      RV4[] += RV1[] * B[i+j+1][k:k+31];
      RV4[] += RV2[] * B[i+j+1][k+32:k+63];
      RV5[] += RV1[] * B[i+j+2][k:k+31];
      RV5[] += RV2[] * B[i+j+2][k+32:k+63];
      RV6[] += RV1[] * B[i+j+3][k:k+31];
      RV6[] += RV2[] * B[i+j+3][k+32:k+63];
    }
    plus-reduction(Rv3[],RV4[],RV5[],RV6[]);
    Vh[4*j]=RV3[0];
    Vh[4*j+1]=RV4[0];
    Vh[4*j+2]=RV5[0];
    Vh[4*j+3]=RV6[0];
  }
  RV7[] = Vh[];
  gsum(RV7[]);
  /* scaling eh */
  RV8[] = h[i:i+31];
  RV7[] = RV7[] * RV8[];
  RV8[] -= 1;
  RV8[] = RV8[] * RV7[];
  eh[i:i+31] = RV8[];
}
```

Table 6.2: Pseudocode for the error backpropagation. The vectors **h**, **eo** and the intermediate results **Vh**$[0:31]$ are stored in the D-cache. The weight matrix **B** and the vector **eh** are accessed in the main memory. RV[] is an abreviation for RV$[0:31]$.

Local computation

We refer to the two inner loops (indices j and k) of the backpropagation pseudocode as the local computation. This part of the code and one global sum are executed in parallel.

Cache analysis There are two memory ports involved per load of a 32-element weight vector. Two of these loads are overlapped. The memory system has four ports with four chips each. Therefore there occur 16 misses when loading one memory cache line, but with the above access pattern only one fourth of them is visible. One cacheline has the size $16l_{RDC} \geq 16KB$. So the part of the matrix read in the two inner loops requires $2B \cdot 8 \cdot 4n_{po}/(16l_{RDC}) = 4n_{po}/l_{RDC}$ memory cache lines. In the local computation there are

$$n_{miss} = 4 \cdot \frac{4n_{po}}{l_{RDC}} = \frac{16n_{po}}{l_{RDC}}$$

RDRAM cache misses visible when reading the weight matrix. One of these misses is a dirty miss. During the local computation the processors only read from the main memory, but during the scaling they wirte one vector. The dirty cache miss occurs, when a weight vector is loaded into a cache block which is also used for the result vector.

Inner loop In this loop, the processors lood two vectors from the D-cache and eight from the main memory. The memory accesses bypass the D-cache. Two of the loads from the

main memory are overlapped, so only half the access time is visible. Parallel to these two loads, the nodes process the data of the previous memory accesses. The processors schedule one multiply-accumulate instruction and a D-cache access or some scalar operations for the loop overhead. The speed of the processing nodes would allow to start a weight access every four cycles, but unfortunately the main memory is slower and so we get the following run time formula for one iteration of the inner loop, when assuming RDRAM cache hits:

$$T_{inner} = 4 \cdot \max(8, r_h) = 4r_h.$$

In the case of a cache miss r_h has to be replaced by $r_{m,/d}$. These are the access times for a 32-element vector as shown in table 4.5. The last iteration is slightly faster because the last two arithmetic vector operations are not overlapped with memory accesses. This iteration can be executed in $3 \cdot \max(8, r_h) + 7 = 3r_h + 7$ cycles.

Total run time In the j-loop, the processors start the inner loop and execute it $n_{po}/64$ times, reduce four vectors and store the result in the D-cache. Starting the inner loop takes $\max(10, r_h) = r_h$ cycles or $\max(10, r_{m,/d}) = r_{m,/d}$ cycles in the case of an RDRAM cache miss.

The four vector reductions are done in five steps, each with two instructions per vector and four instructions overhead. The reduction is executed in

$$T_{red} = 5 \cdot (4 \cdot 2 + 4) = 60$$

cycles. Further eight instructions are necessary for storing the results and for the loop overhead.

During the local computation, the processors execute the j-loop eight times and also execute two instructions for the loop overhead. Consequently, one pass of the local computation takes time:

$$\begin{aligned} T_{lc} &= 8\left(\frac{n_{po}}{64}T_{inner} - r_h + 7 + T_{red} + 8\right) + 2 + n_{miss}(r_{m,/d} - r_h) + r_{m,d} - r_{m,/d} \\ &= \frac{n_{po}}{8}T_{inner} - 8r_h + 8T_{red} + 122 + +n_{miss}(r_{m,/d} - r_h) + r_{m,d} - r_{m,/d} \\ &= \frac{n_{po}}{2}r_h - 8r_h + n_{miss}(r_{m,/d} - r_h) + r_{m,d} - r_{m,/d} + 602. \end{aligned}$$

Global sum

Per iteration of the outer loop the processors compute the global sum of a 32-element vector. The data are spread over p_i nodes. It takes $\log p_i$ multiple transfers to compute the global sum. The size of p_i depends on the size of the machine and the data distribution.

In the 128-node machine p_i equals 8. Every fourth node of a ring is involved in the same global sum computation. The processors execute three multiple transfers over the distances 4, 8 and 16.

In the 1024-node machine p_i equals 16. Every second node of a ring is involved in the same global sum computation. The processors execute four multiple transfers over the distances 2, 4, 8 and 16.

Sending message handlers also compute the direction of the transfer. We assumed in chapter 4.2.2 that this additional overhead requires 20 cycles. The following formula describes the run time of one global sum computation. The CPU is only busy for $40 \log p_i$ cycles, the rest of the time is spent in network interfaces and the interconnection network.

$$T_{gsum} = \begin{cases} \sum_{i=2}^{4} T_{MT}(2^i) + 20 \log p_i = 28 m_{length} + 56 + 59 \log p_i & ; 128 \text{ nodes} \\ \sum_{i=1}^{4} T_{MT}(2^i) + 20 \log p_i = 30 m_{length} + 60 + 59 \log p_i & ; 1024 \text{ nodes} \end{cases}$$

Scaling the error vector

In this step, the processors scale the error of the hidden layer by the hidden vector and store the final result. Loading 32 elements of the hidden vector from the D-cache takes four cycles. Afterwards, the processors execute three arithmetic vector operations and one memory access. Only two of the vector operations are data independent. The memory access is always a clean RDRAM cache miss because in the preceding computation the weight matrix swaps the error vector out of the cache. Consequently, the processors can execute step 7 for a 32-element vector in $4 + 2 \cdot 4 + 1 + w_{m,/d} = 13 + w_{m,/d}$ cycles.

Interleaving the three main routines

The interleaving in the error backpropagation code is similar to the scheme used in the steps 2 and 3 (chapter 4.2.4 and table 4.1), but now we have scaling instead of sigmoid computation. There is only one global sum overlapped with the local computation, so that the handlers interrupt the local computation only $\log p_i$ times.

There also occurs some overhead. The processors handle three threads per iteration of the outer loop and one procedure call. The thread handling takes T_{thr} cycles and the procedure handeling T_{proc} cycles. The procedure has 6 parameters, four are passed in scalar registers. This time is already included in T_{proc}. The two remaining parameters are passed in the main memory. The processors load them after entering the procedure. The first access is an RDRAM cache miss, the second a hit. While waiting for the memory data, the nodes process some CPU intern computations. The following formula describes the run time of the error backpropagation:

$$\begin{aligned} T_{step6,7} &= T_{proc} + \max(11, r_{m,/d}(4B)) + \max(7, r_h(4B)) + \frac{n_{ph}}{32}(13 + w_{m,/d}) \\ &\quad + T_{lc} + \left(\frac{n_{ph}}{32} - 1\right) \cdot \max(T_{lc} + 40 \log p_i, T_{gsum}) + T_{gsum} + 3\frac{n_{ph}}{32} T_{thr}. \end{aligned}$$

Penalty of I-cache misses

One I-cache line holds 32 instructions. The local computation fits in four lines, the scaling code in one line and the two message handlers in three lines. The sending handler is longer than the receiving handler because it also contains code for computing the direction of the transfer. At worst, these three codes use the same cache lines. That causes several misses. A miss stalls the processor up to 44 cycles and can also influence later data accesses.

Amount of I-cache misses

In the first iteration, the processors load the whole code of the local computation, that causes four I-cache misses. In later iterations, the global sum message handlers interrupt the local computation. Both handlers together have three blocks of code; that means three misses per two interrupts. The scaling causes one additional miss per iteration. At worst, the local code has to be restored after each interrupt. That doubles the misses caused by message handlers and scaling. The last iteration is different because the message handlers and the scaling code do not share the cache with the local computation. There is no restoring and the handlers are resident in the cache during the whole global sum computation. The following formula describes the amounts of I-cache misses N_{Im} during the error backpropagation:

$$N_{Im} = 4 + \left(\frac{n_{ph}}{32} - 1\right) \cdot 2 \cdot (3 \log p_i + 1) + 3 + 1 = \left(\frac{n_{ph}}{32} - 1\right) \cdot (6 \log p_i + 2) + 8.$$

Influence on data accesses I-cache misses have no impact on data accesses of the global sum or the scaling code because the processors access no memory data during global sum and the only cache I-cache miss in the scaling code occurs during the first instruction fetch.

In the local computation, the I-cache misses can collide with memory loads accessing the weight matrix. Each I-cache miss swaps one fourth of a cache line. The processors can load these data back into the RDRAM cache with two overlapped vector loads. At worst, each I-cache miss extends the local computation by $r_{m,/d} - r_h$ cycles.

The number of swapped blocks can be derivated from N_{Im}. Misses occuring in the messages handlers and in the scaling code have no direct impact, only restoring the code of the local computation causes significant misses. For these reason, the amount of cache blocks containing weight vectors and being swapped by I-cache misses is not higher than:

$$N_{Im}(lc) = 3 + \left(\frac{n_{ph}}{32} - 1\right) \cdot (3 \log p_i + 1).$$

6.2.3 Weight updates

The processors update both weight matices. The matrices have different dimensions but they are distributed in a similar manner and so we can analyze both steps together. The processors load the weight matrix rowwise in blocks of 64 elements. Accessing 64 elements at once guaranties that all four memory ports are involved. The matrix elements are two bytes wide and the memory ports are interleaved every 32 byte. The processors update four rows at the same time, because four elements of the data vector **d** are stored in one memory word and the processors can load them with one cache access.

Table 6.3 shows the pseudocode for the weight update. **d** represents the data vectors **h**, and **i**, **e** represents the error vetcors **eo**, and **eh** and M represents the two weight matices. Vectors **d** and **e** have n_{p1} and n_{p2} elements. R and r refer to the symbolic use of the vector and scalar registers. The inmost of the three loops is completely unrolled in the assembler code (appendix A.2.2). We refer to the two remaining loops as inner and outer loop.

```
for i=0, np1 - 1, 4;
{ rd[0:3] = d[i:i+3];
  for j=0, np2-1, 64
  { Rt[0:63] = e[j, j+63];
    for k=0, 3
      M[i+k][j:j+63] -= Rd[0:63] * rd[k];
  }
}
```

Table 6.3: Pseudocode of the weight update.

Run time of the inner loop

First, the processors load eight weight vectors and compute their new values and then they store the result vectors. Two of these loads or stores are overlapped, so only half of the memory access time is visible. The processors execute three vector operations, a cache load, a multiplication and an addition, and some scalar operations per main memory vector load. The next load of a weight vector could be scheduled every 7 cycles, but the main memory accesses might slow it down. At best, a vector load from the main memory takes $r_h = 14cc$ and at worst, it takes twice as long. Except for the first and last iteration, a load and a store are overlapped twice per loop. Stores are always hits and they are faster than loads. There is no more than one instruction scheduled per vector store. For the standard case with RDRAM cache hit we therefore get the following run time formula, the last iteration is slightly faster.

$$T_{hit} = \begin{cases} 7\max(7, r_h/2) + 7\max(2, w_h/2) & \text{; standard iteration} \\ +2\max(7, r_h/2, w_h/2) & \\ = 9r_h/2 + 7w_h/2 & \\ 7\max(7, r_h/2) + 8\max(2, w_h/2) & \text{; last iteration} \\ +\max(7, r_h/2, w_h/2) & \\ = 4r_h + 4w_h & (r_h = 14cc; \quad w_h = 10cc) \end{cases}$$

When loading the weight vectors there always occurs two RDRAM cache misses in a row. They slow the execution down by $2(\max(7, r_{m,d}/2) - \max(7, r_h/2)) = r_{m,d} - r_h$ cycles.

Amount of RDRAM cache misses

In our benchmark the local matices have no more than 4096 elements per row, so four rows occupy at most 32KB cache space. The RDRAM has two cache lines with $16l_{RDC} \geq 16KB$ each. The processors therefore can keep four matrix rows in the RDRAM cache and update the matrix in $2B \cdot n_{p1} \cdot n_{p2}/(16l_{RDC})$ blocks. There occur 16 dirty misses per cache line, but only four are visible. Two ports are involved per load and two loads are overlapped. The RDRAM cache misses add up to $n_{miss} = n_{p1}n_{p2}/(2l_{RDC})$.

Total run time of the weight update

The procedure overhead takes time T_{proc}. The processors transfer four parameters in scalar registers and one in the main memory. Loading the fifth parameter requires $r_{m,d}(4B) = 20cc$ or $21cc$. During this time the processors schedule the 11 cycles startup of the outer loop. Two further cycles are required for leaving the procedure. The outer loop is executed $n_{p1}/4$ times. The loop overhead is completely hidden behind vector operations of the inner loop. The inner loop is executed $n_{p2}/64$ times per iteration of the outer loop. The following formula describes the run time of the outer loop, ignoring cache misses:

$$\frac{n_{p2}}{64}T_{hit} + \frac{w_h}{2} - \frac{r_h}{2}.$$

The whole weight update can be done in time:

$$\begin{aligned} T_{update}(n_{p1}, n_{p2}) &= T_{proc} + max(11, r_{m,d}(4B)) + 2 + n_{miss}(r_{m,d} - r_h) \\ &\quad + \frac{n_{p1}}{4}\left(\frac{n_{p2}}{64}T_{hit} + \frac{w_h}{2} - \frac{r_h}{2}\right). \end{aligned}$$

Penalty of I-cache misses

Loading an I-cache block slows the first instruction of that block down by 44 cycles. The code of the weight update fits in 4 blocks, and so there occur only 4 I-cache misses during the weight update. The first I-cache miss occurs immediately after entering the procedure. At that point, the RDRAM caches do not yet contain relevant data. The other three misses might influence data accesses. Loading an I-cache block destroyes one fourth of the RDRAM cache line. At worst, the processors still need these data. That causes an additional penalty of $3(r_{m,d} - r_h)$. For this procedure, the penalty of I-cache misses consequently adds up to

$$P_{enalty} = 4 \cdot 44cc + 3(r_{m,d} - r_h).$$

The processors execute the same code during the setps 8 and 9, only the parameters are different. After step 8, the code is present in the I-cache, and so the penalty for I-cache misses occurs only during step 8.

Preloading the error vector

We use the same code for both weight update. This code should be equally optimized for both steps. The simplest way to reach that is to achieve the same memory situation for both steps. In step 8 the data and error vectors stand in the D-cache and the weight matrix is accessed in the main memory. In step 9 the situation is a bit different. Before entering the weight update procedure in step 9, the processors preload the error vector **eh**, because it would otherwise only stand in the main memory.

The error vectors are two bytes wide. One vector load therefore occupies two memory ports. Two loads are overlapped. Some of these loads are dirty RDRAM cache misses but most of them are hits. The whole vector requires $2n_{ph}/(16l_{RDC})$ cache lines and there are four misses visible per cache line. The procedure overhead requires T_{proc}, all parameters

are passed in scalar registers. The remainig overhead is executed in six cycles. That adds up to the following total run time, ignoring the impact of I-cache misses:

$$T_{preload} = T_{proc} + 6 + \frac{n_{ph}}{64} r_h + 4 + \frac{n_{ph}}{2 l_{RDC}} (r_{m,d} - r_h).$$

In this procedure, I-cache misses have virtually no impact on the run time. The code is very short and therefore fits into one cache block. The only I-cache miss occurs after entering the procedure. This slows the execution down by 44 cycles but has now impact on data accesses.

This code might swap parts of the weight update code out of the I-cache. That would add another $44 + r_{m,d} - r_h$ cycles to the penalty.

Run time of the steps 8 and 9

The following formula describe the run time of the steps 8 and 9.

$$\begin{aligned} T_{step8} &= T_{update}(n_{ph}, n_{po}) \\ T_{step9} &= T_{update}(n_{pi}, n_{ph}) + T_{preload} \end{aligned}$$

6.3 Cache problems

Sofar, we assumed that the local vectors have 1K elements at most, and so most of the vectors can be stored in the D-cache. But two of the systems mentioned in table 3.1 have much larger vectors. In the following, we analyze how we can rearrange the code for these cases.

6.3.1 Step 1

When reading inputs, the run time is mostly spent in transfering data from the Hydrants to the Torrents. The Torrents receive new data every $m_{length} + 1$ cycle. The transfer keeps the CPU busy for 10 cycles, then the processors store the data in the D-cache and the main memory. The result patterns are only stored in the main memory.

The cache accesses will be obmitted, when the input vector is too large for the D-cache. This could only speed up the last iteration of the I/O loop, but in the last iteration the processors read a part of the result vector. The run time of step 1 remains the same even with D-cache problems.

6.3.2 Steps 2 and 3

The analysis for these two steps is similar to the recall case, but now the limit is 2KB data per vector instead of 1KB because only one pattern is processed at a time. In our benchmark, the input and output vector never exceed this limit, so only the hidden vector causes problems. In the recall analysis we also presented optimizations which keep the performance impact of too large input or output vecors under 5%. The optimizations can also be used for training if necessary.

Problems in step 2

We already analyzed this case in chapter 4.4.2. The hidden vector when exceeding 2KB is directly accessed from main memory. This only changes the sigmoid code. The hidden vector is stored in main memory after the sigmoid computation. During training the processors have only two instead of four vectors and so the sigmoid run time is increased by $w_{m,d} + 4$ cycles. That is 27 cycles for the small memory system and 26 cycles for the large memory system.

Problems in setp 3

We also analyzed this case in chapter 4.4.2 but for the input vector in step 2. Adapting the solution and the analysis yields the following: The hidden vector is loaded in 2KB blocks into the D-cache. The blocktransfer occurs $n_{ph}/2K$ times and one transfer takes time

$$4r_{m,d} + \left(\frac{2048}{4 \cdot 32} - 4\right) r_h = 4r_{m,d} + 12r_h.$$

6.3.3 Step 5

The error computation basically deals with three vector, the ouptut, the result and the output error vector. The result vector is always accessed in the main memory and the two other vectors are originally stored in the D-cache. The output layer does not exceed 1024 elements in our benchmark, and so the vectors still fit in the D-cache. But the output vector might not stand in the cache when entering this step. In that case the data are preloaded in a blocktransfer, similar to the one in the previous section. The output vector can be loaded in time:

$$4r_{m,d} + \left(\frac{n_{po}}{4 \cdot 32} - 4\right) r_h.$$

This case occurs when the hidden layer has more than 2K elements and therefore the output vector is directly stored in the main memory during step 3.

6.3.4 Steps 6 and 7

During the error backpropagation, the processors access three vectors. The error vector of the output layer and the hidden vector are originally in the D-cache. The error vector of the hidden layer is always considered to be in the main memory. The size of the error vector **eo** is at most 1K elements, that is 2KB of storage space. This vector therefore always fits in the D-cache, but a large hidden vector ($n_{ph} > 2K$) causes problems.

After step 5 the large hidden vector is only present in the main memory, and there is also not enough space in the D-cache during the steps 6 and 7. So the vector is directly accessed in the main memory. That changes the run time of the scaling. Loading the vector is a dirty RDRAM cache miss and therefore increases the run time by $r_{m,d} - 4$ cycles per pass.

6.3.5 Steps 8 and 9

The processors update the weight matices during these two steps. Both codes deal with a weight matrix, an error vector and a data vector. Originally the vectors are stored in the D-cache and the matrix in the main memory. The processors access the error vector much more often than the data vector. For that reason, the processors keep as many elements of the error vector in the D-cache as possible and access the data vector in the main memory.

Problems in step 8

The error vector is already in the D-cache after the error backpropagation, so only the hidden vector can cause problems. A hidden vector with more than 2K data has to be accessed in the main memory. The processors read four elements of the vector per iteration of the outer loop. That takes now $r_{m,d}(4B)$ cycles instead of 4 cycles and can not be overlapped with other memory accesses. These loads slow the weight update of step 8 down by $n_{ph}/4 \cdot r_{m,d}(4B)$ cycles.

Problems in step 9

The input vector is small enough to fit in the D-cache, but we still want to use the fast cache for the error vector. The error vector requires the whole D-cache, when it has more than 1K elements. In that case, the processors access the input vector in the main memory and not in the D-cache. That increases the run time by $n_{pi}/4 \cdot r_{m,d}(4B)$ cycles, analogue to the previous case.

The D-cache is too small to store the local part of the error vector at a whole when the vector has more than 2K elements. The processors therefore only load chunks of 4KB in the cache. Table 6.4 shows the new pseudocode. That is now like executing the weight update of $n_{ph}/2K$ matrices of the size $n_{pi} \times 2K$ and preloading the corresponding error vectos. The following formula describes the run time of step 9, assuming a very large hidden layer ($n_{ph} > 2048$):

$$\begin{aligned} T_{step9} &= \frac{n_{ph}}{2048}\left(T_{update}(n_{pi}, 2048) + 10 + \frac{2048}{64} r_h + r_{m,d} - r_h\right) + T_{proc} \\ &= \frac{n_{ph}}{2048}\left(T_{update}(n_{pi}, 2048) + 10 + 31\, r_h + r_{m,d}\right) + T_{proc}. \end{aligned}$$

```
for l=0, nph-1, 2048
{ for i=0, npi - 1, 4;
   { rd[0:3] = d[i:i+3];
      for j=0, 2047, 64
      { Rt[0:63] = e[l+j, l+j+63];
         for k=0, 3
            M[i+k][l+j:l+j+63] -= Rd[0:63] * rd[k];
      }
   }
}
```

Table 6.4: Pseudocode of the weight update in step 9 with $n_{ph} > 2048$.

Chapter 7

Performance Results of Kernel 2

Based on the run time formula from the previous chapter, we now determine the performance of the CNS when training large neural networks. We then analyze how the parallelization scheme, the data access pattern, the code optimizations and some other parameters influence these performance numbers.

7.1 Performance results

In our analyses, we vary the number of processing nodes, the RDRAM type, the dimension of the neural network and the size of the lookup table and also consider best and worst cases for the I-cache misses and for the access patterns of the lookup table. The performance of the CNS actually also depends on the refresh time of the main memory, but that impact is less than 0.1%, even in the worst case, and so we obmit that parameter. After dealing with the general case, we do a detailed analysis of the model case, defined in chapter 5.1.1.

7.1.1 General results

The absolute performance of the CNS running kernel 2 is measured in *connection updates per second* [CUPS]. During one pass, each node updates $n_{con} = (n_{pi} + n_{po})n_{ph}$ connections. The amount of arithmetical operations for this pass equals $n_{ma} = (2n_{pi} + 3n_{po})n_{ph} + 0.5n_{po}$ multiply-accumulate instructions per node. Each node has a peak performance of 1 GMAS (Giga multiply-accumulate per second). The peak performance for kernel 2 is therefore n_{con}/n_{ma} GCUPS. Comparing the absolute performance with the peak performance yields the relative performance measured in %.

We now analyze how the processor number and the size of the neural network influence the performance. In order to do so, we keep the other parameters constant and assume worst cases for I-cache misses and table lookup times. Table 7.1 summarizes the performance for training under these conditions.

The 128-node machine executes about 22 GCUPS, and the 1024-node machine executes between 165 and 174 GCUPS. These large training problems scale optimal on the CNS. With fixed local problem size, the performance scales by 7.9 (99%) when going from 128 to 1024 nodes. The relative performance lies for both machines between 37 and 40%. So

		Performance				
Nodes	Connections	Abs. [GCUPS]		Rel. [%]		Peak
128	192M	21.5	22.0	39	40	55
	786M	21.5	22.3	38	41	
	1536M	20.3	21.7	38	40	
1024	786M	164.1	166.1	37	38	439
	1536M	169.9	173.5	39	40	
Memory chip size		4.5 Mb	18 Mb	4.5 Mb	18 Mb	

Table 7.1: Rounded performance of the CNS running kernel 2, assuming worst case for I-cache misses and table lookup

training is less efficient than recall (57 to 86%), but the realtive performance varies much less. We analyze these aspects in section 7.3.

With fixed global problem size, the relative speedup even lies between 7.4 and 8.4. The small machine has to handle more cache problems, that causes superlinear speedup. The performance impact of the RDRAM version increases with the local problem size, but it only varies between 1 and 7%. So the RDRAM version only has a minor influence on the run time.

The number of processors, the problem size and the RDRAM version have a less than 8% impact on the training time. For a detailed analysis of further parameters, we therefore only consider the model case, defined in chapter 5.1.1.

7.1.2 Detailed results under model conditions

Under model conditions, the CNS-1 has 128 processing nodes with 4.5 Mb memory chips. The neural network with a 8K input layer, a 16K hidden layer and a 4K output layer comprises 192M connections. The corresponding values for the local problem size and the partitioning of the machine are summarized in table 3.1.

Impact of table lookups and I-cache misses

In the previous analysis, we assumed worst case models for table lookups and I-cache misses. We now analyze their performance impacts. Two parameters influence the table lookup time: the size of the table and the access patterns. Table 7.2 shows that these parameters have virtually no impact on the performance, at least under model condition. The difference between the best and the worst case is less than 2%, even considering both parameters together. Table lookups are only a small fraction of the forward code, and do not occur in the backward pass.

The model of I-cache misses is more important than the model of table lookups, but it is still a minor aspect (tables 7.2, 7.3). The run time difference between the best and the worst case is 3 to 12%, increases with the number of processors and decreases with larger problem size, because this means more locality and less misses.

I-cache Misses	Sigmoid Lookup Table				
	Best Access Pattern		Worst Access Pattern		Maximal
	16 KB	32 KB	16 KB	32 KB	Deviation
Best Case	21.8	21.7	21.6	21.6	0.9 %
Worst Case	20.3	20.2	20.1	20.1	1.0 %
Max. Deviation	7.4 %	7.4 %	7.5 %	7.5 %	

Table 7.2: Performance for training under model conditions [10^9 CUPS]

Processors		128			1024	
Connections		192M	768M	1536M	768M	1536M
Memory	4.5Mb	7.5	3.8	3.2	11.8	9.7
Chips	18Mb	7.3	3.7	3.1	11.5	9.4

Table 7.3: Maximal performance impact of I-cache misses depending on machine size and memory version [%]

Changing the models of table lookups and of I-cache misses slightly influence the performance of the CNS. For further analysis we therefore assume worst case for both parameters.

Run time distribution

Table 7.4 summarizes run time values for all steps of the training code. These values show that under model conditions the error computation, the I/O and the overhead for handling threads and procedures have virtually no impact on the run time. The CNS spents over 98% of the time for feedforward, error backpropagation and weight update. These three codes contain matrix vector operations. The CNS is twice as fast on the small matrix B than on the large matrix A ($n_{pi} = 2n_{po}$). The visible administrativ overhead in these procedures is rather small.

Feedforward, backpropagation and weight update require nearly the same amount of arithmetic operations, but nevertheless, the weight update takes 56% longer. The main differences between these codes are the amount of memory accesses and the number of transfer operations. The weight update can be parallelized without any transfer overhead, but it reads and writes the matrices. That recudes the amount of arithmetic operations per main memory access and makes the vectorization less efficient. The following analysis shows how the run time of these three procedures comes together.

Run time impact of the transfer

Feedforward and error backpropagation have the same structur. The execution is split in several rounds to enable the overlapping of computation and transfer. In the feedforward routines, the transfer requires less than 50% of the time necessary for arithmetic opera-

Steps		Vector	Run time	
No.	Function	Dimension	Absolute [cc]	Relative [%]
1	I/O	$n_{pi} + n_{po}$	2013	0.17
2	Feedforward (A)	$n_{pi} \times n_{ph}$	263712	23
3	Feedforward (B)	$n_{ph} \times n_{po}$	127444	11
5	Error Computation	n_{po}	270	0.02
6, 7	Backpropagation	$n_{ph} \times n_{po}$	139376	12
8	Weight Update (B)	$n_{ph} \times n_{po}$	203542	18
9	Weight Update (A)	$n_{pi} \times n_{ph}$	407860	35
	Overhead		11970	1
	Total	$(n_{pi} + n_{po}) \times n_{ph}$	1156187	100

Table 7.4: Run time of kernel 2 under model conditions, assuming worst case for I-cache misses and table lookup. The relative times are rounded.

Step	Matrix or Multiply	Sigmoid Vector Scaling	Global Sum			
			CPU	Network		
				Visible	Hidden	
2	16015	668	240	7898		per Round
3	15156	668	320	1820		
5	4075	36	120	3949		
2	256240	10688	3840	0	126368	total
3	121248	5344	2560	1820	12740	
5	130408	1152	3840	3949	122419	

Table 7.5: Run time distribution for feedforward and error backpropagation under model conditions ([cc]).

tions, but in the backpropagation algorithm both times are nearly the same (table 7.5). Nevertheless, the network part of the transfer can almost completely be hidden in all three procedures. The local computation is therefore responsible for over 90% of the run time.

Run time impact of memory accesses

The arithmetic vector operations in feedforward, error backpropagation and weight update are mostly multiply accumulate instructions. Each node can start eight multiply accumulates per cycle. That determines the minimal run time of the three procedures. Comparing the minimal run times with the real run times shows the amount of overhead largely due to problems with the memory latency and bandwidth (table 7.6).

In the training step, the nodes spent about 50% of the matrix multiply time waiting for data from the main memory. That even gets worse for the weight update, because there are twice as much matrix accesses per arithmetic operation. And so 68% of the time is wasted

Step	Matrix	Required Cycles for Arithmetic	Run time [cc]	Overhead [%]
Feedforward	A	131072	256240	48.8
	B	65536	121248	46.0
Backpropagation	B	65536	130408	49.8
Weight Update	A	131072	407860	67.8
	B	65536	203542	67.8

Table 7.6: Comparision of the rel run time and the minimally required amount of CPU cycles for the most timeconsuming steps of the training code.

in the main memory. A faster memory system could reduce the training time by a factor of up to 2.1, but then the transfer gets partially visible during backpropagation.

7.2 Influence of the CNS specific optimizations

We used several optimizations to speed up the assembler code of kernel 2, but only three of them yield a considerable performance win. Most of the optimizations were also applied to the recall code. After describing the different optimizations, we analyze their performance impact.

7.2.1 Description of the optimizations

Overlapping computation and transfer

The CNS uses active messages which principally enables the nodes to overlap computation and transfer. There are some specific optimizations required to make optimal use of this feature. Without active messages, the best thing a programmer can do, is to send as many data at once as possible. But that is different for the CNS. Pure computation codes and codes with transfer are interleaved. The computation is split in smaller blocks, and the transfer is started as soon as the data of a vector register are valid. That allows us to hide almost all the transfer. Using the optimization for non-active-message machines would make all transfer time visible. This optimization affects the run time of feedforward and error backpropagation.

Larger matrix stripes

There are several aspects relevant for finding an efficient access pattern of the weight matrices. At least four rows fit in an RDRAM Cache line. Accessing the matrix in those chunks reduces the amount of RDRAM cache misses. Even more important than the cache misses is the fact that the four RDRAM banks are interleaved every 32B. Accesses of 64 weights guaranty that all four memory ports are busy. This enables the RAMBUS controller to overlap two memory accesses; and consequently, the CPU sees only half the memory access

time. 64 is twice the standard vector length, but using stripes of 32 elements halves the effective memory bandwidth for feedforward, error backpropagation and weight update. The memory accesses would take twice as long without this optimization.

Overlapping memory accesses and computation

The memory access time is quite large when compared to the CPU cycle time, even with optimal memory bandwidth. Unrolling the inner loops and software pipelining the next makes it possible to overlap the memory access from the current vector operation with the computation of previous vector operations. With this optimization, the memory access time is partially hidden behind computation, otherwise both times would add.

Combining steps 6 and 7

We combined the execution of the steps 6 and 7. In step 6, the processors multiply a matrix with a vector. The result is 4 bytes wide. In the following step, they scale these data and store the 2 byte results in the error vector **eh**. When combining both stpes, the intermediate results can be kept in vector registers and only the final result is stored in the main memory. That reduces memory traffic. Accessing the intermediate results in the main memory increases the run time by

$$\frac{n_{ph}}{32}(w_{m,d} + r_h) + \max\left(4, \frac{n_{ph}}{4\, L_{RDC}}\right)(r_{m,/d} - r_h)$$

cycles. But this optimization has virtually no performance impact, it changes the run time of the error backpropagation by about 1%.

Storing the matrix twice

The backpropagation training requires matrix vector multiplications with both, standard and transposed matrices. On some machines ([Mül93]) the matrices are stored twice, in row-major and in column-major order, to get equal speed for both types of multiplications. This optimization is not adequat for the CNS at all.

On the CNS, the multiplication with the transposed matrix is only 7.6% slower than the multiplication with the standard matrix (table 7.6), but the weight update would be 50% slower. Almost the whole update time is spent loading and storing the matrices. Storing them twice therefore increases the update time by about 50%. The weight updates consume 53% of the total run time, the transposed matrix multiplication only 12% and so this optimization would cause a 24% performance loss.

7.2.2 Performance impact of the optimizations

We need some information about the run time of the main procedures to determine the performance impact of the optimizations described above. Table 7.7 summarizes these information.

The first optimization overlaps computation and transfer. The hidden transfer would be visible when obmitting this optimization. That would add $23 \cdot 0.48 + 11 \cdot 0.1 + 12 \cdot 0.09 =$

Procedure	Matrix	Run Time [%]	Percentage of the Procedure Run Time		
			Hidden Transfer	Computation	Memory Accesses
Feedforward	A	23	48	51	97
	B	11	10	54	95
Backpropagation	B	12	9	50	94
Weight Update	A	35	0	32	99
	B	18	0	32	99

Table 7.7: Run time characteristics of the main procedures. All numbers are rounded. The run time of the procedures is given as percentage of the total run time.

13.22 % to the total run time. The performance impact is rather small because the most timeconsuming procedures require no transfer at all.

Accessing the matrix in stripes of 64, doubles the effective memory bandwidth for matrix accesses. The penalty for loading the matrices in stripes of 32 (standard vector length) is $23 \cdot 0.97 + 11 \cdot 0.95 + 12 \cdot 0.94 + 53 \cdot 0.99 = 96.51$ % of the original run time. This almost doubles the training time.

Unrolling and software pipelining of loops makes it possible to overlap computation and memory accesses. Without these optimizations both times would add, and the run time would be $23 \cdot 0.51 + 11 \cdot 0.54 + 12 \cdot 0.50 + 53 \cdot 0.32 = 40.63$ % larger. The penalty for using none of these optimizations is fairly large, the training would take 2.5 times longer.

Storing the matrix twice and using all the other optimizations decreases the performance by 24%, as shown above, but without the other optimizations the training gets even three times slower. Storing the matrix twice increases the amount of memory transfers, but bad access patterns even make it worse. We therefore have to add $2 \cdot 24\%$ to the previous penalty of 150%.

7.3 Comparision of recall and training

Three differences get obvious when comparing the performance results of recall and training.

7.3.1 Scaling of the performance

When going from 128 to 1024 nodes, the recall performance scales by 94% (68%) and the training performance scales by 99% (105%), assuming fixed local (global) problem size. Training scales better than recall, because the weight update is parallelized without any transfer. This is the most timeconsuming procedure of the training, but does not appear in the recall code.

Nodes	Connections	Absolute Performance [CPS / CUPS]		Relative Performance [% / %]	
128	192M	4.9	5.0	2.1	2.1
	786M	4.8	5.0	2.1	2.1
	1536M	4.7	4.9	2.0	2.1
1024	786M	3.6	3.6	1.5	1.5
	1536M	4.7	4.7	2.0	2.0

Table 7.8: Comparision of the absolute and relative performance of recall and training under model conditions. The values are rounded.

7.3.2 Limiting memory bandwidth

In the recall procedures, less than 8% of the run time is spent waiting for data from main memory, but in the training procedures up to 68% of the time is wasted in the memory system. Consequently, the memory bandwidth strongly limints the training performance but not the recall performance. This is due to the fact, that there are more arithmetic operations scheduled per memory access during recall than during training. This difference has mainly two reasons:

First, the processors compute an outer product and read and write a matrix when updating the weights. In all other procedures with matrix operations they only read the matrix but do not write it. The weight update, a major part of the training, therefore has twice as much memory accesses per arithmetic vector operation than other procedures.

Second, during recall we pass two patterns at a time trough the neural network. That doubles the amount of arithmetic operations per memory access. This optimization is not feasible for per-pattern training. Without this optimization, the memory bandwith would also limit the recall performance, the run time would be increased by a factor 1.8.

7.3.3 Efficiency of the assembler code

Table 7.8, comparing the absolute and relativ performance of recall and training, shows that recall is much more efficient than training. The absolute performance of recall is 3.6 to 5.0 times higher than the performance of training, but training requires 2.3 times more arithmetic operations and 3.3 times more memory accesses. The ratio of the relative performances takes the different amount of arithmetical operation into account and is therefore much lower. It lies between 1.5 and 2.1, depending on the local problem size. The smaller efficiency of the training mainly depends on the decreased amount of arithmetic operations per memory access. We analyzed this aspect already in the previous paragraph. One reason was, that we pass two patterns in parallel through the recall code. Without this optimization, recall would only be 3 to 3.7 times faster than training.

Chapter 8

Conclusion

8.1 General results

Our analyses show that a 128-node CNS could achieve up to 111 GCPS and 22 GCUPS on large dense networks. When executing the recall code, the machine reaches about 87% of the peak performance, that is a very good sustained performance. The codes scale well on the CNS, especially the training code. Increasing the number of nodes from 128 to 1024 gives a speedup of 7.9, that is almost ideal.

The main bottleneck seems to be the memory system. The memory access time limits the performance achieved on the training code. The training time could be 2.1 times smaller with a faster memory system. This problem could be solved for the recall code, because it contains much more parallelism than the per-pattern training. It is also possible to overcome this limitation in the training by using a per-set training regime.

It is very important to reduce the amount of I-cache misses caused by handler interrupts in order to get a high recall performance. At worst, these misses can increase the run time by 30%, but a careful code mapping can avoid that.

It also turned out, that the CNS can compute a matrix vector multiplication using a transposed matrix almost as fast as a standard matrix vector multiplication. The run time difference is less than 8%.

8.2 Implication for the CNS design

A faster memory system would speed up the training up to 100%.

Besides that, there are two open problems in the design, one is the implementation of a multiply-accumulate instruction and the other is the location of scalars for vector operations. Obmitting a special multiply-accumulate instruction would have no visible impact on the run time of the two benchmark kernels. Anyway, there might still exist some important codes, which would profit from this instruction.

The scalars for vector operations can be stored in the scalar unit or in a special registerfile in the vector unit. This change also has no impact on the benchmark run time, because there are enough idle cycles in the assembler codes.

8.3 Optimization strategies

Starting from optimized parallel algorithms, we were able to reduce the recall time by a factor of 4 and the training time by a factor of 2.5 with some CNS specific optimizations.

In the current design, the main memory is rather slow compared with the CPU cycle time. It is therefore very important to use the whole memory bandwidth. That requires more sophisticated access patterns. The memory has four ports which can work in parallel. Overlapping two or four accesses, depending on the width of the vector elements, reduces the effective access time.

The RDRAM chips have two cache lines which are almost twice as fast as the RDRAM itself. It therefore helps to access as many data per cache line as possible. This might require re-arrangements of memory accesses.

Software pipelining and loop unrolling are two other optimizations. They reduce the amount of loop overhead and combine the operations of several iterations. That makes it possible to overlap the memory accesses of later vector operations with the computation of current operations. This helps hiding the memory access times.

The CNS uses active messages which principally enables the nodes to overlap computation and transfer. Without active messages, the best thing a programmer can do, is to send as many data at once as possible. But that is different for the CNS. Pure computation codes and codes with transfer are interleaved. The computation is split in smaller blocks, and the transfer is started as soon as the data of a vector register are valid. That allows us to hide almost all the transfer. Otherwise the whole transfer times are visible.

Appendix A

Assembler Code

We write the assembler code in a register transfer language and not in CNS-assembler [Asa93, AC93], but we only use CNS instructions aside from two exceptions. First, we use a multiply-accumulate instruction for the vector unit, and second, we assume that the scalars for vector operations are stored in a special register file in the vector unit. That requires instructions to move these scalars around.

`$0`, ... `$31` indicate scalar registers and `$v0`, ... `$v31` indicate vector registers. The scalars used in vector operations are accessed via `$vs0`, `$vs1`, ... The registers `$0`, `$v0` and `$vs0` hold the constant zero; writes to these registers are ignored.

We choose this language for three reasons. First, we made some assumptions which do not match the current design. Second, the CNS-assembler language was still changing at the time we started writing the codes. Using a register transfer language decoupled us from most of the changes. Finally, it should be possible to understand the code even without knowing anything about the CNS-assembler language.

Some routines require several parallel threads. We do not know yet, how to indicate that in assembler code. The codes of those threads are therefore arranged sequentially and comments indicate the overlapping.

A.1 Kernel 1: Recall

```
// Each iteration of the loop performs a 4x64 tile of the weight
// matrix array. Two activations are consumed at once.
// Scalar unit picks up four bytes using a word load. It then uses
// shift-left, shift-right arithmetic operations to sign-extend each
// byte in turn to pass it as an operand to the multiply-accumulate
// instruction.
//
// C function interface is
//
//  forward(short* weights, int rows, int cols, char* inputs1, int* inputs2,
//             int* outputs1, int* outputs2, int scale, int* sigmoid)
//
//          weights points to matrix with #columns=#neurons, #rows=#inputs
//
// Mips calling convention passes args as follows:
//
//  $4 = weights (in routine, points to top of current column of 32)
//  $5 = rows
```

```
//  $6 = cols
//  $7 = inputs1
//  $sp(inputs2_offset) = inputs2 (passed in memory)
//  $sp(outputs1_offset) = outputs1 (passed in memory)
//  $sp(outputs2_offset) = outputs2 (passed in memory)
//  $sp(scale_offset) = scale (passed in memory)
//  $sp(sigmoid_offset) = sigmoid (passed in memory)
//
// Local temporaries (32 scalar registers assumed)
//
// $8 is outputs1 pointer
// $9 is weight matrix stride (2bytes * #neurons)
// $10 is pointer just past end of inputs1
// $11 holds last four bytes of input1
// $12 holds current extracted input byte
// $13 holds current input1 pointer
// $14 holds current weight pointer
// $15 holds pointer just past end of first row of weights
//
// $16 is pointer start of inputs2
// $17 is outputs2 pointer
// $19 holds last four bytes of input2
// $18 holds current input2 pointer
// $20 holds current weight2 pointer
// $21 holds extracted activation of input2
// $22 holds scale value
// $23 is sigmoid pointer
//
// Vector temporaries
//
// $v1-v3 hold last weight vectors from memory, 32 elements each
// $v5-v8 hold both sum vectors, 32 elements each
//
start:
    $8 = m[sp+outputs1_offset].w;      // Get output1 pointer
    $9 = $6 << 1;                      // Calculate weight matrix stride
    $10 = 32;                          // Set vector length
    $17 = m[sp+outputs2_offset].w;     // Get output2 pointer
    $vlr = $10;
    $10 = $7 + $5;                     // Calculate input end pointer
    $15 = $4 + $9;                     // Calculate weight end pointer
    $16 = m[sp+inputs2_offset].w;      // Get input2 pointer
    $14 = $4;                          // Copy pointer to weights
    $20 = $4 + 64;                     // pointer to the second w-vector
    $13 = $7;                          // Copy pointer to input1
    $22 = m[sp+scale_offset].w;        // Get scale value
    $23 = m[sp+sigmoid_offset].w;      // Get sigmoid pointer
//
outerloop:                             // matrix vector multiplication
    $v1 = m[$14+=$9].h;                // Get first vector (0:31) of weights
    $11 = m[$13].w;                    // Get four inputs1, one byte each
    $v5 = v0;                          // Clear sum1, elements 0:31
    $v6 = v0;                          // Clear sum1, elements 32:63
    $12 = $11 shl 24;                  // Extract first byte of input1
    $12 = $12 shr 24;                  // ...
    $18 = $16;                         // Copy pointer to input2
    $v2 = m[$20+=$9].h;                // Get first vector (32:63) of weights
    $19 = m[$18].w;                    // Get four inputs2, one byte each
    $v7 = v0;                          // Clear sum2, elements 0:31
    $v8 = v0;                          // Clear sum2, elements 31:63
    $21 = $19 shl 24;                  // Extract first byte of input2
```

```
    $21 = $21 shr 24;                   // ...
    $vs1 = $12;                         // Move to a scalarReg
innerloop:
    $v3 = m[$14+=$9].h;                 // first byte of 2 activations
    $vs2 = $21;
    $v5 += $v1 * $vs1;
    nop
    $12 = $11 shl 16;
    $12 = $12 shr 24;
    $v7 += $v1 * $vs2;
    nop
    $v4 = m[$20+=$9].h;                 // Second half of the weight vector
    nop
    $v5 += $v2 * $vs1;
    nop
    $21 = $19 shl 16;
    $21 = $21 shr 24;
    $v7 += $v2 * $vs2;
    $vs1 = $12;

    $v1 = m[$14+=$9].h;                 // second byte of 2 activations
    $vs2 = $21;
    $v6 += $v3 * $vs1;
    nop
    $12 = $11 shl 8;
    $12 = $12 shr 24;
    $v8 += $v3 * $vs2;
    $18 +=4;
    $v2 = m[$20+=$9].h;                 // Second half of the weight vector
    $13 +=4;
    $v6 += $v4 * $vs1;
    if ($13 = $10) endloop              // Exit interleaved loop
    $21 = $19 shl 8;
    $21 = $21 shr 24;
    $v8 += $v4 * $vs2;
    $vs1 = $12;

    $v3 = m[$14+=$9].h;                 // third byte of 2 activations
    $vs2 = $21;
    $v5 += $v1 * $vs1;
    nop
    $12 = $11 shr 24;
    $11 = m[$13].b;
    $v7 += $v1 * $vs2;
    nop
    $v4 = m[$20+=$9].h;                 // Second half of the weight vector
    nop
    $v5 += $v2 * $vs1;
    nop
    $21 = $19 shr 24;
    $19 = m[18].b;
    $v7 += $v2 * $vs2;
    $vs1 = $12;

    $v1 = m[$14+=$9].h;                 // fourth byte of 2 activations
    $vs2 = $21;
    $v6 += $v3 * $vs1;
    nop
    $12 = $11 shl 24;
    $12 = $12 shr 24;
    $v8 += $v3 * $vs2;
```

```
    nop
    $v2 = m[$20+=$9].h;                // Second half of the weight vector
    nop
    $v6 += $v4 * $vs1;
    nop
    $21 = $19 shl 24;
    $21 = $21 shr 24;
    $v8 += $v4 * $vs2;
    goto innerloop
    $vs1 = $12;

endloop
    $21 = $21 shr 24;
    $v8 += $v4 * $vs2;
    $vs1 = $12;
    $v3 = m[$14+=$9].h;                // third byte of 2 activations
    $vs2 = $21;
    $v5 += $v1 * $vs1;
    nop
    $12 = $11 shr 24;
    nop
    $v7 += $v1 * $vs2;
    nop
    $v4 = m[$20+=$9].h;                // Second half of the weight vector
    nop
    $v5 += $v2 * $vs1;
    nop
    $21 = $19 shr 24;
    nop
    $v7 += $v2 * $vs2;
    $vs1 = $12;
    nop                                // fourth byte of 2 activations
    $vs2 = $21;
    $v6 += $v3 * $vs1;
    $v9 = $v5;                         // save partial result1(0:31)
    $4 += 128;                         // next two columns of weights
    nop
    $v8 += $v3 * $vs2;
    $v11 = $v7;                        // save partial result2(0:31)
    nop
    nop
    $v6 += $v4 * $vs1;
    $14 = $4;                          // weight pointer update
    $20 = $4 + 64;                     // ..
    nop
    $v8 += $v4 * $vs2;
    $v10 = $v6;                        // save partial result1(32:63)
    nop
    nop
    nop
    $v12 = $v8;                        // save partial result2(32:63)
//
// global sum runs in parallel to the next iteration of the multiply code
globalsum:
    gsum($v9,$v10,$v11,$v12, st)       // gsum mostly in handlers

sigmoid:
    $v0 = m[$23+low_bound].w           // preload sigmoid table into
       // RDRAM cache
    $v0 = m[$23+0].w                   // only for 32K table, 4.5Mb chip
    $v9 = $v0 addap0 $v9;              // clip the address
```

```
    ; stall till RDRAM ports are idle again
    $v9 = m[$23+$v9].b;                 // Index into sigmoid table.
    $v10 = $v0 addap0 $v10;             // clip the address
    ; stall till RDRAM ports are idle again
    $v10 = m[$23+$v10].b;               // Index into sigmoid table.
    m[$8+=32].w = $v9;                  // result1 (0:31) in cache
    $v11 = $v0 addap0 $v11;             // clip the address
    ; stall till RDRAM ports are idle again
    $v11 = m[$23+$v11].b;               // Index into sigmoid table.
    m[$8+=32].w = $v10;                 // result1 (32:63) in cache
    $v12 = $v0 addap0 $v12;             // clip the address
    ; stall till RDRAM ports are idle again
    $v12 = m[$23+$v12].b;               // Index into sigmoid table.
    m[$17+=32].w = $v11;                // result2 (0:31) in cache
    ; stall for 1 cycle
    $13 = $7;                           // copy pointer to input1
    if ($4 != $15) outerloop
    m[17+=128].w = $v12;                // result2 (32:63) in cache

return:
    goto $31;
    nop;
```

A.2 Kernel 2: Training

A.2.1 Forward pass

This is almost the same code than the recall code (section A.1) but now for only one pattern at a time. In the original code, the registers `$16` - `$19`, `$21`, `$v7` and `$v8` store information of the second pattern. Instructions containing one of these registers are now obmitted.

```
// Each iteration of the loop performs a 4x64 tile of the weight
// matrix array. Scalar unit picks up four bytes using a word load.
// It then uses shift-left, shift-right arithmetic operations to
// sign-extend each byte in turn to pass it as an operand to the
// multiply-accumulate instruction.
//
// C function interface is
//
//    forward(short* weights, int rows, int cols, char* inputs1,
//            int* outputs1, int scale, int* sigmoid)
//
//    weights points to matrix with #columns=#neurons, #rows=#inputs
//
// Mips calling convention passes args as follows:
//
//  $4 = weights (in routine, points to top of current column of 32)
//  $5 = rows
//  $6 = cols
//  $7 = inputs1
//  $sp(outputs1_offset) = outputs1 (passed in memory)
//  $sp(scale_offset) = scale (passed in memory)
//  $sp(sigmoid_offset) = sigmoid (passed in memory)
//
// Local temporaries (32 scalar registers assumed)
//
// $8 is outputs1 pointer
// $9 is weight matrix stride (2bytes * #neurons)
// $10 is pointer just past end of inputs1
```

```
// $11 holds last four bytes of input1
// $12 holds current extracted input byte
// $13 holds current input1 pointer
// $14 holds current weight pointer
// $15 holds pointer just past end of first row of weights
// $20 holds current weight2 pointer
// $22 holds scale value
// $23 is sigmoid pointer
//
// Vector temporaries
//
// $v1-v3 hold last weight vectors from memory, 32 elements each
// $v5-v6 hold sum vectors, 32 elements each
//

start:
    $8 = m[sp+outputs1_offset].w;  // Get output1 pointer
    $9 = $6 << 1;                  // Calculate weight matrix stride
    $10 = 32;                      // Set vector length
    $vlr = $10;
    $10 = $7 + $5;                 // Calculate input end pointer
    $15 = $4 + $9;                 // Calculate weight end pointer
    $14 = $4;                      // Copy pointer to weights
    $20 = $4 + 64;                 //   pointer to the second w-vector
    $13 = $7;                      // Copy pointer to input1
    $22 = m[sp+scale_offset].w;    // Get scale value
    $23 = m[sp+sigmoid_offset].w;  // Get sigmoid pointer
//
outerloop:                     // matrix vector multiplication
    $v1 = m[$14+=$9].h;        // Get first vector (0:31) of weights
    $11 = m[$13].w;            // Get four inputs1, one byte each
    $v5 = v0;                  // Clear sum1, elements 0:31
    $v6 = v0;                  // Clear sum1, elements 32:63
    $12 = $11 shl 24;          // Extract first byte of input1
    $12 = $12 shr 24;          // ...
    $v2 = m[$20+=$9].h;        // Get first vector (32:63) of weights
    $vs1 = $12;                // Move to a scalarReg of the vectorunit
     ; stall
innerloop:
    $v3 = m[$14+=$9].h;        // first byte of 2 activations
    $v5 += $v1 * $vs1;
    $12 = $11 shl 16;
    $12 = $12 shr 24;
     ; stall
    $v4 = m[$20+=$9].h;        // Second half of the weight vector
    $v5 += $v2 * $vs1;
    $vs1 = $12;
     ; stall
    $v1 = m[$14+=$9].h;        // second byte of 2 activations
    $v6 += $v3 * $vs1;
    $12 = $11 shl 8;
    $12 = $12 shr 24;
     ; stall
    $v2 = m[$20+=$9].h;        // Second half of the weight vector
    $13 +=4;
    $v6 += $v4 * $vs1;
    if ($13 = $10) endloop      // Exit interleaved loop
    $vs1 = $12;
     ; stall
    $v3 = m[$14+=$9].h;        // third byte of 2 activations
    $v5 += $v1 * $vs1;
```

```
    $12 = $11 shr 24;
    $11 = m[$13].b;
     ; stall
    $v4 = m[$20+=$9].h;          // Second half of the weight vector
    $v5 += $v2 * $vs1;
    $vs1 = $12;
     ; stall
    $v1 = m[$14+=$9].h;          // fourth byte of 2 activations
    $v6 += $v3 * $vs1;
    $12 = $11 shl 24;
    $12 = $12 shr 24;
     ; stall
    $v2 = m[$20+=$9].h;          // Second half of the weight vector
    $v6 += $v4 * $vs1;
    goto innerloop
    $vs1 = $12;
endloop
    $vs1 = $12;
     ; stall
    $v3 = m[$14+=$9].h;          // third byte of 2 activations
    $v5 += $v1 * $vs1;
    $12 = $11 shr 24;
     ; stall
    $v4 = m[$20+=$9].h;          // Second half of the weight vector
    $v5 += $v2 * $vs1;
    $vs1 = $12;
     ; stall
    $v6 += $v3 * $vs1;
    $v9 = $v5;                   // save partial result1(0:31)
    $4 += 128;                   // next two columns of weights
     ; stall
    $v6 += $v4 * $vs1;
    $14 = $4;                    // weight pointer update
    $20 = $4 + 64;               // ..
    $v10 = $v6;                  // save partial result1(32:63)
//
//
//
globalsum:
    gsum($v9,$v10, st)            // gsum mostly in handlers

sigmoid:
    $v0 = m[$23+low_bound].w     // preload sigmoid table into RDRAM cache
    $v0 = m[$23+0].w             // only for 32K table and 4.5Mb chip
    $v9 = $v0 addap0 $v9;        // clip the address
    ; stall till RDRAM ports are idle again
    $v9 = m[$23+$v9].b;          // Index into sigmoid table
    $v10 = $v0 addap0 $v10;      // clip the address
    ; stall till RDRAM ports are idle again
    $v10 = m[$23+$v10].b;        // Index into sigmoid table
    m[$8+=32] = $v9;             // result1 (0:31) in cache
    ; stall for 1 cycle
    $13 = $7;                    // copy pointer to input1
    if ($4 != $15) outerloop
    m[$8+=32] = $v10;            // result1 (32:63) in cache

return:
    goto $31;
    nop;
```

A.2.2 Backward pass

Error computation

```
// The output layer error is a 2B vector. We use the entropy error metric.
// This error is the differnce between the desired result and the outputs
// of the network; both are 1B vectors. The result is scaled by fac.
//
// C function interface is
//
//    errcalc(char* results, int vlength, char* outputs, short* errors,
//            int fac)
//
// Mips calling convention passes args as follows:
//
//  $4 = results
//  $5 = vlength
//  $6 = outputs
//  $7 = errors
//  $sp(fac_offset) = fac (passed in memory)
//
// Local temporaries
//
// $8 holds stride for results and outputs
// $9 holds stride for errors
// $10 holds pointer just passed outputs
// $11 holds the learning rate fac
//
// $v1, ..., $v4 hold results, loaded from main memory
// $v5, ..., $v8 hold outputs, loaded from the D-cache
// $v9, ..., $v12 hold errors, to be stored in the D-cache
//
start:
    $11 = m[sp+fac_offset].w;     // get learning rate
    $8 = 32;                      // stride for results and outputs
    $9 = $8 << 1;                 // stride for errors
    vlr = $8;                     // set vector length
    $10 = $6 + $5;                // set end of outputs pointer
loop:
    $v1= m[$4+=$8].b;             // load 1st vector of results
    $v5= m[$6+=$8].b;             // load 1st vector of outputs
     ; stall
    $v2 = m[$4+=$8].b;            // load 2nd vectors of results
    $v6= m[$6+=$8].b;             // load 2nd vectors of outputs
     ; stall
    $v3 = m[$4+=$8].b;            // load 3rd vectors of results
    $v7= m[$6+=$8].b;             // load 3rd vectors of outputs
     ; stall
    $v4 = m[$4+=$8].b;            // load 4th vectors of results
    $v8= m[$6+=$8].b;             // load 4th vectors of outputs
     ; stall till memory data valid
    $v1 = $v1 - $v5;              // 1st error vector
     ; stall
    $v2 = $v2 - $v6;              // 2nd error vector
    $v1 = $v1 * $11;              // scale 1st vector
     ; stall
    $v3 = $v3 - $v7;              // 3rd error vector
    $v2 = $v2 * $11;              // scale 2nd vector
    m[$7+=$9].h = $v1;            // store 1st result
     ; stall
    $v4 = $v4 - $v8;              // 4th error vector
    $v3 = $v3 * $11;              // scale 3rd vector
```

```
    m[$7+=$9].h = $v2;            // store 2nd result
     ; stall
    $v4 = $v4 * $11;              // scale 4th vector
    m[$7+=$9].h = $v3;            // store 3rd result
     ; stall
    if ($6 != $10) loop:          // computation finished?
    m[$7+=$9].h = $v4;            // stall 4th result
return:
    goto $31;
    nop
```

Error backpropagation

```
// The intermediate vector is the product of the weights and the
// errors of the outputs (outerr). Combining this vector with the
// inputs yields the error vector for the inputs (inerr). Matrix,
// outerr and inerr are 2B, inputs 1B and intermediate results 4B.
//
// C function interface is
//
//    wupdate(short* weights, int rows, int cols, char* inputs,
//            short* inerr, short* outerr)
//
//    weights points to matrix with #columns=#neurons, #rows=#inputs
//
// Mips calling convention passes args as follows:
//
//  $4 = weights (in routine, points to top of current column of 32)
//  $5 = rows
//  $6 = cols
//  $7 = inputs
//  $sp(inerr_offset) = inerr (passed in memory)
//  $sp(outerr_offset) = outerr (passed in memory)
//
// Local temporaries
//
// $8 is outerr pointer
// $9 is weight matrix stride (2bytes * #neurons)
// $10 is vector length, stride for outerr
// $13 holds current output pointer
// $14 holds current weight pointer (0:31)
// $15 holds current weight pointer (32:63)
// $16 holds pointer just passed end of inputs
// $17 holds pointer just past end of first row of weights
// $18 holds pointer to inerr
// $19 holds pointer to intermediate results
// $20 holds offset for vector reduction
// $21 holds pointer just passed end of interm. results
// $22 holds current interm. pointer
// $23 holds current column pointer ($14 = $4 + $23)
// $24 holds 4*stride, used to reset addresses for stores
//
// Vector temporaries
//
// $v1, $v2 hold 2 outerr vectors
// $v3, ..., $v6 hold 4 partial sums
// $v7, ..., $v10 hold 4 weight vectors
// $v11, $v12 hold vectors for global sum
//

start:
```

```
    $8 = m[sp+outerr_offset].w;   // Get output pointer
    $9 = $6 << 1;                 // Calculate weight matrix stride
    $24 = $6 << 3;                // Four times stride
    $10 = 1;
    vs1 = $10;                    // set vector constant vs1 = 1
    $10 = 32;                     // Set vector length
    $vlr = $10;
    $10 = $10 << 1;               // stride of outerr and weights
    $16 = $7 + $5;                // Calculate input end pointer
    $17 = $4 + $9;                // Calculate weight end pointer
    $14 = $4;                     // Init. current weight pointers
    $18 = m[sp+inerr_offset].w;   // get interr pointer
    $15 = $4 + $10;               // ..
    $23 = $0;                     // Init. current column pointer
    $21 = $19 + 128;              // Calculate end of interm. vector
    $19 = sp + interm_offset;     // get pointer to interm. results
    $13 = $8;                     // Init. current output
    $22 = $19;                    // set current pointer for interm
//
// input and output vector are already in the D-cache
outerloop:
loop:
    $v7 = m[$14+=$9].h;           // first vector of weights (0:31)
    $v1 = m[$13+=$10].h;          // first output vector (cache)
    $v3 = $v0                     // init partial sum
    $v4 = $v0                     // init partial sum
     ;stall
    $v8 = m[$15+=$9].h;           // first vector of weights (32:63)
    $v2 = m[$13+=$10].h;          // second output vector (cache)
    $v5 = $v0;                    // init partial sum
    $v6 = $v0;                    // init partial sum
     ; stall
innerloop:
    $v9 = m[$14+=$9].h;           // second vector of weights (0:31)
    $v3 += $v7 * $v1              // weight(1,1) * 1st vector
     ; stall
    $v10 = m[$15+=$9].h;          // second vector of weights (32:63)
    $v3 += $v8 * $v2              // weight(1,2) * 2nd vector
     ; stall
    $v7 = m[$14+=$9].h;           // third vector of weights (0:31)
    $v4 += $v9 * $v1              // weight(2,1) * 1st vector
    $23 = $23 + $10;              // update column pointer
     ; stall
    $v8 = m[$15+=$9].h;           // third vector of weights (32:63)
    $v4 += $v10 * $v2;            // weight(2,2) * 2nd vector
    $23 = $23 + $10;              // update column pointer
     ; stall
    $v9 = m[$14].h;               // fourth vector of weights (0:31)
    $v5 += $v7 * $v1              // weight(3,1) * 1st vector
    $14 = $4 + $23;               // set weight pointers for next pass
     ; stall
    $v10 = m[$15].h;              // fourth vector of weights (32:63)
    $v5 += $v8 * $v2              // weight(3,2) * 2nd vector
    if ($23 == $17) innerend;     // end of inner loop?
    $15 = $14 + $10;              // set weight pointers for next pass
    $v7 = m[$14+=$9].h;           // first vector of weights (0:31)
    $v6 += $v9 * $v1              // weight(4,1) * 1st vector
    $v1 = m[$13+=$10].h;          // first output vector (cache)
     ; stall
    $v8 = m[$15+=$9].h;           // first vector of weights (32:63)
    $v6 += $v10 * $v2;            // weight(4,2) * 2nd vector
```

```
    goto innerloop;
    $v2 = m[$13+=$10].h;          // second output vector (cache)
innerend:
    $4 = $4 + $24;                // update weight pointer
    $v6 += $v9 * $v1              // weight(4,1) * 1st vector
    $14 = $4;                     // set current weight vector1
    $15 = $4 + $10;               // set current weight vector2
    $23 = $0;                     // reset column pointer
    $v6 += $v10 * $v2;            // weight(4,2) * 2nd vector
    $20 = 16;                     // init reduction length
reduce:
    vlr = $20;                    // set new vector length
    $v7 = $v3[$20];               // halve 1st result vector
     ; stall
    $v8 = $v4[$20];               // halve 2nd result vector
    $v3 = $v3 + $v7;              // reduction
    $v9 = $v5[$20];               // halve 3rd result vector
    $v4 = $v4 + $v8;              // reduction
    $v10 = $v6[$20];              // halve 4th result vector
    $v5 = $v5 + $v9;              // reduction
    $20 = $20 >> 1;               // halve reduction length
    if ($20 > 0) reduce;          // last pass?
    $v6 = $v6 + $v10;             // reduction
    $20 = 4;
    m[$22+=$20].w = $v3;          // save 1st result in the cache
    m[$22+=$20].w = $v4;          // save 2nd result in the cache
    m[$22+=$20].w = $v5;          // save 3rd result in the cache
    m[$22+=$20].w = $v6;          // save 4th result in the cache
    $20 = 32;
    if($22 != $21) loop;          // 8*4 results collected?
    vlr = $20;                    // reset vector length
    $v11 = m[$19].w               // load partial sums from cache
    $22 = $19                     // reset pointer to partial sums
     ; stall
    $v12 = m[$7].b;               // load inputs from cache
//
//  global sum is a new thread a goes parallel with next iteration
    gsum($v11)
//  the following code (%) has to be executed between this and the next
//  gsum, may be done after the next matrix multiplication
    $v11 = $v11 * $v12;
    $v12 = $v12 - vs1;
    $22 = $19;                    // reset current pointer to interm.
     ; stall
    $v11 = $v11 * $v12;
     ; stall
    if ($7 != $16) outerloop;     // last round in process
    m[$18+=$10].h = $v11;         // store inerr vector in the memory
return:
    goto $31;
    nop
```

Weight update

```
// In the inner loop,  the scalar unit picks up four bytes using a
// word load. It then uses shift-left, shift-right arithmetic
// operations to sign-extend each byte in turn to pass it as an
// operand to the multiplyccumulate instruction.
// One pass through inner loop updates a 4x64 block of the matrix.
//
// C function interface is
```

```
//
//    wupdate(short* weights, int rows, int cols, char* inputs, short* outputs)
//
//         weights points to matrix with #columns=#neurons, #rows=#inputs
//
// Mips calling convention passes args as follows:
//
//  $4 = weights (in routine, points to top of current column of 32)
//  $5 = rows
//  $6 = cols
//  $7 = inputs
//  $sp(outputs_offset) = outputs (passed in memory)
//
// Local temporaries
//
// $8 is outputs pointer
// $9 is weight matrix stride (2bytes * #neurons)
// $10 is vector length, stride for outputs
// $11 holds last four bytes of input
// $12 holds current extracted input byte
// $13 holds current output pointer
// $14 holds current weight pointer (0:31)
// $15 holds current weight pointer (32:63)
// $16 holds pointer just past end of inputs
// $17 holds pointer just past end of first row of weights
// $23 holds current column pointer ($14 = $4 + $23)
// $24 holds 4*stride, used to reset addresses for stores
//
// Vector temporaries
//
// $v1, $v2 hold error term vectors
// $v3, ..., $v10 hold eight vectors of (updated) weights
// $v11, $v12 hold intermediate products
//
start:
    $8 = m[sp+outputs_offset].w;   // Get output pointer
    $9 = $6 << 1;                  // Calculate weight matrix stride
    $24 = $6 << 3;                 // Four times stride
    $10 = 32;                      // Set vector length
    $vlr = $10;                    // set vector length
    $10 = $10 << 1;                // stride for outputs and weights
    $16 = $7 + $5;                 // Calculate input end pointer
    $17 = $4 + $9;                 // Calculate weight end pointer
    $14 = $4;                      // Init. current weight pointers
    $15 = $4 + $10;                // ..
    $23 = $0;                      // Init. current column pointer
    $13 = $8;                      // Init. current output
//
// input and output vector are already in the D-cache
outerloop:
    $v3 = m[$14+=$9].h;            // first vector of weights (0:31)
    $11 = m[$7].b;                 // 4 inputs, 1 byte each (cache)
    $v1 = m[$13+=10].h;            // first output vector (cache)
    $7 = $7 + 4;
    $12 = $11 shl 24;              // extract first byte of inputs
    $12 = $12 shr 24;              // ..
    $vs1 = $12;                    // move constant to SIMD unit
    $v4 = m[$15+=$9].h;            // first vector of weights (32:63)
    $v11 = $v1 * $vs1;             // first product (0:31)
    $v2 = m[$13+=10].h;            // second output vector (cache)
    $12 = $11 shl 16;              // extract second byte of inputs
```

```
    $12 = $12 shr 24;              // ..
    $vs2 = $12;                    // move constant to SIMD unit
    $12 = $11 shl 8;               // extract third byte of inputs
    $v5 = m[$14+=$9].h;            // second vector of weights (0:31)
    $v12 = $v2 * $vs1;             // first product (32:63)
    $v3 = $v3 + $v11;
    $12 = $12 shr 24;              // extract third byte cont.
    $vs3 = $12;                    // move constant to SIMD unit
    $12 = $12 shr 24;              // extract fourth byte
    $vs4 = $12;                    // move constant to SIMD unit
innerloop:
    $v6 = m[$15+=$9].h;            // second vector of weights (32:63)
    $v11 = $v1 * $vs2;             // second product (0:31)
    $v4 = $v4 + $v12;
     ; stall
    $v7 = m[$14+=$9].h;            // third vector of weights (0:31)
    $v12 = $v2 * $vs2;             // second product (32:63)
    $v5 = $v5 + $v11;
     ; stall
    $v8 = m[$15+=$9].h;            // third vector of weights (32:63)
    $v11 = $v1 * $vs3;             // third product (0:31)
    $v6 = $v6 + $v12;
     ; stall
    $v9 = m[$14].h;                // fourth vector of weights (0:31)
    $v12 = $v2 * $vs3;             // third product (32:63)
    $v7 = $v7 + $v11;
    $14 = $4 + $23;                // reset the current weight pointer
     ; stall
    $v10 = m[$15].h;               // fourth vector of weights (32:63)
    $v11 = $v1 * $vs4;             // fourth product (0:31)
    $v8 = $v8 + $v12;
    $15 = $14 + $10;               // reset the current weight pointer
     ; stall
    m[$14+=$9].h = $v3;
    $v12 = $v2 * $vs4;             // fourth product (32:63)
    $v9 = $v9 + $v11;
     ; stall
    m[$15+=$9].h = $v4;
    $23 = $23 + $10;
    $v10 = $v10 + $v12;
     ; stall
    m[$14+=$9].h = $v5;
    $23 = $23 + $10;
     ; stall
    m[$15+=$9].h = $v6;
     ; stall
    m[$14+=$9].h = $v7;
     ; stall
    m[$15+=$9].h = $v8;
     ; stall
    if ($23 == $17) loopend;
    m[$14].h = $v9;
    $14 = $23 + $4;               // reset weight pointer1
     ; stall
    m[$15].h = $v10;;
    $15 = $14 + $10;              // reset weight pointer2
     ; stall
    $v3 = m[$14+=$9].h;           // first vector of weights (0:31)
     ; stall
    $v1 = m[$13+=10].h;           // first output vector (cache)
     ; stall
```

```
    $v4 = m[$15+=$9].h;           // first vector of weights (32:63)
    $v11 = $v1 * $vs1;            // first product (0:31)
    $v2 = m[$13+=10].h;           // second output vector (cache)
     ; stall
    $v5 = m[$14+=$9].h;           // second vector of weights (0:31)
    $v12 = $v2 * $vs1;            // first product (32:63)
    goto innerloop;
    $v3 = $v3 + $v11;
     ; stall
loopend:
    $23 = $0;                     // reset current column pointer
    $4 = $4 + $24;                // matrix offset increased by 4 rows
     ; stall
    m[$15].h = $v10;
    $14 = $4 + $23;               // set current weight pointer1
    if ($7 != $16) outerloop;     // Check for end.
    $15 = $14 + $10;              // set current weight pointer2
     ; stall
return:
    goto $31;
```

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. Iternal document, revisions 1.5/1.9.

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

[Cal93] T. Callahan. *Network Iinterface Manual.* International Computer Science Institute and UC Berkeley, 1993. Iternal document, revision 1.4.

[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.

[Gre92] M.C. Greenspon. Ring Array Processor: Programmer's guide to the RAP libraries. Technical Report TR-92-060, International Computer Science Institute, 1992.

[KSA93] V. Kumar, S. Shekhar, and M.B. Amin. A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architctures. Technical Report TR-92-54, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, 1993.

[MR88] J.L. McClelland and D.E. Rumelhart. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises.* The MIT Press, 1988.

[Mül93] U. A. Müller. *Simulation of Neural Networks on Parallel Computers*, volume 23 of *Microelectronics.* Hartung-Gorre Verlag, 1993. Reprint of the Ph.D Thesis: ETH No 10188.

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

[Tos92a] Toshiba. *Advanced Information: $2M \times 9$ RDRAM*, 1992.

[Tos92b] Toshiba. *Advanced Information: $512K \times 9$ RDRAM, TC59R0409*, 1992.