A Massively Parallel Neurocomputer

Michael Soegtrop and Henrik Klagges
IBM Research Division
Physics Group Munich
Schellingstrasse 4/III
D–8000 Muenchen 40
Federal Republic of Germany
E-mail: henrik@robots.ox.ac.uk, uh311ae@sun1.lrz-muenchen.de

1 Abstract

We have developed a SIMD massively parallel digital neural network simulator — called GeNet for Generic Network — which can evaluate large networks with a variety of learning algorithms at high speed. A medium-size installation with 256 physical nodes and 1 Gbyte of memory can sustain e.g. 1.7 giga 16bit-connection crossings/sec at network sizes of 2 layers with 64K neurons each, a fan-in of 1K and a random wired topology. The neural network core operations are supported by optimized and balanced computation and communication hardware that sustains heavily pipelined processing. In addition to an array of processing units with one global scalar (16 bit) bus, the system is equipped with a ring-shifter (32 bit) and a parallel (256×16 bit) vector bus that feeds a tree-shaped global vector accumulator. This eases backward communication and the calculation of scalar products of distributed vectors. The VLIW-architecture is highly scalable. A prototype has been cost-effectively implemented without custom VLSI chips.

2 Processing Unit Overview

The main part of GeNet's hardware is the array of processing units (called *matrix units*), which are streamlined for matrix-operations, i.e. sequences of similar parallel vector operations. Additionally, a GeNet system can be equipped with several special-purpose *vector units*, which are streamlined for sequential vector operations. However, usually vector operations are also performed by the matrix units, at a small loss of speed.

GeNets processing nodes execute even complex instructions in a single instruction cycle of 50ns (20MHz instruction frequency). This is achieved by using VLIW (very long instruction word) instructions. The matrix unit instruction word is 176 bits wide, while the vector units, control units and communication-components have instruction word widths of 32 to 200 bits. This makes it possible to use all hardware components in parallel without any restrictions due to a limited instruction word length.

2.1 Matrix Units

The key design choice of the matrix units is that they are SIMD (single-instruction/multiple-data). Therefore, to achieve efficient load balancing, at least as many simulated neurons as there are physical processing units in the system should execute the same operations synchronously, differing only in parameter values. This doesn't restrict GeNet to a single neuron type, but only to pools of homogenous types, because it can simulate significantly more neurons than it has physical nodes.

As the matrix units are SIMD, they do not contain instruction memory and program flow control units, but only an execution unit and data memory. The memory is organized into two banks of 1M 16-bit words each, thus two parallel memory accesses may be performed per instruction cycle. GeNet's speed depends very much on the programer's ability to distribute the data among the banks so that the execution unit can use this dual access feature. Then, a primitive operation with two memory operands and one register operand or an operand from the communication hardware is performed in only one instruction cycle (after pipeline stall). This requires that the address calculations are performed parallel to the operations on the data. As replicated local address generators, which can perform the necessary addressing in an array of records in a single cycle, would have been too expensive, we used a single global address generator. The disadvantage is that all matrix units receive the same addresses. This is no problem if the evaluated neural network has a regular structure, but networks with irregular structure (e.g. random wired) need local address generation. Thus we made it possible to use local data as address instead of the global address. For the most common type of local address, a pointer, this is very effective, as one memory bank indexes the other. In the rare cases that some kind of calculated local address is needed, the execution unit must be used for address calculations.

The execution unit was streamlined for the task of neural network evaluation, but is nevertheless universal. It has a single instruction-cycle $16 \times 16 \rightarrow 48$ bit multiplier/accumulator and a 16-bit ALU (Arithmetic and Logic Unit) which can perform addition, subtraction, shift and logical operations. As the accumulator has 48 bits, as many as $2^{17} = 131072$ signed products[(1)] may be accumulated without an accumulator overflow. A $32/16 \rightarrow 16$ bit division takes 24 instruction cycles, a $32/16 \rightarrow 32$ bit division 48. All operations can be performed in multiple precision with the usual decrease in operation speed (quadratic for multiplication and division, linear for other operations).

A problem with all SIMD machines is the implementation of conditional execution. As all matrix units are receiving the same instructions, it is not possible to implement conditional execution by program flow control. GeNet deals with this by providing two kinds of conditional instructions. First it is possible to conditionally disable write operations into memory, and second it is possible to conditionally select one of two data-items. In addition to these two types of conditional instructions, GeNet has a lot of instructions (e.g. maximum, minimum, absolute value, sign copy) which reduce the need for conditionals. Most of GeNet's arithmetic instructions allow for hardware overflow clipping, e.g. a division by zero returns the largest representable number. Thus an exception handler, which usually would need a lot of conditionals, is rarely used and doesn't need to be efficient. These combined features make it possible to implement many neural network algorithms without any conditionals in the inner loops at all.

The execution units rely on integer and fixed-point arithmetic and do not have direct support for floating-point operations. As many neural network algorithms imply some value range for node activations and weights, a fixed-point representation is very appropriate. Furthermore, standard (ANSI-IEEE 754) normalizing floating-point systems have the troublesome property,

(1) The product of two signed 16-bit values is a signed 31-bit number

that addition is not associative, and that

$$\frac{\partial}{\partial a_i} \sum_{i=1}^{n} a_i b_i \neq b_i$$

as with $n = 2$, $a_1 = 1$, $b_1 = 1$, $a_2 = 0.0001$, $b_2 = 0.0001$ and single precision (32-bit) floats. In this case, the sum is 1 and it remains 1 even if a_2 is changed by a factor of 5. Thus the true derivative of the sum with respect to a_2 is 0 rather than b_2. As the above equation is of some importance to many learning algorithms, it may be doubted whether (single-precision) floating-point numbers should be used in the neural network area. In contrast, the fixed-point system used by GeNet with a triple-length accumulator (48-bit) does not have these deficiencies. Another advantage of using fixed-point arithmetic is that multiple precision operations can be programmed easily and efficiently if they are needed e.g. for parameter fine tuning.

2.2 Vector Units

The vector units are not necessary, because the matrix units are universal. However, e.g during pipelined backprop-style feedforward, the matrix units can be kept busy with the input/weight vector product ($O(n)$ per neuron), while a single vector unit would execute the activation function table lookup ($O(1)$ per neuron) in parallel. This setup would not need the replication of the lookup table on every node.

The current version of GeNet does not contain vector units, but it is prepared for their installation. A vector table-lookup unit is probably released soon.

2.3 Control Units

The global control units implement all functions that are shared between multiple matrix units, vector units and the communication components. These are the generation of instruction and data addresses and the conversion of instruction addresses to control commands.

The instruction address generator contains an address counter, a return address stack, a jump target address memory and three hardware loop units. Every loop unit can implement up to 32 nested loops via its 32 loop control register sets. More than one loop unit are needed for special operations during longer loops. E.g., if there are 1000 simulated neurons and 256 matrix units, a loop over the number of neurons usually requires that every 256 cycles something special, like loading a new vector into the shifter bus, needs to be done. Depending on the state of the three loop units and on some input from the matrix units, the instruction address generator selects one out of seven possible addresses as the next instruction address. The potential address sources are the current address (one instruction loop), the successive address, a procedure return address or one of four immediate jump target addresses. After executing an instruction, the system could e.g. execute the same instruction once more, perform some special task (because a sub-loop counter reached zero), terminate the outer loop, abort the current procedure (and return due to an error) or perform some special task due to matrix unit input (like a rescaling after an overflow). All this condition checking, address calculation and selection takes only a single instruction cycle even for loops with complex break conditions. Furthermore, all this is done in parallel with the matrix unit, vector unit and communication operations. Even in complex situations an operational instruction is executed every (50ns) instruction, so program-flow instructions take no extra time.

The two data address generators (used for the two matrix unit memory banks) contain 32 base-, 32 offset-, 32 offset-increment and 32 offset-modulus registers and an immediate offset memory, all 24 bits wide. The corresponding offset and base registers may only be used together, but any modulus and increment register may be used for any offset-register. The only addressing mode is register indirect with register-offset, offset-postincrement and additional immediate offset. If not needed, the offset-register and the immediate offset are just zero. If the offset register is

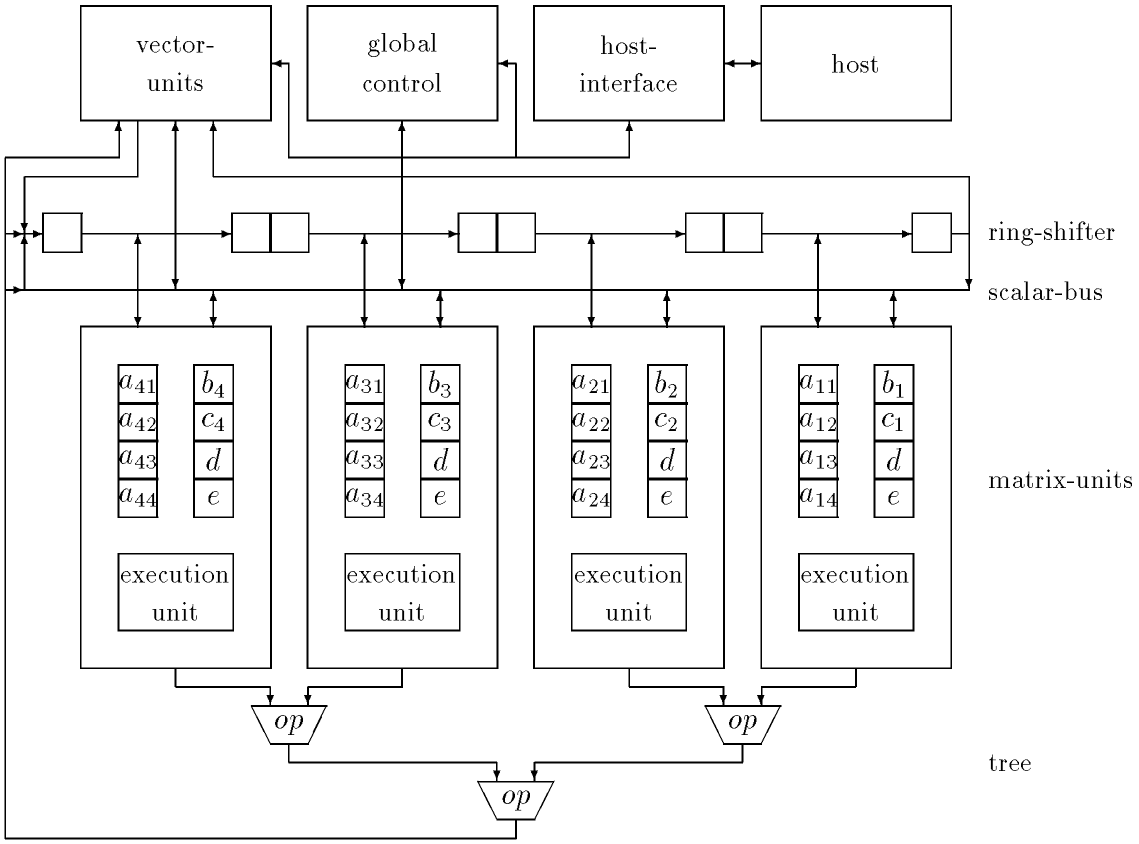

figure 1: Communication structure overview

greater than the modulus register after increment, the modulus register is subtracted from the offset register. For the implementation of FFT-algorithms with reverse carry addressing, it is possible to mirror the address.

3 GeNet's Communication Structure

GeNet's communication structure for a system with four matrix units is shown in figure 1. The amount of hardware needed for communication is proportional to the number of matrix units in the system. This non-obvious feature makes it possible to scale GeNet to very large sizes without unbalancing the design.

GeNet has three major communication components, which are described in the following sections. The *I/O bus* transmits scalar values and vectors in a (word-) serial manner. The *shifter* is used for nearest neighbour communication and to transpose vectors. The *tree* is strictly speaking an operation component rather than a communication component, but as the operation it performs is non-local, we treat it as a communication component.

3.1 I/O-bus

The 16-bit wide I/O bus connects the I/O-ports of all matrix, vector and global control units (und thus with the host). During forward propagation, it is functioning as a kind of time-multiplexed axon. It distributes data (e.g., activations) from one node to a lot of other nodes, distributed over multiple matrix units. In a bigger GeNet installation, the I/O-bus may be dynamically subdivided into multiple I/O-busses, thus partitioning the matrix units into clusters with higher internal and lower inter-cluster communication. The breakpoints are fixed and depend on the specific installation. The prototype with 256 matrix units may operate with 1 cluster of 256 units, 2 clusters of 128 units, 16 clusters of 16 units and 256 clusters of 1 unit.

The physical structure of the I/O-bus is tree-like: the lowest level of I/O-busses connects matrix units and all subsequent layers connect I/O-busses of the next higher level. As more advanced technology may be used for higher levels, there are only weak limits on the physical size of a GeNet system. The prototype's level-3 I/O-bus may connect systems with a physical distance of up to 10 meters without a reduction in system performance.

3.2 Shifter

Figure 1 shows how data is distributed over multiple matrix units. Vectors b and c are stored with one element per matrix unit. A multiple-length vector is stored with multiple elements per unit. These are called distributed, while vectors stored completely in the memory of a single matrix unit are called local or serial vectors. Matrices are stored as parallel vectors of serial vectors or vice versa. Parallel vectors are always processed in parallel, while serial vectors are processed sequentially. To convert between (i.e., transpose) parallel and serial vectors, the ring shifter is used. A parallel vector is converted to a serial vector by loading the shifter with the whole vector and then shifting it one position to the right every instruction cycle. The values shifted out are usually distributed via the I/O bus. On the other hand, a serial vector, which is arriving sequentially, is first fully shifted into the shifter from left to right, then transferred in parallel to the matrix units in a single instruction. Sometimes, it is useful to parallelize two vectors or one double-precision vector simultanously. Therefore, the shifter has double length and can operate with twice the instruction frequency. Thus two values may be shifted in or out every cycle. Furthermore, it is possible to serialize one vector while another is parallelized, because the first is shifted out to the right while the other is shifted in from the left. Thus the shifter may perform as many as four operations in parallel.
A less frequent operation is that of nearest neighbour communication. For this task, a parallel vector is loaded to the shifter, a double shift is done (ring-shift is possible), and the vector is loaded back to the matrix units. This kind of communication could be used to implement the backward-communication of backprop. As the GeNet has the tree for this task, which is more efficient, this operation is not used very frequently.

3.3 Tree

The tree is a binary tree of ALUs (Arithmetic and Logic Unit). The ALUs may perform addition, maximum, minimum and logical operations. The tree receives input from every matrix unit, and is usually used to add all the received values to calculate the scalar product of two parallel vectors. Furthermore, it may be used to calculate the maximum or the minimum element of a vector in parallel.
The ALUs in the higher levels of the tree have always one bit more precision than the ALUs of the next lower level in order to avoid overflows. The tree has two operation modes: pipelined and non-pipelined. In pipeline mode, one operation may be initiated every instruction cycle, while the fall-through time is $\log_2 N$, where N is the number of matrix units. In non-pipeline mode, the whole tree executes only one operation at a time. This decreases the fall-through time to $1 + 0.12 \log_2 N$.
The tree is a very important feature, because it makes it possible to multiply a matrix with a vector from the left and from the right or to implicitly transpose a matrix. Many standard learning algorithms, e.g. backprop and its derivatives, need this feature for the weight matrix transposition during the backpropagation phase. Furthermore, local address-generation – which is expensive on SIMD units – would be required for ring-shifter backpropagation.

4 Implementation Examples

In this section somefrequent operations are illustrated to show the cooperation of communication hardware and processing units.

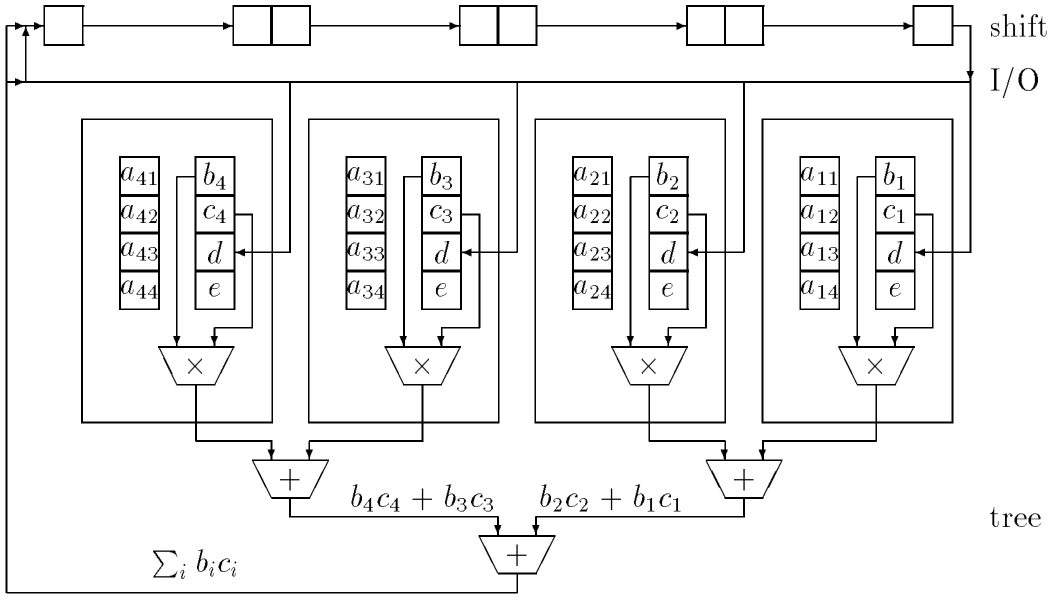

figure 2: Scalar-product of two parallel vectors

4.1 Matrix- and Vectoroperations

The first three examples show how to efficiently calculate scalar products of parallel vectors, and also that matrices can be multiplied with vectors from the left and from the right (i.e. with a serial or parallel vector) without transposing the matrix.

Figure 2 shows the calculation of the scalar product of the two parallel vectors b and c. The components of b and c are multiplied using the multiplier of the matrix units. The products are transferred to the tree, which performs an add operation. The scalar result, which is available after $\log_2 N$ or $1 + 0.12 \log_2 N$ instruction cycles, where N is the number of matrix units, is distributed via the I/O bus and stored in all matrix units in the variable d. The reason for storing a scalar multiple times is that it would be inefficient to store a scalar in one matrix unit and transfer it to all other units if it is needed. Furthermore, the memory layout of all matrix units must be equal.

Assuming 256 values and 16-bit products, the result has 24 bits. As mentioned in section 3.3, the tree-result has the necessary precision. The extension bits must be transferred in a second cycle and stored in a second memory-location. If a 32-bit product is used, first the sum of the lower product parts is calculated. The 8 extension bits of this sum may be added to the sum of the upper product halves and need not be transferred.

Figure 3 shows the first step of the calculation of the product of the matrix a with the parallel vector b. At first, the elements of the vector b are multiplied with the corresponding elements of the first row $a_{?1}$ of matrix a in the matrix units. The products are transmitted via the tree's parallel vector bus to the first level of tree-ALUs. The complete sum of all products is calculated in $log_2 N$ instruction cycles by the tree-ALUs. The result is shifted into shifter. While the tree-ALU's are performing the task to calculate the first scalar product, the matrix units are continuously calculating new partial products, thus the delay of $log_2 N$ instruction cycles is only a pipeline-stall. After n steps (see figure 4), all scalar products have been calculated. The results have been shifted sequentially into the shifter, which has transposed the serial output-vector to a parallel vector.

After $log_2 N$ pipeline delay-steps, the result-vector is complete. Instead of feeding the shifter directly with the tree-results, the results could have been fed through a sequential vector unit,

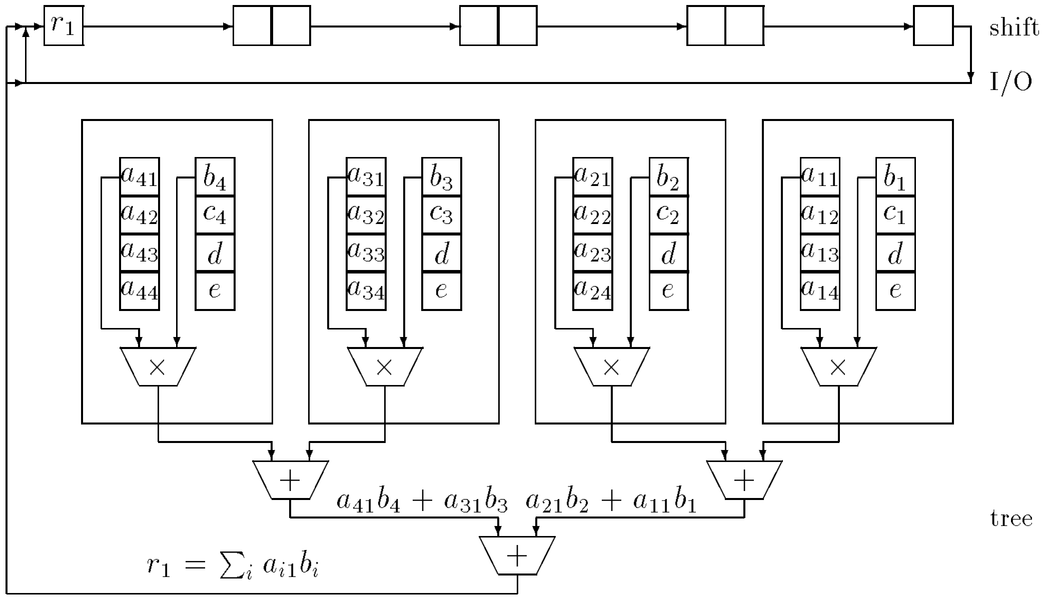

figure 3: First step of the multiplication of a matrix by a parallel vector

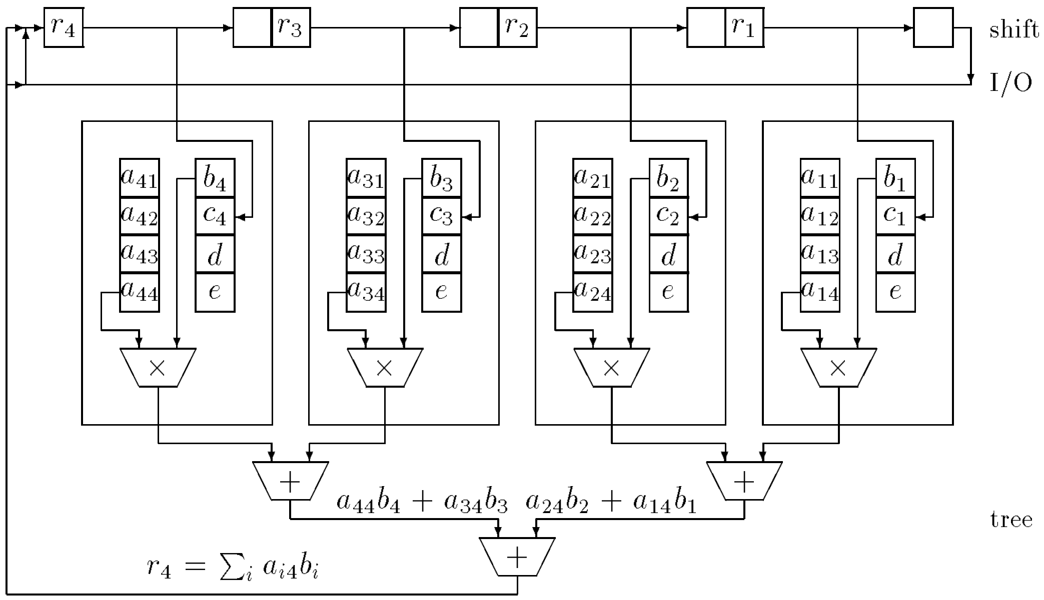

figure 4: Last step of the multiplication of a matrix by a parallel vector

which could have performed a special operation on the vector like an element-by-element table-lookup (e.g. calculation of the activation function after feed-forward). The cost for an additional vector operation is only 1 instruction cycle for the whole operation. As the shifter can take two values per instruction cycle, it is even possible to perform two vector-operations in parallel with the matrix operation (e.g. feed-forward, activation-function and activation-function-derivative calculation in parallel). After the calculation is completed, the whole result vector is transferred from the shifter-register into memory location c of the matrix units.

Figure 5 shows the first step of the calculation of the product of the matrix a with a serial vector. At first, the parallel (distributed) vector b is loaded into the shifter. The shifter transposes this vector to a serial vector. Alternative sources for serial vectors are sequential

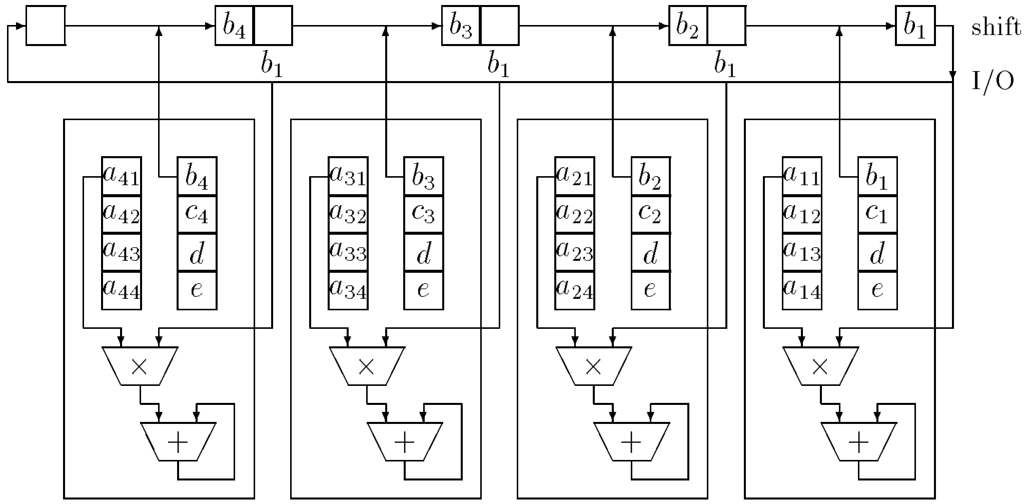

figure 5: First step of the multiplication of a matrix by a serial vector

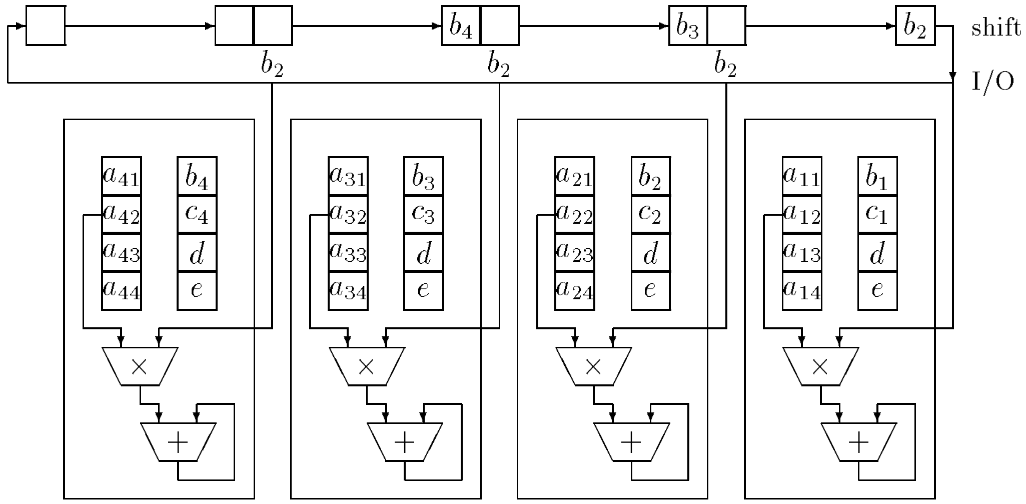

figure 6: Second step of a multiplication of a matrix by a serial vector

vector units like table-lookup units or vector memory units. The first element of the serialized vector is distributed via the I/O-bus. Every matrix unit receives the same value and multiplies it with the corresponding local matrix element. The products are stored in the matrix units accumulators.

In the next instruction cycle (see figure 6), the second input-vector element is distributed to all matrix units via the I/O-bus. The matrix units' operation is the same as in the first cycle, except that the products are added to the accumulator values. After n cycles, every matrix unit accumulator has calculated the scalar product of the serial input-vector and the local subvector of the matrix a. The result-vector is transferred from the accumulators to the matrix units' memory location c in a single instruction cycle.

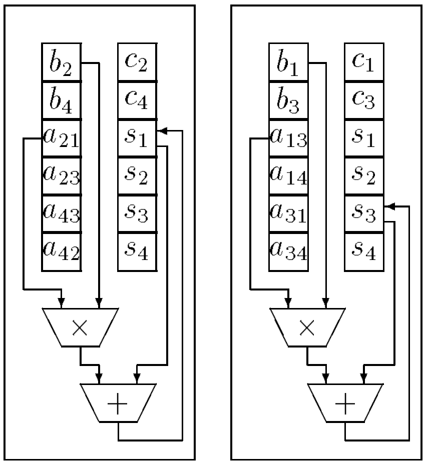

figure 7: Multiplication of a sparse matrix by a parallel vector

4.2 Sparse Matrix- and Vectoroperations

The multiplication of a sparse matrix with a vector is the most important operation in random wired neural networks. The execution of such an operation depends very much on the representation of sparse matrices. We recommend that a sparse matrix is represented as a pair of two matrices of the same size, an index and a data matrix. These matrices have the same number of columns as the corresponding sparse matrix and as many rows as the maximum number of non-zero elements in any column of the sparse matrix. Every element of the data matrix corresponds to an element of the sparse matrix with the same column-index. The row-index of the data-element in the sparse matrix is found in the same position as the data-element in the index-matrix. The multiplication of such a matrix with a vector is shown in figures 7 and 8. These operations usually require temporary storage in the matrix units and additional operation steps. If a sparse $n \times n$ with no more than r elements in any column is multiplied with a n-vector using m matrix units, $2n \cdot r/m$ storage cells are nedded in every matrix unit to store the matrix a. Usually, n storage cells are needed in every matrix unit as temporary memory. If $n < 2n \cdot r/m$ and thus $2 \cdot r > m$, the temporary memory is smaller than the matrix-memory. In a neural network, n is the number of neurons and r is the number of inputs to every neuron. Thus the above condition is met if the number of inputs is greater than half the number of processing nodes. The number of instruction cycles needed to perform the multiplication is of the same order.

In figure 7, the matrix a is a sparse 4×4 matrix with a constant number of two non-zero elements per column. As only two matrix units are used, every matrix unit holds two sub-column-vectors of the matrix a and two elements of the vectors b and c. a is stored as a data matrix and an index-matrix as described above. The index matrix is not shown. Its contents may be derived form the shown indizes. The serial vectors s, which are different in the two matrix units, hold the partial sums. They are cleared at the beginning. In the first step, every matrix unit multiplies the first element of the data matrix with the local element of b. The product is added to the partial sum s_i with the same index as the first element of the data matrix (i.e. the first element of the index vector). The same procedure is performed for every element of the data matrix subvectors. There after, the vectors s of all matrix units must be added to get the final result-vector. This is done using the ALU-tree.

In the next figure 8, the matrix a is of the same type as above. The serial vectors b are a local copy of the parallel vector b. During the accumulation, the elements of the local copy of b are indexed by the row-indices stored in the index-matrix. Apart of this, the operation is the same

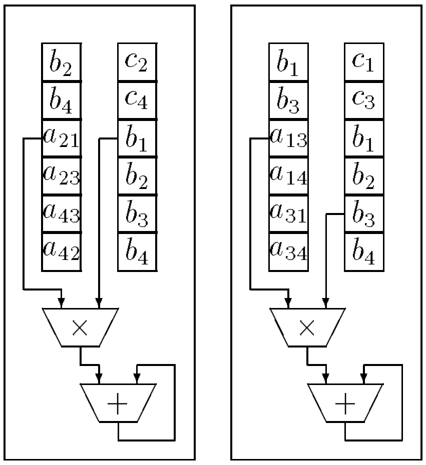

figure 8: Multiplication of a sparse matrix by a serial vector

as in figures 5 and 6.

5 Conclusion

We have shown that standard neural network operations can be implemented efficiently on a massively parallel, special purpose SIMD architecture. The design is flexible enough to support non-regular connection topologies and has benign scaling properties. An attractive option for future work would be to implement the non-DRAM part of the matrix unit architecture on a single VLSI chip.

Literature

[1] James L. McClelland and David E. Rumelhart. *Parallel Distributed Processing*, volume 2. MIT Press, Cambridge, Massachusetts, USA, 1986.