POLITECNICO DI TORINO

Corso di Laurea Magistrale in Electronic Engineering

Tesi di Laurea Magistrale

Intelligent Scheduler for Heterogenous Systems on a Chip

Relatori:
Prof. Mariagrazia Graziano
Prof. Amit Trivedi, University of Illinois at Chicago

Candidato:
Andrea Ciccardi

Dicembre 2018

Acknowledgments

Firstly, I would like to thank prof. Amit Trivedi, that gave me the opportunity of working in this challenging problem, in his lab, with special people. His determination in working in problems that will affect the future of this world was of great inspiration to me.

I am deeply thankful to Prof. Mariagrazia Graziano, her point of view always gave me a different way of thinking that helped me solving some of the challenges encountered during my thesis. Thanks to her classes, my interest in the subject grew incredibly and she gave me a precise path to follow for my future.

I would also like to thank Suraj and all the Clock Team at Nvidia. They boosted my practical knowledge and gave me inspiration during the internship.

I am extremely grateful to my family. They have always given me the full support in whatever was needed to pursue my dreams.

I would also like to extend my deepest gratitude to all my friends in Chicago, you have been like a family during this period of study abroad and you made the experience impressive, as well as to all the people that supported me from Italy.

I would like to extend my sincere thanks to Ahish, he has given me great help when I needed someone to confirm my ideas and he helped a lot in making the work less painful in that areas that are not my first expertise.

Summary

This thesis presents the design of an intelligent scheduler for heterogeneous systems. The quest for performances require the heterogeneity of the systems, but in the meantime, this may represent a problem from the power point of view. In this complex scenario, the scheduling of the tasks becomes vital. Being the scheduling a *NP*-complete problem, the core idea is to move all the complexity to an offline phase, train a modeled neural network and exploit it to supply the sub-optimal scheduling during online use. The solution proposed consists of a hardware binarized neural network that, in negligible time with respect to the running time of the tasks, is able to provide with an address to point a memory containing the sub-optimal scheduling for that combination of the inputs. Inputs to the system are the condition of the running execution unit, in particular, the number of clock cycles that each machine would take to complete each task and the communication cost between resources. In fact, this system is the final end of a more complex structure used to provide the neural network with the necessary inputs. Since the hardware is massively parallel the new scheduling can be computed in few nanoseconds. The efficiency of this work resides in the fact that, given the speed of the accelerator, this can be used both to adapt the scheduling to the running conditions and to compute the real scheduling every time, lowering the amount of work the operating system has to do. This would imply a slight modification in the way the system works normally, but in general, would provide the target computer with a lot more computation power and in the meantime lower the amount of work of the operating system or who is in charge of the scheduling.

Table of contents

List of figures

Chapter 1

Introduction and Theoretical Background

Modern-day high-performance computing (HPC) systems increasingly comprise disparate computing platforms (FFT processor, video processor, DSP processor, etc.) and a large number of general purpose cores. For many high throughput applications, HPCs need to run multiple applications in parallel, where each application is broken down to several tasks and multiple tasks are concurrently executed. However, given increasing heterogeneity of HPCs (many computing cores can run a task, but not all are always available) and complexity of workloads (increasing randomness due to memory access dependence, inter-task dependency, user and cloud-dependency, etc), scheduling of tasks becomes a challenging problem. In fact, the level of uncertainty in the execution time for every task opens to a variety of solutions that in general result in completely different performances.

Several papers [1][2][3] present and refer to works that state the importance of heterogeneity of the systems, where different cores are able to perform different applications with different performances. Briefly, the impact of heterogeneous architecture is related to both power consumption and speed of the system. In fact [1], [2], and [3] claim that from the power consumption point of view, the application should run the high speed resources only when needed, preferring the low power ones every time it is possible. Anyway, having specialized processors helps for both the aspects.

According to [4], the multiprocessor scheduling is considered an *NP*-hard problem, and the growing number of execution units even increases the complexity of the problem to a great extent and various trade-offs between speed and accuracy of the algorithm has to be found. With this approach, we try to move all of the complexity to the offline execution of the program, allowing a reliable method for the real-time

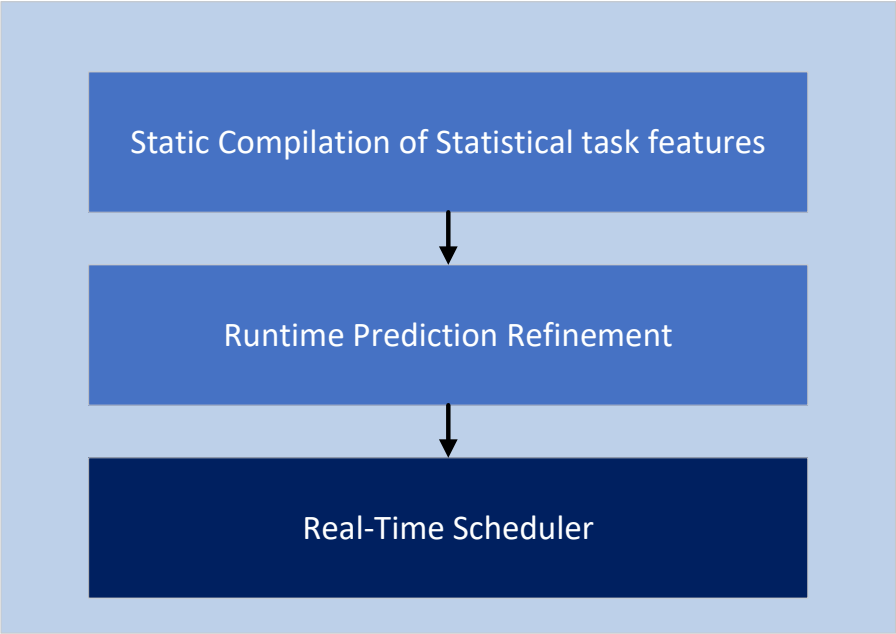


Figure 1.1: The whole system

scheduling, that requires almost no time. For this reason, we present an hardware-driven approach where a dedicated accelerator proactively updates task schedules under varying resource availability and workload to maximize the throughput of HPC.

This thesis enters inside a bigger system made up of several parts. In particular, the proposed solution aims to be the last stage of the whole system presented in 1.1. The first two steps are part of the future works.

Before getting into the details of the proposed solution, the statement of the problem, a brief state-of-the-art for the solutions, and the technical background are given.

1.1 Multiprocessor Scheduling problem

When referring to an algorithm, in most of the times the best way of representing it is by means of a graph that specifies all the dependencies between the tasks. For computer applications, this graph is the Directed Acyclic Graph (DAG), generally referred to as taskgraph. A DAG $G = (v,e)$ is a set of v nodes and e edges. Each

$e_{i \to j}$ is the edge that connects the node v_i to v_j. In this case, i is named as parent for the node j, and j is a child for node i. Every node that does not have any parent is the source or entry node, while every node that does not have any child is a sink node. Every edge is associated with a non-negative weight, that defines the amount of data that have to flow between one node to the other for the correct computation of the output. Moreover, this communication cost is not used if two consequent tasks run in the same execution unit. In this case data are already available in the resource.

Together with the DAG, the scheduling problem requires two more matrices, PC and MC:

- PC: given the number of execution units where the program to schedule can run, this matrix represents the number of clock cycles that each task would take on each execution unit. Its size is then:

$$s_{PC} = n_v \cdot n_{EU} \tag{1.1}$$

  and the matrix is like:

$$\begin{bmatrix} pc_{0,0} & \cdots & pc_{0,n_{EU}} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ pc_{n_v,0} & \cdots & pc_{n_v,n_{EU}} \end{bmatrix}$$

- MC: if two consequent tasks run in the same execution unit, then all the data are available in that execution unit without any additional time. But if those tasks run on different execution units, the data transfer time has to be considered. This matrix represents the rate in which data flow between one execution unit to the other. Its size is then defined from the number of resources available as:

$$s_{mc} = n_{EU}^2 \tag{1.2}$$

And the matrix will have the following shape:

$$\begin{bmatrix} 0 & mc_{0,1} & \cdots & mc_{0,n_{EU}} \\ mc_{1,0} & 0 & \cdots & mc_{1,n_{EU}} \\ \vdots & \ddots & \ddots & \vdots \\ mc_{n_{EU},0} & mc_{n_{EU},1} & \cdots & 0 \end{bmatrix}$$

Assuming a taskgraph with T tasks and P processors, the total time to compute the algorithm (**makespan**) is given by the finish time of the last executed task, written as:

$$T_{finish_T} = T_{start_T} + PC(T,p_T) \tag{1.3}$$

The goal of the scheduling problem is to make this time the smallest possible. p_T represents the assigned resource for the last task.

As written in [4], the start time of each task j is defined as:

$$T_{start}(n_j,p_j) = max\{T_{free}(p_j),T_{ready}(n_j,p_j)\} \tag{1.4}$$

where $T_{free}(p_j)$ is the time when the chosen resource is ready to execute a new task, that means when the last task allocated to that machine finishes the execution. $T_{ready}(n_j,p_j)$ is instead the time when the machine p_j is ready to execute the task n_j. This means that this time is equal to:

$$T_{ready}(n_j,p_j) = max_{n_k \in parents(n_j)}\{T_{finish}(n_k) + \frac{w_{e_{i\to j}}}{MC(p_k,p_j)}\} \tag{1.5}$$

where the first term of the *max* argument is the finish time for the parent node and the second one represents the communication cost. In particular, the communication cost is equal to the amount of data over the data transmission rate between the execution units.

This results in a recursive operation, up to the entry node, that will have $T_{start} = 0$.

As it can be seen, the complexity of problem is really high, and it increases with the number of tasks and the heterogeneity of the architecture where to run the taskgraph.

1.2 Classification of the existing solutions

Given the problem presented in the previous section, the approaches for the solution can be divided in two different big classification methods, based on the moment when the scheduling is done and the quality of the solution with respect to the time implied to find it.

In general, depending on the time when the scheduling is performed, the algorithm may be:

- *Static*: if the scheduling is computed at the compilation time (therefore, rigid and not-adaptive to dynamically varying workload and computing resources);
- *Dynamic*: if the allocation of each task to a resource is done at the run-time.

This work enters in the dynamic scheduling class, even though exploiting static scheduling algorithm. In particular, the system suggested tries to emulate the best solution computed statically, in a real-time fashion.

As remarked in [4], another classification of the scheduling algorithm relies on the *speed-quality of schedule* trade-off:

- Heuristic-based approach: the solution is not necessarily optimal, but it is always found in an efficient way. Sometimes, the result is not even acceptable;
- Stochastic-based approach: the solution is always performing good, but this is paid in time efficiency for providing the scheduling.

The solution that we are presenting enters in both the classes. In fact, it exploits learning architectures to provide real-time scheduling, and, depending on the reference algorithm used for learning, it provides better or worse results in term of makespan.

Basing on the first classification method, the approaches to run the schedule on each CPU are slightly different. In fact, considering the static scheduling, it is done at the compilation level, meaning that every time the executable has been called, it already contains the resource information for each task. In this kind of systems, according to [2], the scheduling is performed by the operating system, that creates a single running queue for each core and then moves the control to the core itself for

the real execution. Instead, an example of dynamic schedule is in [5]. Ramamritham et al. use the OS to create a global job queue and then, basing on the availability of task and resources and on the priority of each task, they update this queue assigning each task to each resource. Nevertheless, the way the scheduling is moved to the hardware is similar.

Given this, the reference method used during the thesis and the neural network model to implement the solution are briefly introduced.

1.3 HEFT

In the time, several methods have been found. The goal of this introduction is not to present all of them, but to present the most significant one, meaning the one used as a reference in most of the papers, as well as in this thesis. It is the HEFT. This is not the best solution present in the market, but it represents a good reference for all the papers related to scheduling.

Again, the goodness of the outcome for this work has to be found in the hardware accelerator developed, the new approach, and the solutions proposed, hence the relative performances will always be given with respect to the reference algorithm. This reference algorithm may influence the optimal makespan in general, but we believe that the network is able to learn the problem not depending on the reference algorithm used. This is the reason why the HEFT has been used, even though it does not represent the most efficient algorithm in term of the makespan outcome.

Heterogenous Earliest Finish Time (HEFT) is an heuristic algorithm that splits the computation of the scheduling in two processes:

- *Prioritizing the tasks*;
- *Assigning each task to an execution unit.*

The prioritization of the tasks is done in the following way:

- The node is a leaf: the rank is the mean of the execution time in all the machines, that means the average of the values of the row relative to that node in the PC matrix;

- The node is not a leaf: the rank is given as the average cost of the node summed up with the maximum over all the children of the average communication cost and the rank of the child. Written in formula, this means:

$$r(n_i) = \overline{pc}_i + \max_{c \in children} \{\overline{mc} + r(n_c)\} \tag{1.6}$$

After each node has a rank, this rank is seen as a priority and the task with the highest priority is the one scheduled first in the execution unit that gives the lowest finish time. After that, every task will be scheduled on the machine that gives the lowest finish time, considering the dependencies with all the previous tasks.

This does not always result in the optimal scheduling, but it is easy enough to use during the design for the training of this structure.

1.4 Binarized Neural Network

The last technical background needed to fully understand the proposed model is the Binarized Neural Network. This is a neural network that has both weights and activation constrained to one bit numbers, meaning that it represents the perfect fit for the hardware realization. As Courbariaux et al. in [6] claims, this network is able to learn with almost the same performances as the full precision ones. They work exactly as a standard neural network, with the difference that, during the training, the binarized activation and weights are used for computing the gradient, but a full precision weight is stored and use for upgrades. In general, the learning of the network happens when there is a change in sign of the not-binarized value, that is the one being updated every time. Both the weights and the local fields are binarized only considering the sign of the output value. Hence, they have value -1 and 1. Moreover, since each layer is going to be binarized, the output of each neuron will be one bit only.

What they do not specify is the behavior of the input layer, but this will be one of the differences that the network realized in this thesis have with the original one. A more detailed model is presented in chapter 3, when the implementation of it is explained as well.

Chapter 2

Overall system and Validation of the Idea

The aim of this project is to exploit learning architectures to solve the scheduling problem presented in the previous chapter. The main idea is to train a neural network on a finite number of reference schedules that allows the minimum possible loss in performances. The HEFT algorithm is used to find a fixed number of sub-optimal schedules for a variety of input configurations that are used for training. The neural network will provide with an ID that maps in real-time the input configuration to a stored scheduling. Hence, the overall system, shown in 2.1, is made up of two blocks:

- The neural network learning the scheduling ID;
- The memory being pointed from the neural network.

Generalizing the problem for all the applications would have meant to increase a lot the complexity of the problem, hence the schedule is performed for a single application architecture firstly.

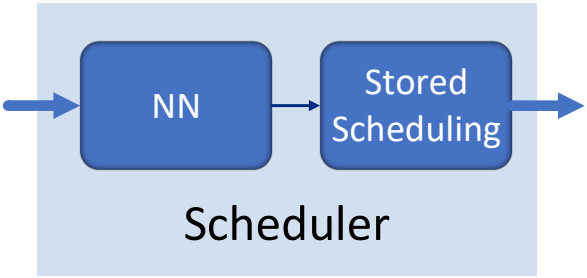


Figure 2.1: Overall system

Before starting with the explanation of the solution, an aspect must be pointed out. During the thesis, performances will be mentioned several times. There are two types of performances that matter in this case:

- *Schedule performance*: this is the performance of a schedule with respect to another. It can be seen as the difference in makespan between two schedules with the same PC and MC.

- *Hardware performance*: this is the performance that an hardware can achieve in term of operation frequency and number of clock cycles taken to provide the output, or simply, the critical path from any input to any output of the hardware structure.

Where not specified, the type of performances we are referring to can be intended from the context for most of the times. Sometimes, it has to be intended as both the performances. In any case, the reader has to think about which kind of performance is the one that a certain decision improves, to get a full understand of the problem and the relative solution.

2.1 Neural Network

Since the goal is to perform the scheduling as fast as possible, the neural network should be the simplest: a multi-perceptron neural network with input layer, one hidden layer and output layer. The number of neurons for each layer is defined by the needs that come up during the developing of the project:

- **Input layer**: to reduce the number of input neurons, it is possible to get rid of all the fixed matrix for each application. That means that both the taskgraph and the data matrices are not considered as inputs to the system. The only real inputs that change the output schedule are PC and MC matrices. Hence, their dimensions (1.1 and 1.2) fix the complexity of the input. Although, for what concerns the MC matrix, it is possible to get rid of some entries of it: if a parent and a child tasks are executed in the same execution unit, data do not have to flow. For this reason, the part of the MC matrix that is passed to

the neural network is:

$$s_{MC} = n_{EU} \cdot \ (n_{EU} - 1) \tag{2.1}$$

The number of inputs to the network is then:

$$n_I = s_{PC} + s_{MC} = n_{EU} \cdot \ (n_{tasks} + n_{EU} - 1) \tag{2.2}$$

For example, for an application with 15 tasks, to run on three different execution units, the input size will be:

$$n_{I_{fft}} = 3 \cdot \ (15 + 3 - 1) = 51$$

- **Hidden layer**: MATLAB simulations demonstrate how, for a variety of applications, $n_H = 50$ hidden neurons are sufficient.
- **Output layer**: the number of output neurons n_O is fixed and it is chosen as a power of two number. This decision came from a study performed at the beginning of the work. This is going to be explained in the next session.

2.1.1 Output format and size

When developing the project, three different ideas on the output format came out:

- *Direct Scheduling Output*: The output already represents the scheduling matrix. The size of the output in this case would be:

  $$n_O = 2 \cdot n_{tasks}$$

  Each task has the correspondent execution unit.
  However, this solution is not able to ensure correctness in task dependencies. For this reason this solution (2.2) has been discarded since the very beginning.

- *Encoded Output*: The output number represents the encoded unsigned ID for the output. This number can be directly used for addressing the memory of

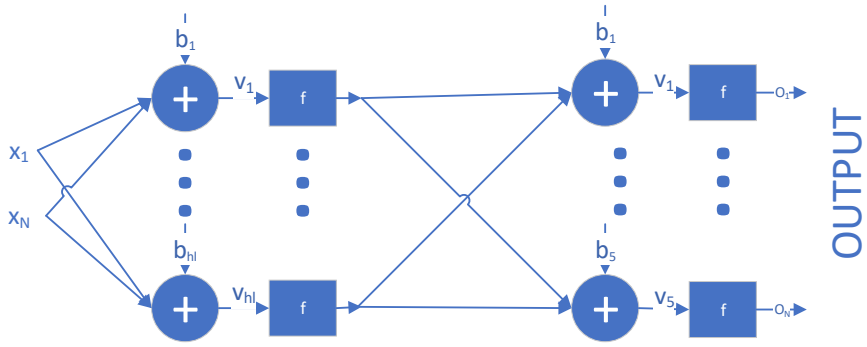


Figure 2.2: Direct scheduling output.

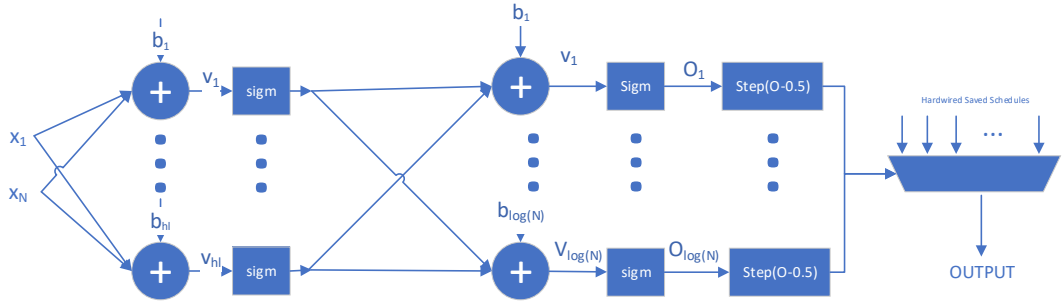


Figure 2.3: Encoded version of the output and its decoding for memory addressing.

the stored schedules. This solution can be implemented in the easiest way in hardware, resulting in a speeder system, but it would require the decoding of the output when selecting the correct schedule. A sketch of the structure is presented in 2.3.

- *One-hot Encoding of the output*: There are $n_O = 2^n$ wires, each one directly corresponds to a stored schedule. In the output of the neural network, the one with maximum value is chosen and its correspondent output wire asserted. All the other wires are zeros.

Beside the first choice, that cannot be carried out because of the incorrect nature of its output, the second and the third solutions are both feasible. Between them, the third one was chosen, because it ensured higher hit rate and lower loss in performances during the testing process. Moreover, it converges to a solution in less time during training.

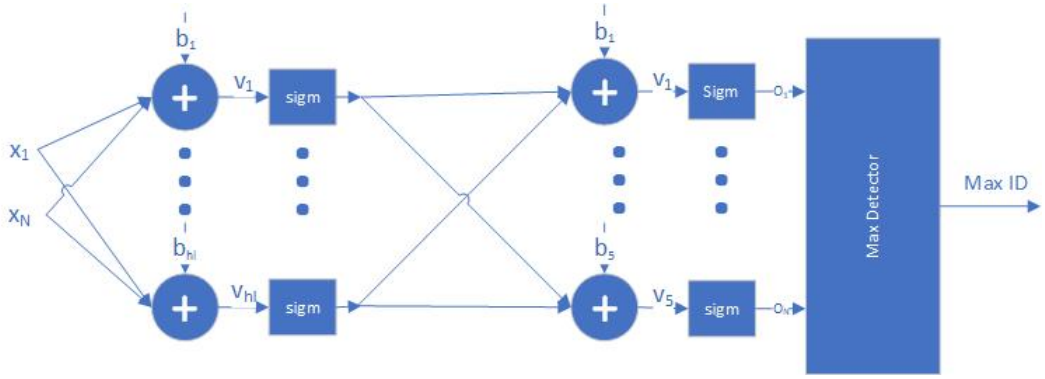


Figure 2.4: One-hot version of the output and possible selection of the schedules.

Hence, the final neural network is the one in 2.4.

The number of output wires has been evaluated looking at performances inserting an optimal block that always computes the best output between the saved ones. In this way it is possible to understand which one is the best trade-off between all the choices. From figure 2.5 and table 2.1, the following behavior has been highlighted:

Table 2.1: TRADE-OFF BETWEEN THE NUMBER OF OUPUTS.

Output wires	**4**	**8**	**16**	**32**
Performances:	-4.14%	-1.00%	-0.15%	0.50 %

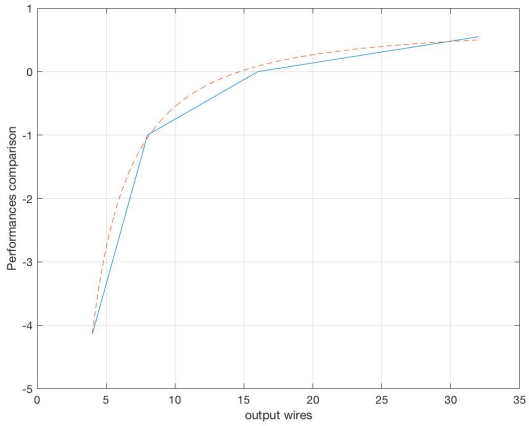


Figure 2.5: All the output values with the optimal system.

From figure 2.5 it can be seen as the change in performances is not marked

while changing from eight to higher output layers. This is expected because, increasing the number of stored scheduling, it is easier that, for each new scenario, one of the reference schedules provide a makespan that is close, or even better, than the one provided from the HEFT. Moreover, it is important to specify one more time that, being HEFT an heuristic algorithm, it does not always provide the minimum makespan. With a rigorous algorithm as reference method, the situation of a makespan shorter than the reference method could be avoided, but, since the main goal of this essay is to validate the idea, the HEFT was a good reference method to utilize, because easy and fast to apply. In a situation where the neural network training takes a lot of time, the first aim was to optimize all the other timings, exploiting the available time in the best possible way.

For this reason, since the hardware performances have to be taken into consideration, only 8 outputs are chosen as tradeoff. Increasing the number of outputs would result in an increase in performance not reflected in the hardware. In fact, the hardware would take longer and the risk to overcome the improvements with the slowness of the input becomes real. Moreover, with eight entries in the stored schedules matrix, the performances are very close to the HEFT ones.

2.1.2 The training

The training of the neural network is divided into three phases:

- Training set creation;
- Real training by means of gradient descent and back-propagation;
- Testing the network on a new test-set coherent with the training.

2.1.3 Training-set creation

The training-set should provide with the optimal schedule for each input matrices. That means that, during the training, a reference method should classify the input in such a way that each set of input has a correspondent labeled schedule associated.

To create the training set it is necessary to know the profile of the input distribution.

Profiling of the application

For the purpose of this study, a real profiling of the applications should be needed. Since the goal of this work is to validate the idea, a precise study on the profiling of the application on each execution unit available has not been carried out. This can be seen as a future work to further increase the confidence in this structure and it is a needed step before using the structure in real-world problems. In fact the profiling is one of the most important aspect to create a real training set for the neural network.

In this thesis, gaussian distribution with respect to mean value for each application has been used. The first PC matrix is the one used as reference and the variance is fixed for all the applications. The training set will hence be created accordingly to this distribution.

This is not perceived as a major problem, since it is likely that any application will have a mean time to execute and then, depending on the current workload, it takes more or less clock cycles, in a gaussian way. It represents the most intuitive way the applications should perform.

Label assignment to each input

Two additional functions are needed:

1. A first one able to evaluate the makespan of the scheduling with the reference method and the scheduling itself, for a given configuration of the inputs.

2. A second one that provides the makespan for a given input configuration and a given scheduling.

The pseudo-code of the training-set creation is the following:

```
Given a taskgraph, create a random scenario;
for i=1:1:n_samples
  Compute the optimal makespan MS_opt and optimal schedule S_opt with
     function (1);
  for j=1:1:n_stored_schedule
    Compute Makespan MS_j correspondent to j schedule and i inputs with
      function (2);
  end
```

```
  if |MS_opt - min(MS_j)|/MS_opt > tol (usually 5%)
    Add S_opt to the stored schedules;
    Assign the schedule label to the scenario;
  else
    Find the schedule label correspondent to min(MS_j);
    Assign this label to the scenario
  end
end
```

During the first phase of the research, while validating the idea, the size of the scheduling library depended on the tolerance value: a smaller tolerance made the schedule library larger and vice-versa. Later on, with an eye on the hardware realization, the size of the schedule library was fixed to a power of 2. That means that the final pseudo-code for the training set creation becomes the following.

```
Given a taskgraph, create a random scenario;
for i=1:1:n_samples
  Compute the optimal makespan MS_opt and optimal schedule S_opt with
     function (1);
  for j=1:1:n_stored_schedule
    Compute Makespan MS_j correspondent to j schedule and i inputs with
      function (2);
  end
  if size_stored_schedule < max_output_number
    if |MS_opt - min(MS_j)|/MS_opt > tol (usually 5%)
      Add S_opt to the stored schedules;
      Assign the schedule label to the scenario;
    else
      Find the schedule label correspondent to min(MS_j);
      Assign this label to the scenario
    end
  else
    Find the schedule label correspondent to min(MS_j);
    Assign this label to the scenario
  end
end
while size_stored_schedule < max_output_number
  enlarge the training set of "n_batch" more units
```

```
  Assign for each of them a new label or assign each of them to a store
      schedule (as in the previous for loop)
end
```

In the pseudocode, the numbers in parenthesis are referred to the functions used for computing the parameters.

As it can be seen from the code, the size of the training set changes depending on the number of schedule stored for a given scenario. Two possibilities happen:

- If the size of the library has reached the maximum number allowed, that particular set of input is linked with the label providing the fastest scheduling for that input scenario;
- If the size of the library is not big enough, since the number of scheduling is fixed, the training set is enlarged in size in order to have at least one input for each label in the scheduling library.

The number n_{batch} is a parameter the user will decide for the learning. It is related to the batch size for the training of the neural network.

2.1.4 Training the network with backpropagation

Once the training set has been created, the network is trained. The gradient descent along with backpropagation algorithm is used for the training.

The cost function to minimize is the MSE, but the algorithm ends when the misclassification in the training set are below a certain threshold ϵ, usually fixed to 5%.

The pseudo code for the training is the following:

```
initialize number of epochs, MSE and weights
while 1
  evaluate the MSE and the number of misclassification
  if number_misc < eps
    break
  end
  for each sample in the training set
```

```
    Evaluate forward network values for the mini-batch;
    Evaluate backward network values for the mini-batch;
    Evaluate the derivatives;
    Update the weights;
  end
end
```

The training process for the full precision network is pretty fast. In fact, a little bit more of 100 epochs are needed to arrive to a precision in the training test of 4% with a thousand inputs to learn, for the FFT problem. 2.6 and 2.7 show how the two quantities decrease during the training.

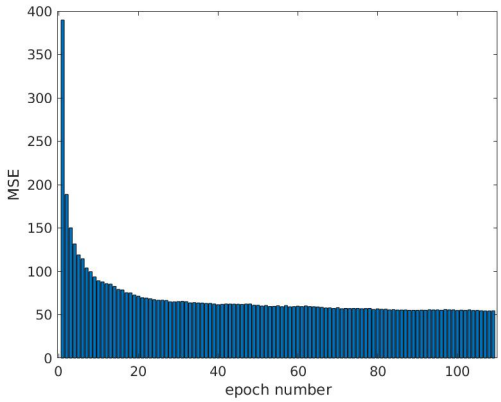


Figure 2.6: MSE decreasing with the training

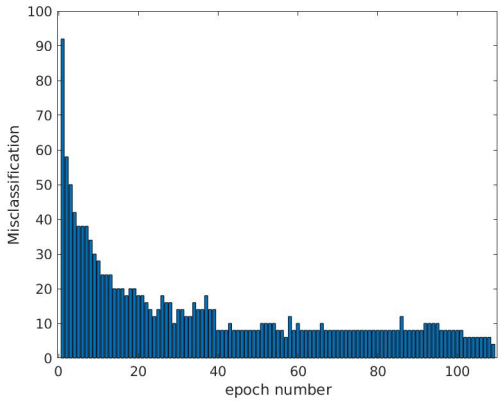


Figure 2.7: Number of misclassification decreasing with the training

2.1.5 Testing the network on a new test set

Once trained, the network is tested on a new set, created coherently with the profile of the application used during the training set, as explained in 2.1.3.

The pseudo-code for the creation of the test set is the following:

```
for i = 1:n_test_samples
  Create a new set of pc and mc following the profiled distribution;
  Evaluate the optimal makespan MS_opt with function (1);
```

```
    Evaluate the minimum makespan in the stored scheduling to assign the
        optimal label label_opt;
    misc = misc + (label_nn == label_opt);
    Use the Neural Network to find the label label_nn addressing the
        memory;
    Evaluate MSj with function (2);
    perf_loss_i = |MS_opt - MSj|/MS_opt;
end
Compute the average of performance loss.
```

New PC and MCmatrices are created following the right probability distribution and then a label is assigned to each input of the test set. Then the new scenario is fed into the network and the output is taken. The label is used to evaluate the number of misclassification, but the real parameter used to confirm the goodness of the design is the percentage in loss. In fact, the proper classification of the input does not mean everything. The output label can provide three different situations:

- *Correct label*: the optimal scheduling is provided as output;
- *Wrong label-Low loss in perfomances*: this situation is tolerated, since the time gained during the computation of the scheduling with this approach is way higher than the time loss for the wrong scheduling;
- *Wrong label-High loss in performances*: this is the worst situation, that has to be avoided. This cause the highest loss in scheduling performances.

If the first two situations are encountered, then the loss in scheduling performances is negligible. The training should serve to avoid the third situation as many times as possible.

2.2 The results with full precision

The results are various. Accordingly to [7], the DAGs normally used to compare scheduling algorithms are:

- FFT (figure 2.8)

- Balanced graph (figure 2.9)
- Gaussian Elimination (figure 2.10)
- Baseband Graph (figure 2.11)

The number of execution units is fixed to three, while the number of tasks in the application changes with the different algorithm to schedule. The hidden layer and output layer number of neurons is instead fixed to 50 and 8. The learning parameter η is equal to the vector [1000 500], where the first number represents the learning parameter used for the first layer of the network and so on. This is an heuristic parameter, meaning that it has to be tuned during the different runs of the program, to provide the fastest training in terms of number of epochs needed.

The network characteristics and performances are presented in table 2.2.

Table 2.2: NETWORK SETUP FOR DIFFERENT DAGS

Network	**FFT**	**Baseband**	**Balanced**	**Gaussian Elimination**
PC size	$[15 \times 3]$	$[10 \times 3]$	$[16 \times 3]$	$[14 \times 3]$
# input	51	36	54	48
Performance loss wrt HEFT	-1.95%	-1.06%	-2.9%	-4.29 %
Performance loss wrt Optimal	-2.65%	-2.05%	-1.4 %	-1.69%
Miss Rate	40%	33%	20%	40%

As it can be seen in the following figures, the learning with different kind of taskgraph is similar. The number of epochs to train is influenced from different factors, such as learning parameter, type of taskgraph or initialization of the weights. Said that, each run of the program converges with different number of epochs and different shapes of the graphs, but eventually, the percentage numbers are very similar.

2.12, 2.13, 2.14, 2.15 show how different quantities vary with the size of the training set for each taskgraph. In these figures the four lines represent what follows:

- *HEFT* line: it represents the gain in scheduling performances with respect to the HEFT result for the same input;

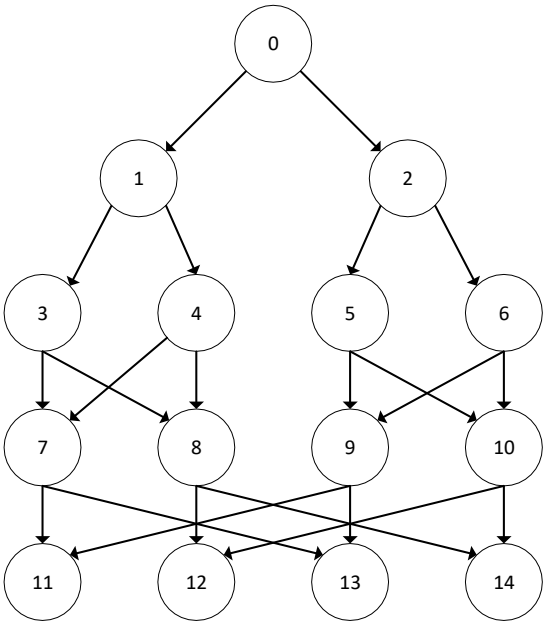


Figure 2.8: DAG for the FFT Problem

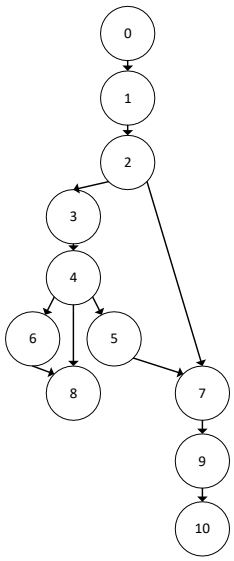


Figure 2.9: DAG for the Baseband Problem

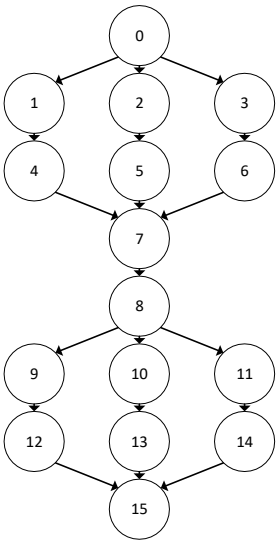


Figure 2.10: DAG for the Balanced Problem

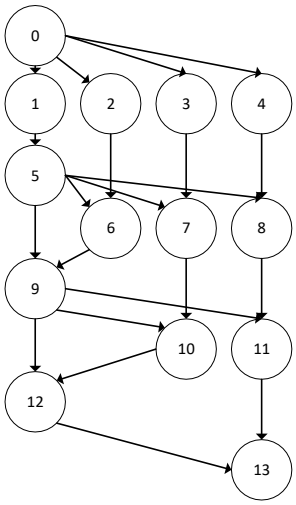


Figure 2.11: DAG for the Gaussian Elimination Problem

- *Optimal* line: it represents the gain in scheduling performances with respect to the optimal scheduling chosen between the stored ones; It is the best indicator

of the goodness for the trained network.

- *Misc*: percentage of misclassifications on the test set.
- *Epochs*: number of epochs to train the scheduler.

In most of the graphs, the trend is not marked. This is due to the fact that the initialization of the weights strongly impact the behavior of the training. The number of epochs to train generally increases with the number of inputs to learn, but it is not strictly related. Furthermore, a trend can be found in the number of misclassifications: in general they decrease with the increasing of training set size.

More in details, for what concerns the *HEFT* line, the performances and the trend really depend on the taskgraph. There are taskgraphs for what the HEFT works better[1], while for others the neural network is able to provide a good improvement with respect to the HEFT makespan. This is particularly true for the FFT. In general, it can be seen as, even if the size of the training set changes, the schedule performances change slightly. This suggests the fact that, during the training, the neural network learns pretty fast how to avoid the big errors. This is likely due to the fact that the distribution of the input is considered gaussian, hence the network is able to easily learn the average case and to make all the similar cases converge in that class. If the gaussian distribution does not reflect the real situation, this statement should be revised. In general, even with a small training set, it can be seen as the loss in performance with respect to the HEFT is negligible: $-[1 \div 2\%]$

For what concerns the *Optimal* and *misc* lines, the trend is the one expected, with the performances improving with the number of inputs in the training set. This is a general statement but is not always true.

[1] balanced and baseband performances of the neural network never overcome the HEFT ones, than it can be said that for thee taskgraphs the HEFT works in a good way

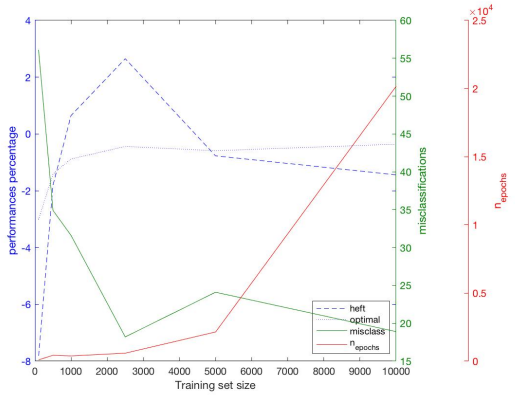


Figure 2.12: FFT taskgraph: scheduling performances

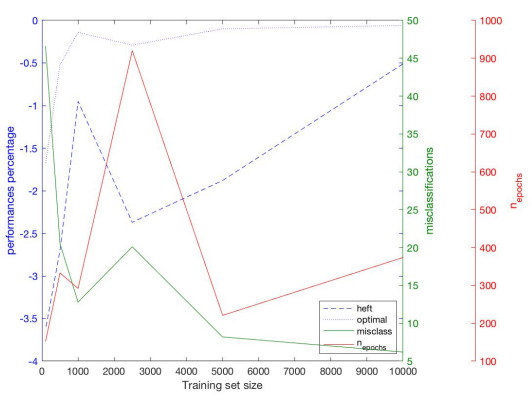


Figure 2.13: Baseband taskgraph: scheduling performances

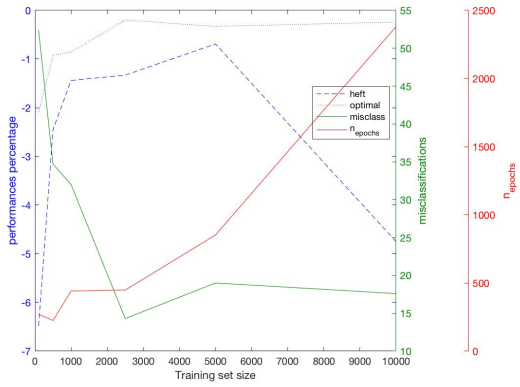


Figure 2.14: Balanced taskgraph: scheduling performances

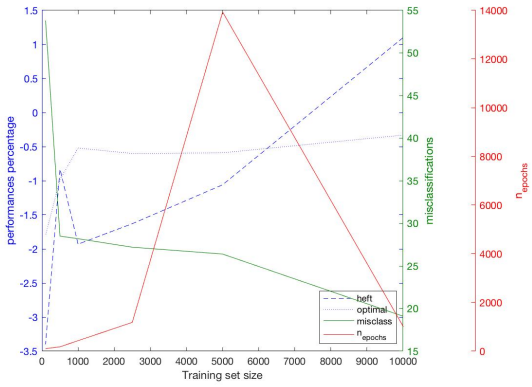


Figure 2.15: Gaussian Elimination taskgraph: scheduling performances

Chapter 3

Moving to Hardware

As seen in Chapter 2, the idea has been validated with the full precision network, but the performances required for allowing real-time operations of the system need the translation of the algorithm in an hardware friendly fashion, that provides the scheduling in the shortest time possible. This chapter will introduce all the modifications needed to make the structure easily moveable to hardware. The approach followed during the work has been to write MATLAB programs that emulate the behavior of the hardware. The structure of the neural network will be completely changed for this reason.

Moving to hardware requires to think about bits instead of full precision numbers. The first idea was to literally translate each information embodied in the neural network to hardware, as suggested in [8], but this would have resulted in times that are not feasible with the problem it is aimed to solve. In fact, keeping in mind what said in the previous chapter, each layer would have required a huge number of multiplications and additions for each neuron. For the network configurations in 2.2, the number of multiplications for the input layer goes from 1800 for the best case, to 2700 for the worst one. Even though all those multiplications could be done in parallel, the total time for a single fixed precision multiplication of 8 bits is high. After that, the summation of all the numbers would have taken even longer. This solution was then abandoned very soon.

However, as suggested by [6], there exists neural networks able to perform classification with both binary inputs and weights, that obviously results in the fastest solution possible, moving all the complexity to the training phase. In fact, it is interesting to notice that the multiplication between numbers of one bits can be represented as:

X_1 \ X_2	-1	1
-1	+1	-1
+1	-1	+1

That, considering the -1 as logical 0 in hardware, is nothing but the xnor port. Since, according to [6], these neural networks are able to recognize properly most of the common artificial intelligence problem, a Binarized Neural Network has been applied to this problem. Moreover, since each layer is going to be binarized, the output of each neuron will be only one bit. The activation function will then result in a counting of the ± 1 coming from all the xnor operations. This is commonly referred to as the *popcount* function.

3.1 Differences with respect to Courbariaux Binarized Neural Network

Although the speed gain obtained by this type of network is remarkable, a choice to better fit the network to this problem can be taken. Few points have been modified, to ensure a faster computation of the output, in particular:

- No *batch normalization* layer has been used;
- The input layer has been treated in a completely different way. In fact, what they suggest is to use just one weight for input and apply the following formula to compute the local field:

$$v = x \cdot w^b \quad v = \sum_{n=1}^{n_{bit}} 2^{n-1}(x^n \cdot w^b)$$

This means that an n_{bit} bit number comes out from the xnor operation. This number has to be summed up with all the other numbers, and a chain of adder would be necessary. A single adder and a loop in the addition may be used as well, but this would force the network to be stacked in a point for longer time, vanishing the possibility of inserting the pipeline as well. Again, the quest for

speed requires to avoid any mathematical structure along with all the loops. For this reason, a new approach has been used. In fact, considering that each input will only be a group of $0s$ a $1s$, it is possible to assign a weight to every single bit. This way, the number of weights required grows, but with it the degrees of freedom of the system, hence the training may be more accurate and the final neural network in hardware will go even faster. It could be argued that this way all the bits would have the same power, losing the information on the MSBs and LSBs. Even though this does not represent the truth, because the neural network trains its weights to reflect the power of each input, we came out with a way to enforce the natural power of the numbers, that is: varying the number of neurons that a bit influences depending on the position of the bit. If an MSB varies all the neurons will be influenced, if an LSB changes, only few of them will change the output value.

This solution increases the number of weights to train and to store with respect to [6], but it allows to have a network with more flexibility and better performances. This means that, once the number of hidden neurons to connect to the least significant bit is decided, then the total number on neurons in the hidden layer is fixed as:

$$n_{hidden} = n_{hidd_{LSB}} \cdot n_{b_{input}}$$

For the FFT problem (51 inputs), with input that can be written down in 6 bits, then, considering 21 neurons connected to the LSBs, the number of hidden neurons is:

$$n_{hidden_{\text{FFT}}} = 21 \cdot 6 = 126$$

and the number of inputs:

$$n_{in_{neurons}} = n_{in} \cdot n_{b_{input}} = 51 \cdot 6 = 306$$

Resulting in a total number of weights for the hidden layer of:

$$n_{InWeights} = n_{in} \cdot \sum_{i=1}^{n_{b_{input}}} i \cdot n_{hidd_{LSB}}$$

$$n_{InWeights_{\text{FFT}}} = 51 \cdot \sum_{i=1}^{6} i \cdot 21 = 22491$$

The output weights are instead:

$$n_{OutWeights} = n_{hidden} \cdot n_{out}$$

$$n_{OutWeights_{\text{FFT}}} = 126 \cdot 8 = 1008$$

- The *activation function*, for the last layer, is a linear ramp function with saturation to ±1 after the input overcomes a given threshold. In fact, even if the hidden layer will only have an output of 0 or 1, the output layer will still need to evaluate the first highest local field between all the output neurons. This way, whatever is saturated to an high or low output will be cut off to the saturation value. The shape is the one in figure 3.1. In neural networks, the edges of a function reduce the learning ability of the network. MATLAB tests demonstrate that, in this particular case, the network learns, even if more slowly.

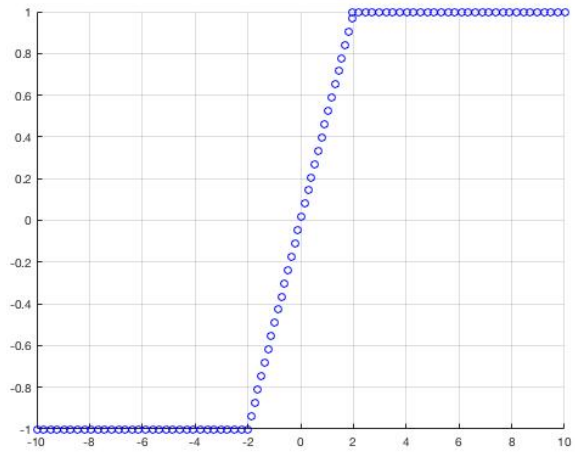


Figure 3.1: Activation function for the output layer of the binary neural network

The final neural network is then the one present in 3.2 and 3.3.

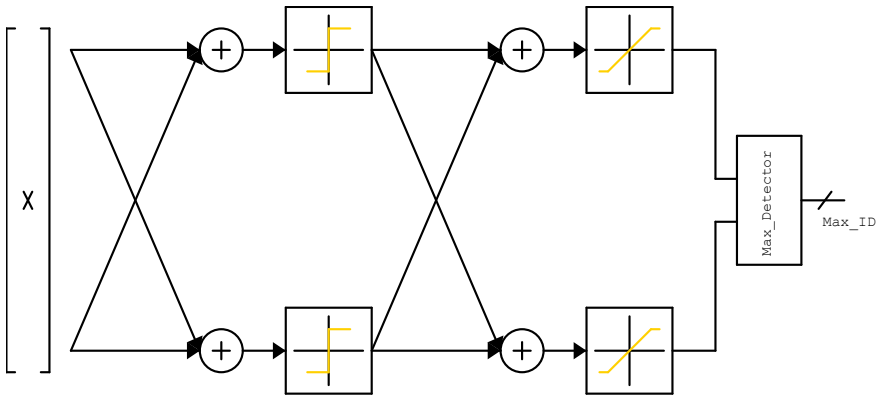


Figure 3.2: Final binarized neural network.

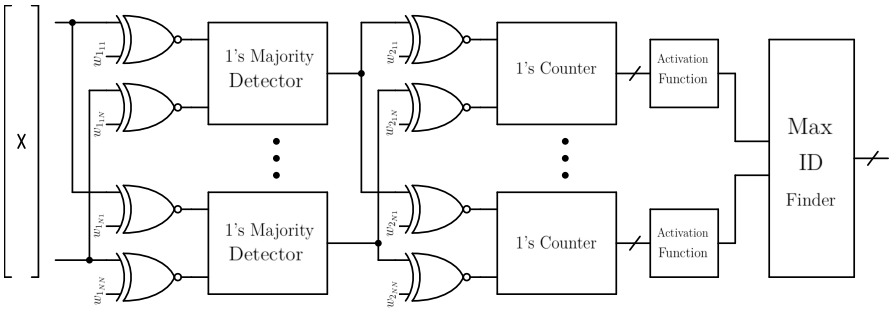


Figure 3.3: Block scheme of the final neural network.

3.2 The Binarized Neural Network Training phase

The training works exactly as explained in [6]. Considering that the network learning comes from the change in sign of the local field, it is important to initialize the weights very close 0. Moreover, the η has to be small as well, to keep the not-binary weights close to 0 and foster the learning. This means that the learning will be slower than the full precision one. Moreover, since there are only two outcomes from the xnor port, the training is mostly not linear. In fact, as it can be seen from 3.4 and 3.5, it is very noisy. It can be noticed as the MSE decreases in a steadily way, but the misclassifications graph has similar envelope to the one with full precision, though the edge goes up and down.

In this example, the FFT problem has been solved. In 4.2, the results for all the taskgraphs analyzed are presented. It can be seen as they are similar to the ones of the network with full precision.

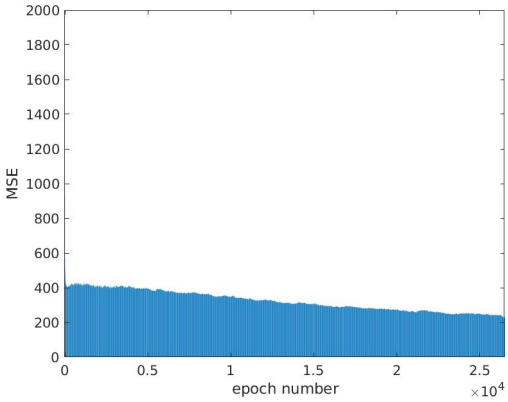


Figure 3.4: MSE for the training of the BNN.

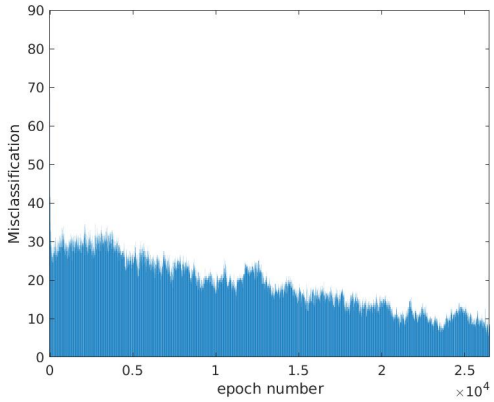


Figure 3.5: Misclassification for the training of the BNN

Table 3.1: SCHEDULING PERFORMANCES OF BNN

	Epochs	Performances wrt HEFT	Performances wrt Optimal	Misclassifications
FFT	26520	+1.52%	-2.2 %	52 %
Baseband	27700	-4.62%	-3.39%	64.3 %
Balanced	25107	-4.53%	-3.05%	65 %
Gaussian	9768	-1.55 %	-1.26%	55 %

The training size for this kind of structure is of 200 samples, in fact the results in performance as a classifier are not high, but it is enough for avoiding critical misses in the test phase, as the scheduling performances results remain remarkable.

3.3 How to move each structure to Hardware

Henceforth, the detailed hardware translation of each single block is carried over. The general structure, applicable to every network with any parameter, will be presented. In particular, it will be covered:

1. The multiplication;
2. The block *Addition + Activation* for the hidden layer;
3. The block *One's Counter* for the output layer;

4. The comparator in output;

5. The activation function in output;

Eventually, a possible solution to tackle the multigraph problem is suggested.

Before starting the presentation it is worth to speak about the two possibilities faced for what concern the *popcount* block, where it is aimed to find out wether there are more $0s$ or $1s$ in the input vector:

- A completely digital structure: this solution fits perfectly the rest of the structure, but it may not be the best. In fact, the number of one's to sum up is high and this would result in the most time critical piece of hardware in the whole system.

- A mixed-signal structure that embodies both the operations in just one clock cycle. This kind of structure does not provide the number of ones coming out from the xnor operation, but directly the bit corresponding to the majority in the input vector. This will be presented in details in the following, but it is important to mention it now, to understand the reasons behind some choices.

In the second layer, the first solution is mandatory. In fact, there is a comparison in magnitude of the output, hence it is important to know the precise quantity of $1s$ in the input vector. In the meantime, the hidden layer may exploit the second solution, to have a faster output computation. However, being a mixed-signal structure, the transistors will be active for most of the time, resulting in a high static power consumption. Anyway, the number of transistors for this solution is lower. This means less capacitances switching and a lower total power consumption. Depending on the needs, different choices may be taken. Several trade-offs has to be evaluated any time this structure has to be finalized. They will analyzed in the following, since a more detailed knowledge of both the structures is needed to compare and constrast the two solutions. Moreover, it is very structure dependent, in the sense that the number of inputs, the environment where this hardware accelerator aims to be put on, and other factors change the specific solution from time to time.

3.3.1 Multiplication

The multiplication is performed by means of xnor port, massively parallel. The output of the xnor operation goes to one of the structures explained in the following. Due to the complex structure of the input, the user should take care about the connections with all the layers. However, thanks to the generalization performed in the code, and explained in the next chapter, this point does not represent a real problem.

3.3.2 Digital Structure

The digital structure is made up of two components:

- The one's counter;
- The comparator with the threshold;

The first structure is a Wallace Tree that sums up all the bits of the same power together. The idea is a generalization of [9]. It consists in the usage of common $3:2$ *compressors* (full-adders) and half-adders to reduce the entire vector of bits into a vector stating the number of ones in input.

Generalizing the structure is complex. To increase the operation frequency, the aim is to use the least number of layers, to reduce the critical path length. For this reason, in each layer, the highest number of full adders have to be inferred. Hence, the number of bits left for a given column has to be divided by three (the possible input of a full adder). The possible results of this operation and the policies applied are:

- $Reminder = 0$: from the given layer all the inputs go in full adders. The next layer will have the n_{fa} wires of the same column (sum bit of the full adder) and n_{fa} wires of one higher column (carry out bit of the full adder).
- $Reminder = 1$: from the given layer all the inputs but one go to full adders. For those inputs, the outputs will be the same as in the previous case. The additional wire will be forwarded without any operation to the next layer.

- *Reminder* = 2: from the given layer all the inputs but two go to full adders. Again, the behavior is still the same for those wires. The two remaining wires will go into an half adder, producing one wire in the same column and one of higher column for the next layer.

This way, looping until all the columns have only one bit, it is possible to create the total structure.

The output layer will have more than the $\lceil log_2(n_{in}) \rceil$ necessary for representing all the possible ones in input, but, since there is no possibility that a 1 arrives to the last bits, it is possible to get rid of that wires, that will not be connected in output.

The following is the pseudo-code for the inferring of the structure:

```
layers[0][0]=   n_in;
i           = 0;
missing       = 1;
while(missing!=0){
  Initialize the values for the current layer
  for each weight v present in the layer
    Assign to next-layer/same-weight number of wires: "floor(layers[i][
     v]/3)";
    Assign to next-layer/next-weight number of wires: "floor(layers[i][
     v]/3)";
    Infer "floor(layers[i][v]/3)" FAs;
    Evaluate reminder
    if(   reminder == 1){
      Forward the wire to next layer
    }elsif( reminder == 2){
      Infer one HA with the remaining wires
    }
  }
```

3.6 shows the structure for an 8 bit input.

When the number of bits in input the structure becomes higher, the critical path length increases. This increase is not linear because it depends on the number of layers of the structure, and increasing the number of bits that number of layers does not increase linearly.

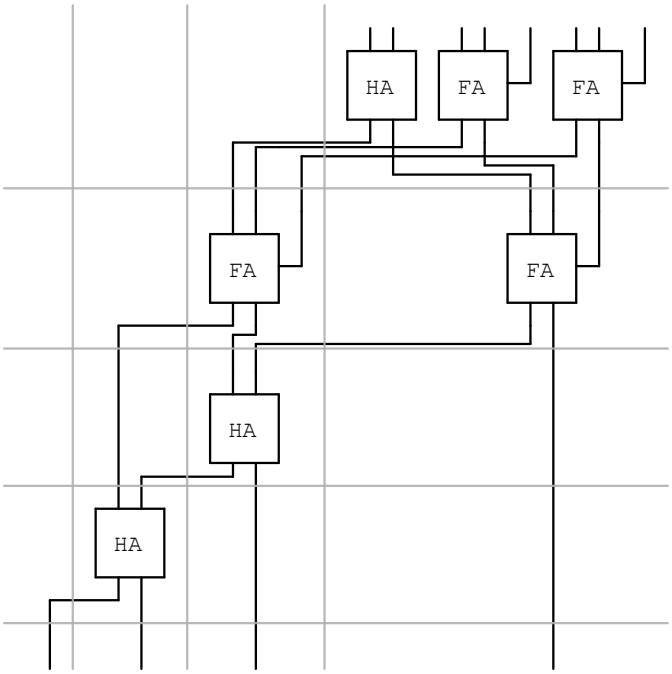


Figure 3.6: 8 bit One'n Counter

The output of the counter is then fed into a comparator with a threshold that gives 1 in output if the number is higher than or equal to $n_{in}/2$.

3.3.3 Mixed-Signal Structure

With a lot of input, the digital solution may insert a long delay. That is why a mixed-signal structure has been developed. This structure embodies both the summation and the comparison with the threshold input. The structure is presented in 3.7.

When the number of $1's$ is higher than the number of $0's$, the output is pulled up to 1 and vice-versa to 0. The working functionality of this hardware is the following: the n-mos and p-mos connected to the CK signal are useful to pre charge the line. If the number of $1's$ is not as high as half the number of the inputs, the output stays pulled up to 1 and the output after the comparator will be 0. The timing driagram in 3.8 shows how it works. In the example, the structure has 3 inputs that switch after the first clock cycle. The *top* line is always pre-charged to 1 and during evaluation the voltage of this line decreases with the number of ones in input. Correctly dimensioning the transistors in the structure, it is possible to

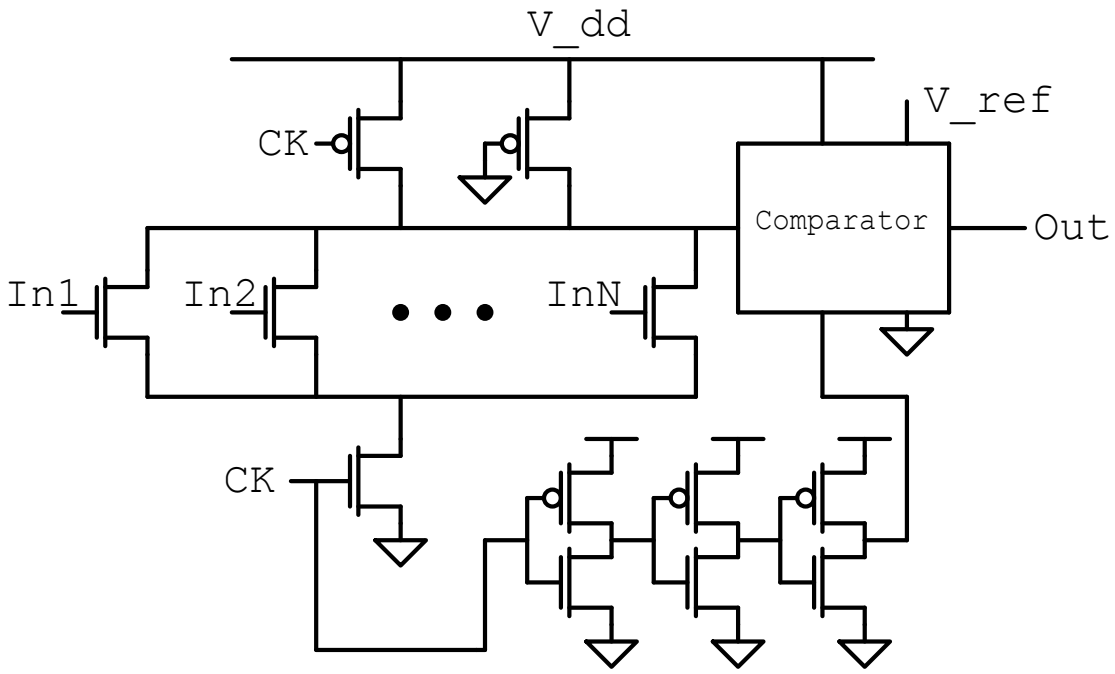


Figure 3.7: Hidden activation function

find the voltage of the required threshold V_{ref}. That voltage is one of the input of the comparator in output. This comparator will provide with a 1 in output when $V_{top} < V_{ref}$ and a 0 otherwise. Since the activation function has to set to 1 the output when half or more of the inputs are one, the ideal threshold condition should be put in the middle between:

$$\begin{cases} \lfloor n_{in}/2 \rfloor \; and \; \lfloor n_{in}/2 \rfloor - 1 & if \; n_{in} \; even \\ \lfloor n_{in}/2 \rfloor + 1 \; and \; \lfloor n_{in}/2 \rfloor & if \; n_{in} \; odd \end{cases}$$

Anyway, in the next chapter it will analyzed as this does not represent the truth, since the comparator is influenced by the rest of the circuit it is inserted in. That is why the tuning of the V_{ref} represents one of the key point for the functionality of this structure. This allows to avoid also the fixed errors due to process variations during the fabrication, but the error due to the actual condition of the chip still has to be taken into consideration.

Moreover, the comparator works in a dynamic way as the rest of the structure.

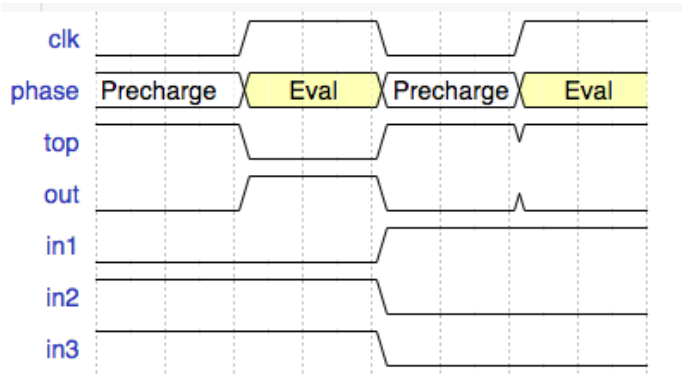


Figure 3.8: Timing hidden layer: explain it better in the final version

From its nature, it evaluates on the lower level of its clock, this means that, to make the structure work, the clock has to be inverted. To make the clock go fast enough, a chain of inverters is needed, but to have the real inversion of the clock, the number of inverters has to be odd. This way, the load seen from the clock is still the same, but the switch happens faster.

Another thing that has to be considered is that this structure may not be perfect. Instead, it may have some noise that has to be tuned before using it. Process variation may impact this factor as well. However, this does not impact the performances. In fact, from MATLAB simulations, it can be noticed as, if the noise remains very low (a percentage of decision that is pretty low), the performances are impacted in a very few extent.

The positive outcomes of this structure are:

- The fact that it is able to perform both the operation in almost no time;
- The area is really small, being the number of transistor negligible with respect to the digital structure, resulting in lower total power dissipated.

However, some downsides are there as well:

- Static power consumption is important: the pmos is always active and the comparator is analog, meaning that it is always active;
- If the output is 0, the load switches at every clock cycle, since the output is pre-charged. If the load is high, the power consumed is high.

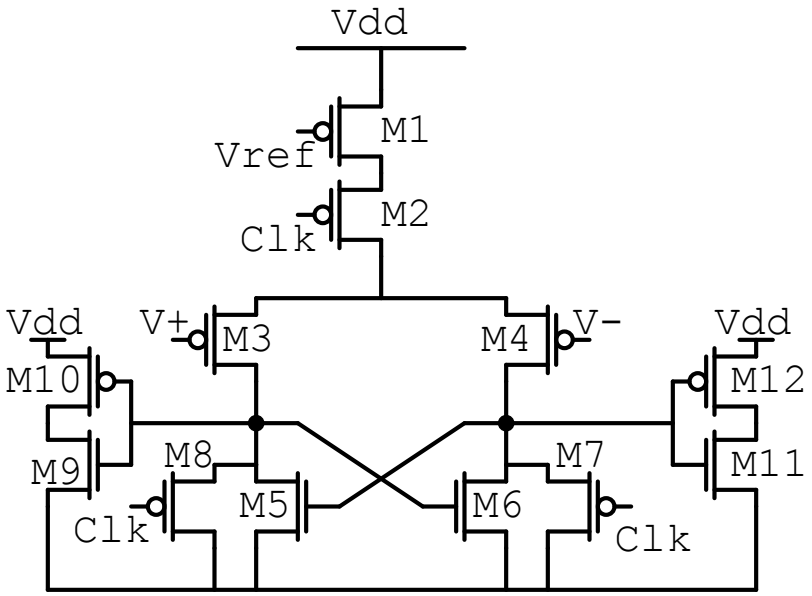


Figure 3.9: Analog Comparator

- If the number of inputs is too high, the threshold between the condition is not going to be precise, meaning that it would be impossible to divide the two situation ideally. This means that an error may be always present.

The comparator structure is the one presented in figure 3.9. The output is differential, but for the scope of this project, only the negated output will be taken. During the high level of the clock the output is pre-charged and during the low one, it will go in evaluation. From spice simulations, it came out that the threshold has to have a precision in the order of $1mV$. This comparator is totally able to cope with this precision.

When clock is 1, the drain of the p-mos $M3$ and $M4$ is connected to ground through $M7$ and $M8$. For this reason, both the output will be pre-charged to V_{dd}. $M2$ makes the sources of $M3$ and $M4$ detached from the power supply. When instead the clock becomes 0, the p-mos $M2$ goes on saturation and connect the source of $M3$ and $M4$ to V_{bias}. At that point, depending on the difference of voltage in their gates, more or less current starts flowing in each of it. The positive feedback, created by $M5$ and $M6$, enhances the difference between the two terminals, moving the voltage to the one that will be present in output. In particular, the small difference in

voltage will make reach the interdiction condition to one of the two n-mos ($M5$ or $M6$), forcing even more the current in the other, that will go in saturation. The output will reach the stable configuration and stay to that voltage up to the point that the clock does not switch to 1 again and the output are both pre-charged to 1.

The inverter in both the outputs are more than one in the real structure, of increasing sizes going towards the output. This is used to create a stronger and faster signal in output. For this reason, the total number of transistors used in the real structure is 28.

3.3.4 All the possible trades-off between the two structures

Earlier, several times the existence of trades-off between the mixed-signal and digital structure have been mentioned. This section aims to give a guide on when to follow the first or the second route depending on the specific number for the final structure.

In general it has to be considered that:

- **Critical Path given from the output comparator**: in that case there is no need of implying the mixed-signal structure, since the way the pipeline registers are inserted makes the comparator the bottleneck. For this reason, the digital structure has to be preferred. This happens when the number of output neurons is high and the comparison stage is going to take longer than the different neuron calculations themselves. Furthermore, the digital structure is way easier to design, since, even with different technology and libraries to develop the physical circuit, there is nothing that needs to be re-designed. Differently, using the analog structure, a re-design of the transistor size has always to be carried out. Moreover, since the way connections are created in the new Binarized Neural Network, the activation functions to re-design would be multiple ($n_{bit_{in}}$[1]). However, if the number of inputs is too high the mixed signal structure cannot be used.

- **High number of inputs**: in this case the digital structure would take long.

[1]This is because there are $n_{bit_{in}}$ different neurons in the hidden layer, and one kind of neuron in the output layer, but the one in output always needs to be digital, and that one can be fully generalized. Here, for kind of neuron, it is meant a neuron with the same number of inputs, that is, a neuron that will switch after a $n_{in}/2$ number of ones are fed into it.

The idea of implying half-clock cycle for evaluating the output is what makes the analog solution preferable. However, sizing the transistors becomes difficult in this condition and a very precise comparator in output is needed, resulting in a costly structure. From experiments, with $V_{dd} = 1.1V$, this structure can be used up to 400 inputs.

3.3.5 Output layer

As aforementioned, the output layer needs to have the count of the ones coming out from the xnor operations. For this reason, the digital structure is compulsory. After it, the winning neuron is decided by selecting the highest local field. To detect which is the winning neuron, it is necessary to insert one more layer to the structure: a majority detector that is able to compare all the inputs and come out with the index of the highest one. From the theory point of view, the activation function should be there as well. However, from MATLAB simulations, it can be demonstrated like the performances are not changing even without applying the saturation layer. This is due to the fact that the output are in most of the cases into the saturation limits, then even without applying this layer, the output does not change. Again some tradeoff on this comes out. In fact, to achieve the fastest structure two situations may show up, but they will be covered in the future, when the structure of both the activation function and the comparator have been presented.

The structure of the majority detector is explained in the following section.

Majority detector

After the number of ones is evaluated, the maximum has to be detected.

While the software simulations use 1 and -1, the hardware structure works with 0 and 1. This does not take to any modification, since the majority detection in output works exactly in the same way. The winning neuron will be the one with the highest number of $1s$ coming out from the xnor operation, meaning the highest local field.

The structure of the comparator is the one presented in figure 3.10. It has a tree of $log_2(n_o)$ layers. The ID of the scheduler is carried in output by means of a bunch of multiplexers.

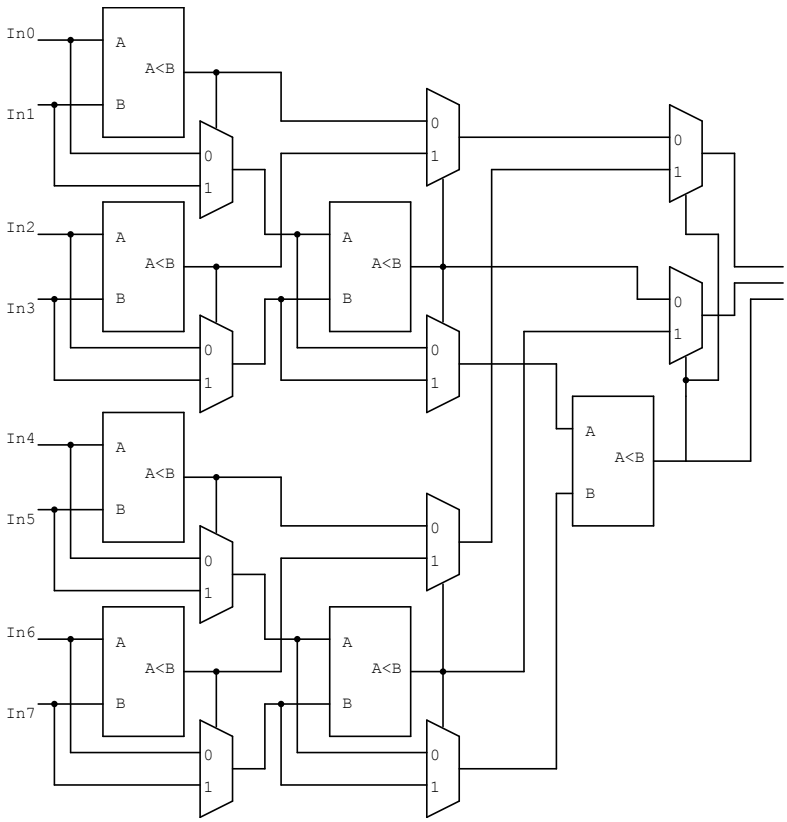


Figure 3.10: Output comparator

The single comparator is presented in figure 3.11. It compares bit-by-bit the two numbers, due to the small amount of bits to compare.

The structure follows perfectly what the *max* function in MATLAB does, considering the first maximum output that is found from the index 0, even if more than one maximum is present.

Activation Function

The activation function, as presented in figure 3.1, is nothing but $in = out$ when the number is inside the threshold. Hence, the general structure is going to be the one presented in 3.12.

The threshold has to be designed depending on its value.

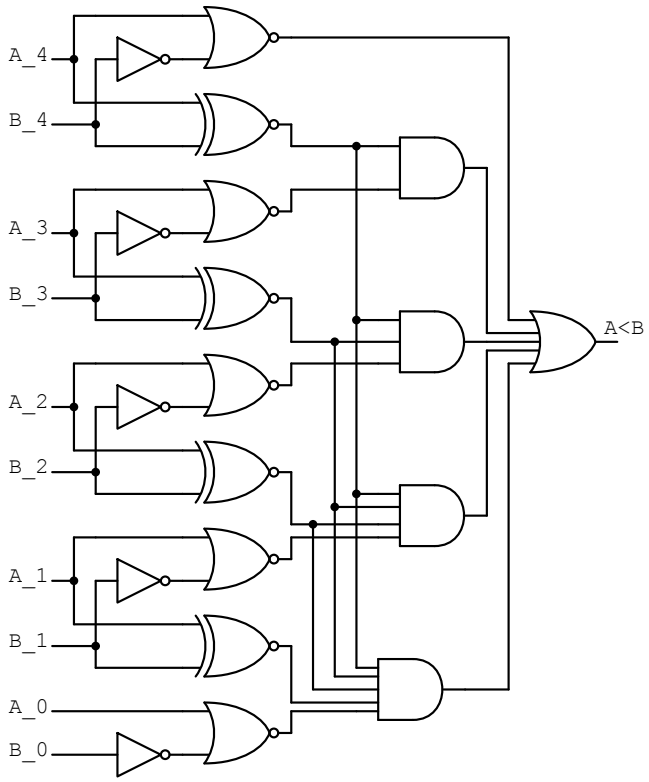


Figure 3.11: Single comparator

Activation function or not

Knowing how the structures are done, it is necessary to think about the possible tradeoffs between the activation function and the comparator.

- If the local field has high number of bits, the comparator in 3.11 will go slower. This enters in all the comparisons, meaning that, to find the total delay of the output comparator, the delay in the bit-by-bit structure is multiplied by the number of layers. In this case, it is convenient to insert the activation function, reduce the number of bits needed in each comparison and make the total structure go faster;

- If the number of bit in the local field is not high, MATLAB simulations demonstrate how the presence or not of the activation function does not change a lot the results, because most of the local fields does not reach saturation. For this reason the activation function can simply be avoided. It is important to point

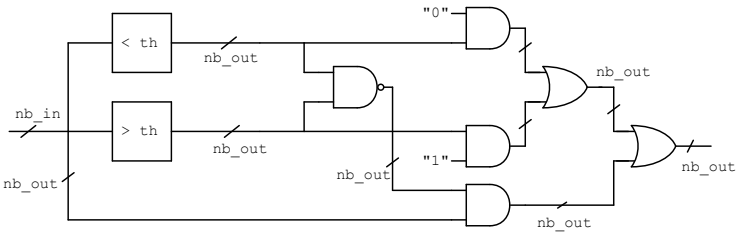


Figure 3.12: Hardware Realization for the Activation Function

out how, though the structure is not required during the real-time utilization phase, it plays a key role in the training, making it faster. In fact, without a clipping layer between ± 1, the difference to apply in the backpropagation algorithm between the ideal and the real output would be too big and the training phase would take longer[2].

3.4 Pipelining the structure

The best way for increasing the throughput of this massively parallel structure is to insert the pipeline. To make this, the position for the pipeline registers has to be optimal. From this point of view, there are two different pipelines to apply, depending on the activation function chosen. In fact, the mixed signal structure strongly depends on the clock behavior, and then it is important to synchronize the rest of the system to it.

[2]Note that the update should be very close to 0 and should not move every of the integer values far from 0, as explained in section 3.2.

3.4.1 Pipeline with digital activation function

If the digital block has been used, then it is important to evaluate the performances of the last stage, to compare them with the performances of the rest of the system and find the right number of stages to divide the structure into. In fact, depending on the speed needed, it is possible to divide the whole hardware in several stages.

In general, the structure representing the bottleneck is the one's counter. As it will be described in the following, the structure has been totally generalized. One of the inputs will be the number of pipeline stages to insert and a software will rewrite the code inserting the registers in the proper position. In fact, considering the pseudo code analyzed before, it is possible to add a layer of registers to the whole vector before feeding it into the next stage of full adders, half adders, or wires.

The code, detailedly presented in the next chapter, allows to structure the pipeline as presented in 3.13. All the registers can be inserted or not just changing a flag or a number in the structure. The N inside the registers means that the user will indicate a number of pipeline registers that he wants to use, and the F means Flag, saying that the user may set or reset a flag to insert or not the pipeline register.

If the engineer decides to exploit the pipeline, then he will be required to add registers in the weights as well, considering that the user will provide all the inputs for a specific schedule in one time, and then it will be duty of the structure to move the numbers inside it, depending on its own timing requirements. This registers have not been inserted in the picture for seek of clarity.

3.4.2 Pipeline with mixed-signal activation function

If the mixed-signal structure is inserted, the time is forced by this structure. In fact, considering that the analog structure is fast enough to be able to evaluate the output in less than half clock cycle [3], the structure must have the inputs ready before the clock goes to 1. This means that the structure has to be as presented in figure 3.14.

In particular, the inputs to the analog structure have to be ready before the 1 arrives. To avoid to lose half clock-cycle, the input can be sampled on the falling edge of the clock. Moreover, the output of the activation functions have to be sampled on the falling edge as well, otherwise the value of each output would be lost.

[3] This assumption is valid for all the number of inputs considered in the simulations.

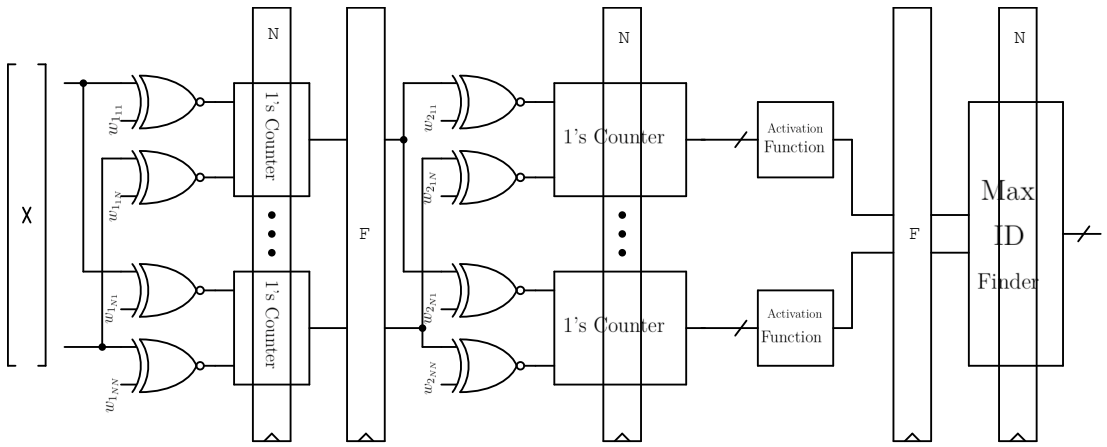


Figure 3.13: Pipeline inserted in the digital structure

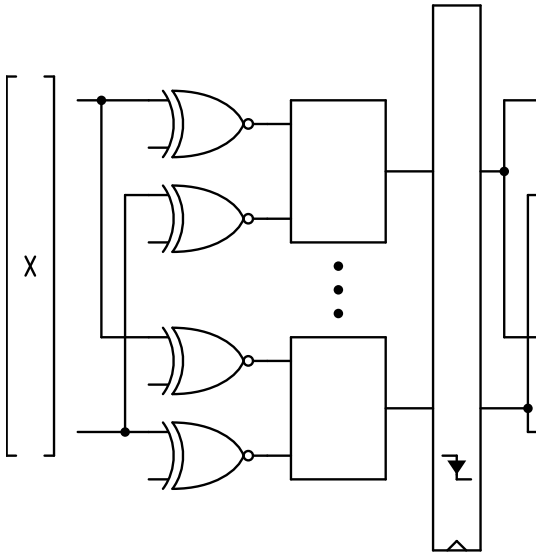


Figure 3.14: Timing of the mixed signals structure with pipeline

After that, the timing can be considered exactly as the one before, and it depends on the clock speed needed. For this reason, the structure is not reported in this picture.

A timing diagram for this structure has been report in figure 3.15

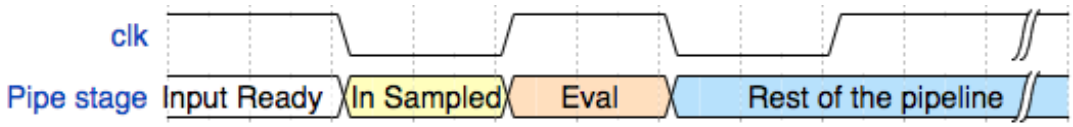


Figure 3.15: Timing diagram for the analog structure

3.5 Multi-TaskGraph Problem

Now that the hardware structure is given, it is possible to think about the generalization for different kind of taskgraphs. In fact, if the weights are precomputed, and the inputs are given in real-time, as long as the sizes are constant, it is enough to provide the values of the weights every time a different application needs to be scheduled and the network will be taskgraph-specific. This allows the generalization to every kind of taskgraph. In fact, if during the installation of an application in a general purpose system, the network is trained with the correspondent taskgraph, then it is enough to store the trained weights in the memory and read them every time they are needed. This kind of solution, giving the weights always available in the right time, does not affect in any way the speed of the accelerator.

A necessary condition for this remains that the size of the taskgraph, the number of execution units available to each application and the number of scheduling stores are fixed. This opens to a series of new problems that will be addressed in section 5.1. In particular, since not all the applications can have the same number of tasks to run, the idea is to create *Graph Partitioning algorithms* that, if the taskgraph is too big, splits it in two different problems, in such a way that the algorithm may be rescheduled in two different iterations of the scheduling in the accelerator. Moreover, considering the pipeline, the solution will be provided in just n clock cycles, where n corresponds to the number of times the taskgraph has to be split to fit into the network.

Nevertheless, the solution that could generalize the structure is provided in the following section.

3.5.1 The computer architecture solution

The proposed solution (3.16) comes from the computer architecture structure.

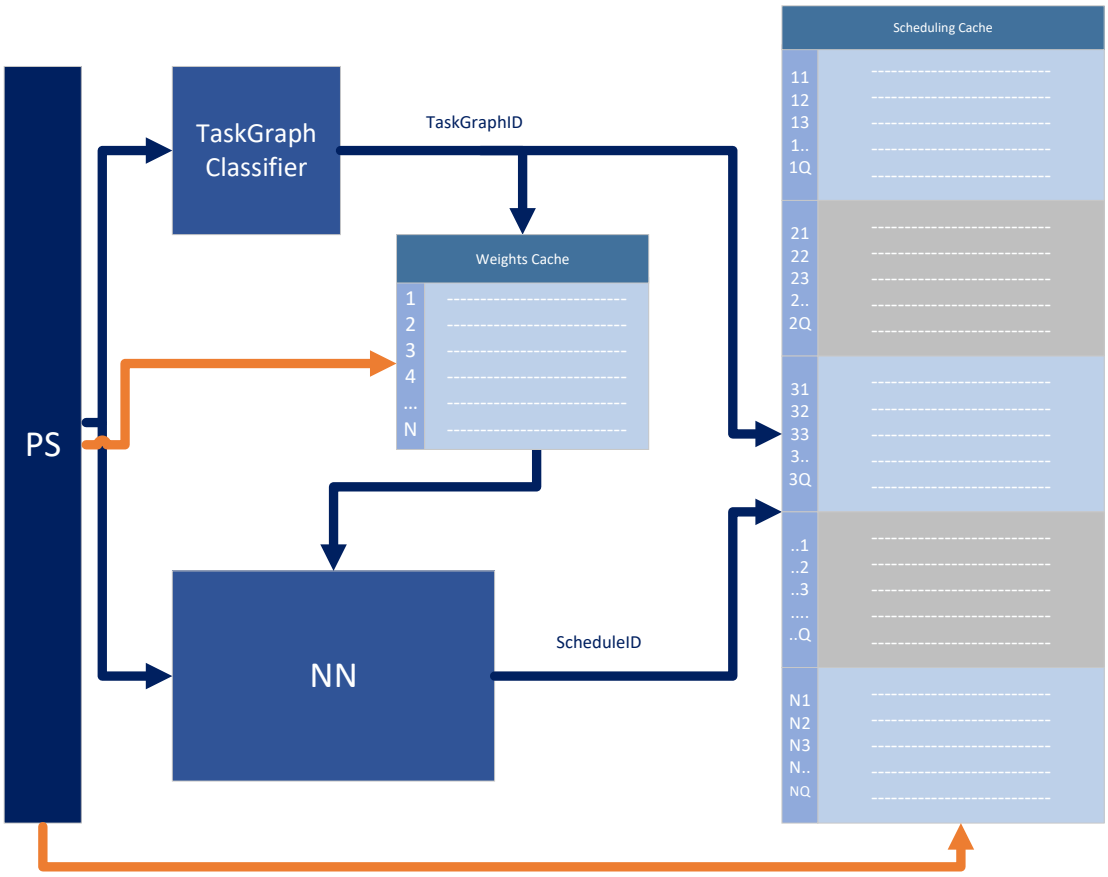


Figure 3.16: Generalized solution

With this solution each taskgraph that the network is able to schedule has one ID associated and weights and reference scheduling are stored in two very fast caches. As soon as a new taskgraph to schedule arrives, a block performs its classification providing the ID. This is then used for two different purposes:

1. It points the weights cache;
2. It creates the first part of the address for accessing the reference-scheduling cache.

The neural network will then compute the second part of the address for the scheduling cache and after the whole system has performed all the calculations, the scheduling is sent as output of the system. The inputs arrive from the previous steps (PS), referring to the ones presented in 1.1

As per the specified solution, the network has to be pre-trained for each specific taskgraph and the size has to be fixed. That opens to the study on how the partition of the taskgraph should be done when the number of a new scheduling does not correspond with the size of the system. This point will be seen in more details in 5.1.

This system has been named *Computer-architecture solution* because, using caches, it can exploit all the well-known algorithm about cache replacing and hierarchical memory.

Chapter 4

Hardware Realization and Overall Perfomances

This chapter introduces a particular solution, coming out with some real hardware performances.

During the chapter, the type of network used, the way the verilog files are created, and the performances results for this particular architecture will be analyzed.

4.1 The Final Network

The final network is the one that has been trained in the previous chapter as an example. The specs are:

Table 4.1: EXAMPLE NETWORK

Taskgraph	FFT
Input Neurons	306
Hidden Neurons	126
Output Neurons	8
n_{bit} out for training	5

Then, once the network has been trained in MATLAB, the weights are passed to hardware and treated as normal inputs to the accelerator. The performances reached are the one presented in 4.2.

The structure analyzed is fully digital, but one example of sizing for the analog structure is provided as well.

Since the purpose of this chapter is to provide with a user guide, hereinafter, the design flow to follow will be presented. In particular, the idea is that, given a clock for the structure, that is fixed, the engineer wants to achieve that frequency

and, therefore, he needs to insert pipeline registers in the architecture. In this case, the user should start with the synthesis of the single bottleneck structures as well as the total structure without any pipeline. This allows him to understand the number of pipeline stages to insert. Then, just inserting the number of pipeline registers, the software will re-write the whole structure code, to make the frequency higher, reducing the critical path and the throughput, paying with latency.

During this chapter, some rules of thumb for performances in the structure will be given as well. This way the user can avoid to synthesize the structures to have approximate numbers to use in the dimensioning of his hardware.

The goal during this chapter will be to create an hardware able to cope with an environment that has a $1GHz$ clock.

4.2 The complete generalization of the verilog code and the folder structure

All the hardware structures are massively parallels and the generalization requires a lot of support variables. In fact, considering structures like the ones presented in the previous chapter, it is necessary to consider power series of number that are not easily realizable with the restrictions of verilog.

For this reason, the generalization is achieved by means of perl scripts that write down the already unrolled verilog code. This technique allows a more readable code (everything is already explicit in the final verilog) and gives the user a lot more power while writing the code. Moreover, thanks to this technique, some code that was not generalizable becomes so. This standard, massively used in tech companies, is what made the writing of the verilog code much easier. Following the same technique, the testbench is written down as well. In this way only the very basic verilog code is written (registers, adders, multiplexers and so on).

Since all the structures are dependent only on a restricted number of inputs, then this idea has been even more generalized. In fact, if all the paths are written down in the optimal way and all the tools are installed, it is enough to give the command *run* to a given terminal to produce all the files necessary for simulation/synthesis in the verilog folder.

The structure of the folder is shown in 4.1.

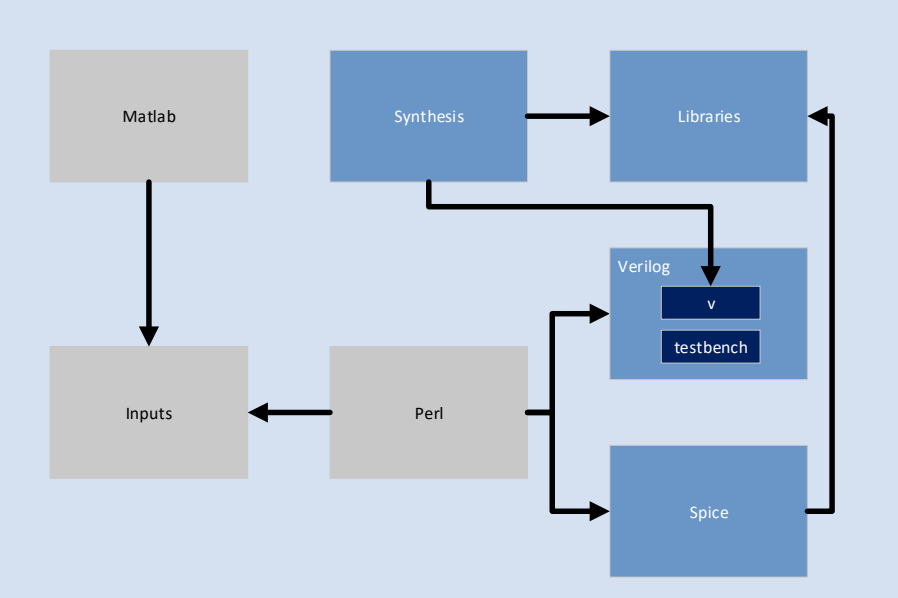


Figure 4.1: Folder structure

The folder is structured like:

- *Input*: it contains all the inputs for the network. All the parameters for the network are considered input as well.
- *Matlab*: it contains all the matlab files. MATLAB is used to compute the weights for the network. It will then write in the *Input* folder all the files with the relative inputs.
- *Perl*: it contains all the perl files, in particular:
  - *hidden_neuron.pl*
  - *comparator.pl*
  - *one_counter.pl*
  - *one_majority.pl*
  - *scheduler.pl*
  - *scheduler_tb.pl*

Where *one_counter* represents the digital activation function while *one_majority* the analog one.

- *Libraries*: it contains all the library files that are used during the spice simulation for the analog structure and the synthesis one during by SYNOPSYS.

- *Synthesis*: it contains all the synthesis files for SYNOPSYS and the script to run the synthesis in the easiest way.

- *Verilog*: it is the folder where all the verilog files are written down. The synthesis reads the file directly from here (*./v/**. Moreover, the testbench folder contains all the testbenches for the structures.

- *Spice*: here the spice file and simulation results are printed down. The input combination for the file are read from the *Input* folder and printed out. This way it is possible to size the structure.

The folders that the user is supposed to touch are *Inputs* and *Matlab*. Moreover, the user is supposed to take actions also in the *verilog/testbench*, *spice*, and *synthesis* folders, but only to run simulations and dump out results.

To make the flow even faster, a script that runs all the needed programs is created. It is the *run.sh* in the main folder. This goes to the perl folder and run all the programs needed. Then it moves to the verilog folder and run the *compile.sh* script, that compiles and checks for errors of all the verilog files.

Moreover, to increase the reliability of the whole process, all the files created from a perl script have restricted writing permissions and have an header that explicitly states not to touch them. The final user should only modify the *Input* folder to create the new structure.

4.3 Performances Results

The synthesis results are provided for the single critical structures at first. After that, results for the whole structure will be shown, for both the not-pipelined and pipelined hardwares.

The synthesis process for every structure has been to create a script that, after the analysis and elaboration of the design, goes with the compilation of the exact map (no optimization requirements from any point of view) and, after that, dumps out all the required power/area/time reports. For synchronous structures, it creates the clock for the structure before the printing of the reports.

4.3.1 One's Counter

The synthesis is done for all the possible One's Counter. In fact, considering six bit inputs, there are 7 different types of one's counter, in particular:

Table 4.2: ONE'S COUNTER ANALYZED

n_{bit}	51	102	153	204	255	306	126
Neurons	H $0 \div 20$	H $21 \div 41$	H $42 \div 62$	H $63 \div 83$	H $84 \div 104$	H $105 \div 126$	O

Where H means hidden and O means output.

The performances results are shown in the 4.2.

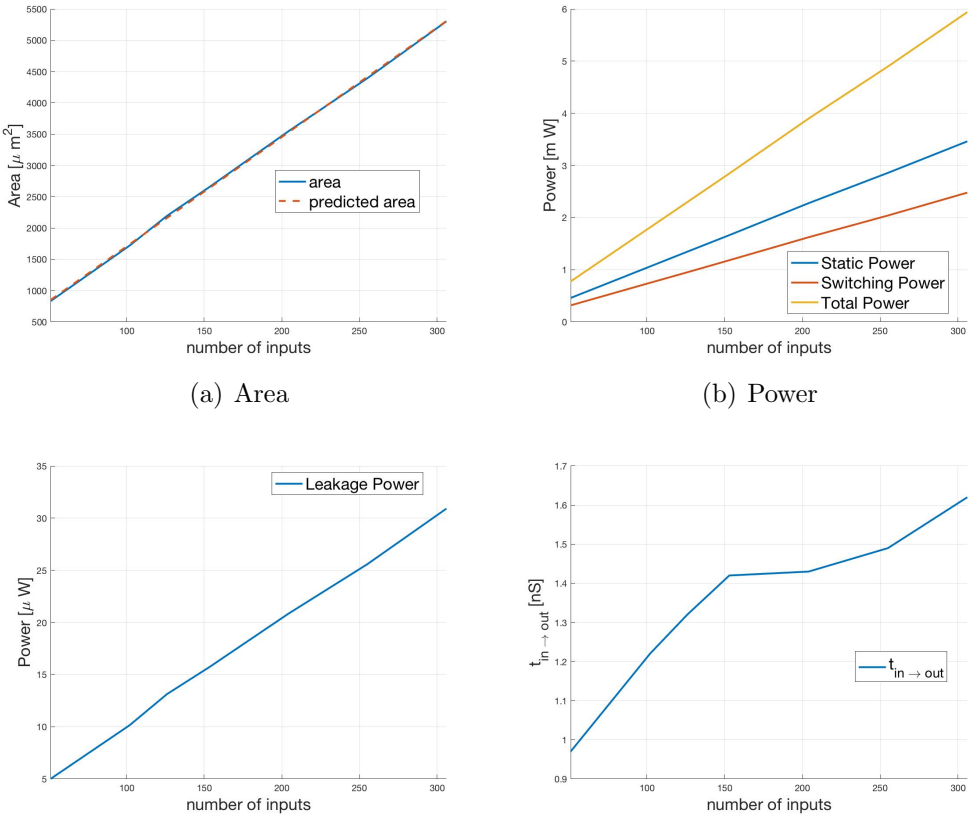
From 4.2, it can be seen as area and all the measures of power grow linearly with the number of inputs. For this reason, to consider an approximate value of all the powers while dealing with the structure, the following first order polynomials are provided:

Area:	$area_{tot}$	$\simeq$	$17.48 \cdot n_{in}$	-41.15
Static Power:	P_{static}	$\simeq$	$0.0118 \cdot n_{in}$	-0.1379
Switching Power:	$P_{switching}$	$\simeq$	$0.0085 \cdot n_{in}$	-0.1148
Total Power:	P_{total}	$\simeq$	$0.0202 \cdot n_{in}$	-0.2528
Leakage Power:	P_{leak}	$\simeq$	$0.1010 \cdot n_{in}$	$+0.0568$

Both the power consumed and the area are significant, but not high if compared with the target structures.

However, for what concern the timing, such a trend cannot be found. For this reason, a focused study on this has been brought on. In this case, the number of inputs has been swiped between $25 \div 1000$ and the result is 4.3.

From the picture, it can be noticed as the trend is exponential. In particular, increasing the number of inputs, the delay increases, but then it stays fixed for a



(a) Area

(b) Power

(c) Leakage Power

(d) Time

Figure 4.2: Performances Graphs for Different Size One's Counters

given number of input. This is due to the fact that the number of layers needed is what makes the delay higher and, when the number of inputs becomes significant, increasing it does not increase the layers and then the delays.

Moreover, a comparator with a fixed threshold has to be added. This should add a low level of ports, then it is believed not being a major problem while considering where to insert the pipeline.

4.3.2 Comparator

After the one's counter, also the comparator has been synthesized. This is because, together with that structure, it represents the bottleneck in the total hardware, and then it creates guidelines for the position on where to insert the pipeline.

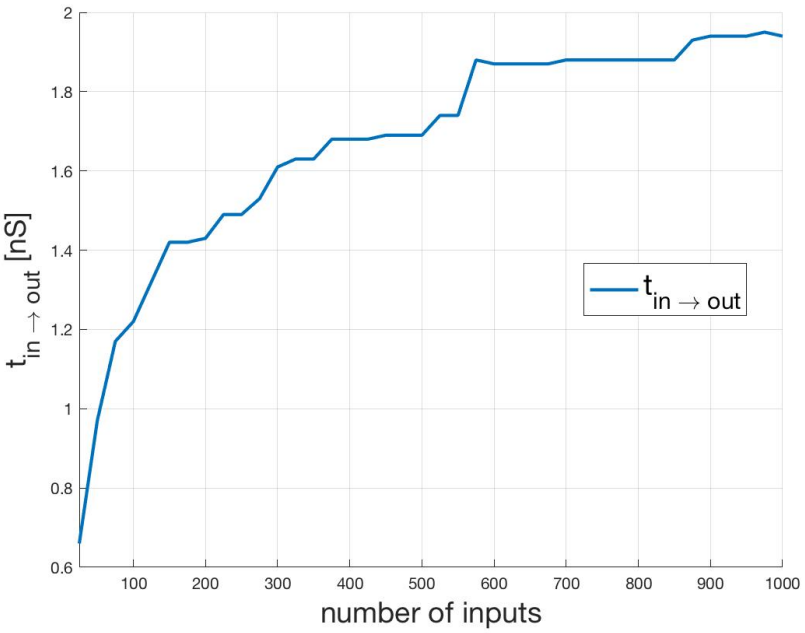


Figure 4.3: Time for One's Counters with $25 \div 1000$ inputs

Differently from before, this comparator is useful to find the index of the highest local field in the output layer. Hence, there will only be one comparator, of a fixed number of output. In this case, the fixed number is 8.

The results are presented in the following table:

Table 4.3: COMPARATOR'S PERFORMANCES

Area	$524.2081\mu m^2$
Time	$1.61ns$
Static Power	$110.95\mu W$
Switching Power	$80.614\mu W$
Total Power	$191.56\mu W$
Leakage Power	$1.969\mu W$

The time is again the performance to focus on. In this case, the delay is similar to the one's counter. Again, to achieve the goal of $1GHz$, it is possible to add registers and pipeline the structure. As shown in 3.10, the structure is regular and easy to pipeline. For how the structure is made, it is possible to add a wall of registers between one of the layers and to find the best tradeoff between the possible choices.

4.3.3 One's Majority

As mentioned before, this is just an example on how to dimension the analog structure for the one's counter. The example will be carried out for the best and worst cases, assuming that, if it is possible to use the structure in both the occasions, then all the middle ground possibilities will be included.

Again, the inputs are contained in *input/Params.pm*. In particular, the following quantities are the ones the user has the freedom to change:

$L_{n_{clk}}$	Length of the NMOS connected to clock
$W_{n_{clk}}$	Width of the NMOS connected to clock
L_{boost}	Length of the PMOS to keep the structure high
W_{boost}	Width of the NMOS to keep the structure high
L_{std}	Length of the NMOS connected to inputs
W_{std}	Width of the NMOS connected to inputs
V_1	Positive voltage
t_clk	Clock period
$size$	Flag: if 1 the inputs are set to half 1 and 0

The last variable is used to dimension the transistors. In fact, while dimensioning, the goal is to make the difference between the condition that should provide a 1 in output and the one that should provide a 0 as high as possible. That point is where the threshold of the output comparator should be. Hence, the created spice netlist will have half inputs fixed to 0, half less one to 1 and one input that switches. This way, the threshold can be find with only one simulation and the output can be observed in the same time.

The normal way of proceeding will then be the following:

1. Size the pull-up transistor as $\frac{n_{in}}{4} \cdot size_{std}$; the division by 4 is heuristic;
2. Move the size of each input transistor to change the impact of each transistor;
3. Size the clock transistor to make the decision boundary fall in the lower-mid part of the output characteristic, where the comparator works better.
4. Tune the threshold.

Moreover, the clock transistor has to make all the current flow through ground. This means that it has to be big enough in order not to be the bottleneck in the speed. In the meantime, it does not have to be huge, to make the input transistor have an impact on the output characteristic. This is where the trade off has to be found for allowing a fast operation as well as a precise one.

Anyway, in the output, a comparator is needed, but the more precise the comparator, the more costly the structure. From the literature, it can be seen that some comparators are able to discern between $10\mu V$, then, the goal of the designer should be to divide the 1 and the 0 situations with at least $1mV$, so that the comparator does not need to be very precise. Moreover, as already presented in the previous chapter, increasing the number of inputs, the situation becomes more critical. To understand this point, it is possible to imagine the whole output dynamic and divide it into a number of levels that is equal to the number of inputs[1]. Increasing the number of inputs, in general, increases the precision of the reference needed.

From the simulations, it will be seen that the theoretical threshold does not work. In fact, the comparator implied in this solution has an offset error that should be found depending on the different configurations. In general, this gives a new parameter to tune.

One Majority with 51 inputs

The sizing and the results for the simplest structure are presented in 4.4 and 4.4.

Table 4.4: ONE MAJORITY COUNTER PERFORMANCES: 51 INPUTS

$L_{n_{clk}}$	$W_{n_{clk}}$	$L_{p_{clk}}$	$W_{p_{clk}}$	L_{pullup}	W_{pullup}	L_{std}	W_{std}	V_{th}
$45nm$	$360nm$	$45nm$	$360nm$	$90nm$	$990nm$	$45nm$	$45nm$	$595mV$

It can be seen as, even if the ideal threshold is at $577mV$, the one that makes the system work is $595mV$.

This way, the behavior is exactly the one required. As stated before, the output has to be sampled on the falling edge of the clock, otherwise the information will be

[1]This is not a precise explanation but just a rule of thumb to understand the issue. In fact, in the output characteristic, all the other transistors (clock, pull-up) enter the game.

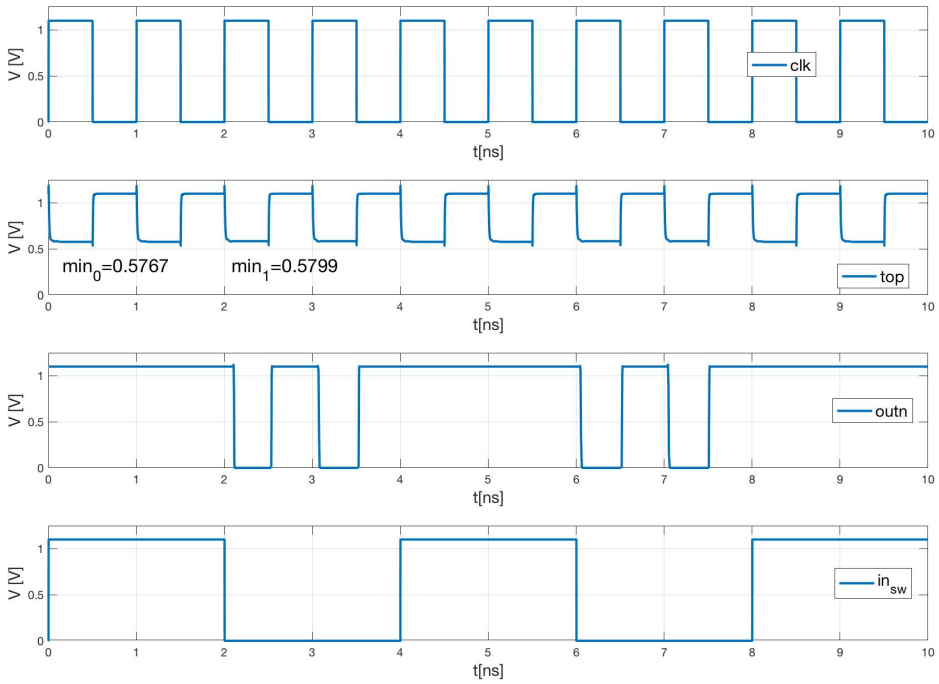


Figure 4.4: Critical waveforms for the One Majority structure with 51 inputs

lost. It can be seen as there is a delay between the switching of the clock and the reaction of the circuit, but it is very low, and the results are ready immediately.

The spice simulation makes the power estimation for the structure possible. In this case, the power is estimated considering the output switching every other clock cycle. The inputs are never switching in this case, but this is the situation for the digital solution as well. The total power is evaluated considering the current flowing in the voltage generator for the whole structure, and then multiplied for the voltage itself. The result in this case is:

$$P_{OM_{51}} = 300.9\mu A \cdot 1.1V = 330\mu W$$

The power consumption is significantly less that the digital structure, and this is due to the low number of transistors used in this hardware with respect to the

digital one. In this structure, the number of transistors is:

$$n_{trans} = 3 + n_{in} + 6 + 28 = 88\ transistors$$

And this is because:

- The whole structure has one nmos and one pmos connected to clock. The pull-up pmos is also there.
- There are n_{in} nmos connected to every input;
- The clock is inverted by means of a chain of three inverters, for a total of 6 inverters;
- The comparator has 28 mos.

The power consumption for the structure is not high and this structure may be actually used in an IC. To further reduce the power consumption of the structure, it is possible to add a signal driving the *Pull-up* mos. This allows to shut the whole structure down while not operating, further reducing its power consumption.

One Majority with 306 inputs

As expected, the same structure with an higher number of inputs result in a lower distance between the threshold. The new parameters in this case are in 4.5 and 4.5.

Table 4.5: ONE MAJORITY COUNTER PERFORMANCES: 306 INPUTS

$L_{n_{clk}}$	$W_{n_{clk}}$	$L_{p_{clk}}$	$W_{p_{clk}}$	L_{pullup}	W_{pullup}	L_{std}	W_{std}	V_{th}
$45nm$	$1.62\mu m$	$45nm$	$360nm$	$90nm$	$4.5\mu m$	$45nm$	$45nm$	$555mV$

Again, the same comments done for the structure with 51 inputs can be done. At this time, the total power consumption is:

$$P_{OM_{306}} = 1.109mA \cdot 1.1V = 1.2mW$$

The total number of transistors is instead:

$$n_{trans} = 306 + 37 = 343\ transistors$$

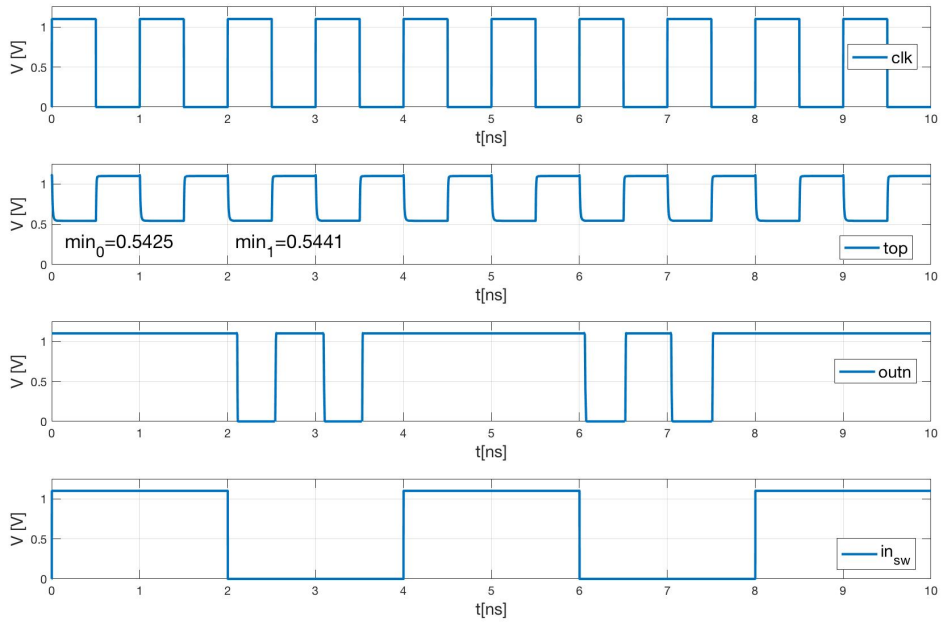


Figure 4.5: Critical waveforms for the One Majority structure with 306 inputs

Again, way lower than the number of transistors used in the digital structure.

This is a limit case, since the division between the two voltages is not marked and increasing the number of transistors would make the decision boundary very small. A more precise comparator would be required in output. Nevertheless, as said before, the structure is able to have noise tolerance if the noise is not too high. This was demonstrated by means of MATLAB simulations that consider the noise. In general, a random noise in the order of ± 5 inputs read wrongly would reduce the scheduling performances less than 0.5%. Hence, this is not believed to be a major problem for the design.

4.3.4 Scheduler without pipeline

Having synthesized all the critical structures, it is now possible to synthesize the whole scheduler without pipeline. The $1GHz$ goal will not be met, but it will give the engineer an idea on how chaining the whole structure deteriorates the performances. The results are presented in the table below.

Table 4.6: PERFORMANCES OF THE SCHEDULER WITHOUT PIPELINE

Area	518264.44 μm^2
Time	5.17 ns
Static Power	382.199 mW
Switching Power	266.843 mW
Total Power	652.199 mW
Leakage Power	3.095 mW

The structure has the expected performances, good in terms of timing, though the area and power performances are important, even if negligible with respect to the ideal target (for example, general purpose computers have power dissipation in the order of tens of watts).

4.3.5 Scheduler with pipeline

It is now possible to insert the pipeline, to achieve the $1GHz$ goal. This is a pretty challenging goal, but thanks to the level of automation of the code, it is possible to achieve it without too much pain. In fact, the *input/Params.pm* contains the following inputs that are used to pipeline:

- *pipe_in*: number of pipeline registers to insert in the One's Counter structures in the hidden neurons;
- *pipe_out*: number of pipeline registers to insert in the One's Counter structures in the output neurons;
- *pipe_comp*: number of pipeline registers to insert in the output comparator;
- *pipe_onecnt_to_cmp*: if 1 a register will be inserted between the output One's Counter and the output comparator.
- *pipe_hl*: if 1, a register will be inserted between z_1 and the input of the output neurons.

The position where the program allows to put the registers is fixed, but it allows enough freedom to the user to achieve the speed he wants. Thanks to this way of

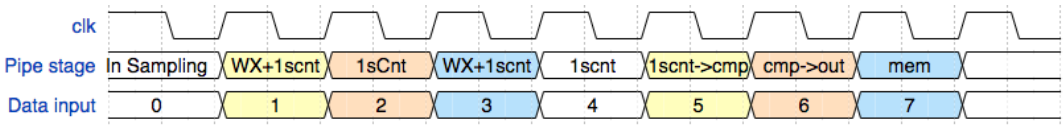


Figure 4.6: Pipeline for achieving the operating frequency of $1GHz$

pipelining, there is technically the possibility of inserting a register every layer of ports, just specifying a high number of pipeline registers.

This inputs are then read from the *run.sh* and *perl/** scripts and inserted directly in the structure.

This way, just changing a value inside an input file, the verilog files are written down again, reflecting the inputs the user wants.

The final structure has then the following characteristics:

pipe_in	1
pipe_out	2
pipe_comp	0
pipe_onecnt_to_cmp	1
pipe_hl	1

Meaning, one barrier of register is inserted between each layer, and inside the counters. The total number of clock cycles the structure would take, provided that the memory access is done is as less as one clock cycle, is 8. A timing diagram of it is presented in 4.6.

The results achieved with this structure are presented in 4.7.

Table 4.7: PERFORMANCES OF THE SCHEDULER WITH PIPELINE

Area	555324.08 μm^2
Time	1 ns
Static Power	426.32 mW
Switching Power	253.84 mW
Total Power	680.16 mW
Leakage Power	3.3756 mW

The area overhead is 7% and the power is 5%, meaning the registers are dramatically improving the speed of the structure, without costing a lot in the total

performances. Moreover, the numbers are still negligible with respect to the rest of the hardware where the accelerator would be inserted.

It can be seen as the performances are remarkable, considering that a new result can be provided every ns, if the pipeline is full. Comparing this number with the reference methods speed and to speed needed to perform the algorithm scheduled, most of the times, this fast operation is not be needed, but, inserting this accelerator in an hardware processor will provide much more computing capability at very high speed. In fact, referring to 1.1, most of the time in the process will be taken in sensing the performances of the single tasks on each execution unit. In this case, the same structure can be used as additional structure for massively parallel and simple operations and it may be implied for completely different purposes, at the cost of adapting the structure to fit other algorithms and adapting the algorithms to this kind of hardware. Moreover, considering the case of multiple cores working in parallel, it is possible to use the same scheduler for each of the core, exploiting in a better way all of its computation capabilities.

Chapter 5

Conclusions

Conclusions and future works will be addressed in this chapter.

The outcome of the project goes beyond the expectation. In fact, both the performances with respect to the HEFT and the overall speed of the system are remarkable. The scheduling can be computed at a very fast pace, and the number of operations to perform is way less than the number of operations needed for the reference algorithm. This has been achieved moving most of the complexity to the software side, but still achieving a good generalization of the structure for different taskgraphs.

Given that, the approach to this problem is completely different, a comparison with previous approaches cannot be done. In fact, the scheduling of real-time systems is mostly based on the availability of the task as well as the resource. In this case, the static algorithm is applied in a dynamic way, resulting in a completely new way of solving the problem.

Moreover, given the high generalization and computational capability, this system may be used both for computing the scheduling the first time, making the scheduling at the compilation time useless and adapting the scheduling to the current conditions of the execution units. This would reduce the amount of work the software should do, allowing it to give the control to the hardware before the scheduling process, leaving both scheduling and dispatch to the hardware.

5.1 Future Work

This work represents a revolutionary way of solving the problem, then a lot of future development can be carried out.

In fact, as stated in the thesis, one of the first easy way to improve the performances of the structure is to have a reference algorithm for the scheduling that works

better than the HEFT. As mentioned in Chapter 1, stochastic scheduling algorithm can be used. This would automatically result in better performances, without any additional work from the hardware/software point of view, other than writing the MATLAB code that describes the scheduling method and without impacting in any way the hardware performances.

Moreover, as highlighted in the statement of the problem, the whole structure has to be made up of a predictor able to sense the status of the machine and predict the number of clock cycles that every task would imply in every execution unit. This has not been implemented yet, but we do not believe this as major issue, since hardware structures that sense the status of the execution units basing on its Instruction per Cycle (IPC) for each application already exist.

Furthermore, to validate the idea in a stronger way, a profiling of the application is needed. This is because the input set (for both training and testing) may not reflect the real behavior of the applications, resulting in different training for the network and highlighting complication or simplification in the structure. However, it is believed that, tuning correctly all the different characteristics, this neural network should have enough degree of freedom to be able to learn with different input set. This kind of work needs to go along with the previous mentioned one. In fact, while profiling the application, it is necessary to understand how those performances change with respect to different parameters of the hosting target.

One other aspect to consider is that, being hardware, the number of input has to be fixed, hence an efficient way of partitioning the network has to be found. This partitioning method should result in the lowest loss in scheduling performances. However, partitioning the taskgraph means to give inputs that are not valid at some time to the network. For example, let us suppose that the user wants to schedule a fifteen tasks taskgraph with a network designed for a 10 tasks taskgraph input. In this case, the user may decide how to divide the original taskgraph, but in any case he has to focus on the missing tasks. He could set the length of those to 0, and make the network learn the more complex taskgraph with some tasks that will take no time. In this case, the designer has to think also to the fact that those tasks do not transfer any data. Nevertheless, all those considerations influence only the training, since during the real-time usage of the accelerator there is no impact in the speed performances.

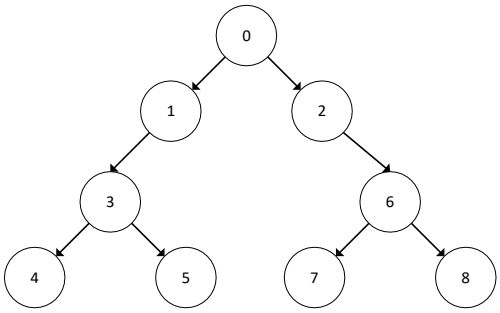


Figure 5.1: Not Partitioned taskgraph.

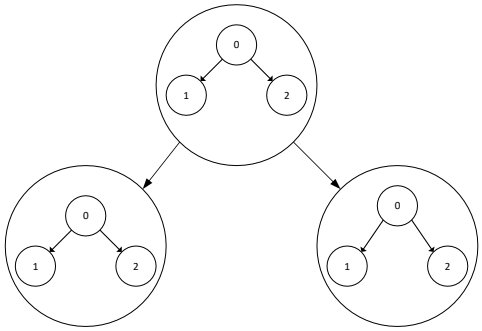


Figure 5.2: Partitioned taskgraph.

One possible partition diagram approach the solution in a hierarchical way, dividing the taskgraphs in sub-taskgraphs and scheduling the high level taskgraphs considering the outcome of the previously scheduled ones. This way, if some common recurrent sub-taskgraphs can be found between all the taskgraphs, the number of stored taskgraphs can decrease. 5.1 presents a possible example of a taskgraph too big to be scheduled. The idea may be the one to divide it into several taskgraphs and then schedule the big taskgraph taking the results of the previous runs on the single taskgraphs as inputs for the last run.

In this example, given that the change in performance between the condition of single task (5.1) and the condition of a group of task (5.2) are not changing in a way the neural network cannot learn the generalized problem, only one taskgraph has to be known, but the whole taskgraph can be scheduled in four iterations inside the accelerator.

One more thing to consider is that the main idea has been to create a scheduler that is able to adapt its outcome during the time the taskgraph is being executed. Thanks to the generality of the structure, this scope has not been the main focus of the work, but it is possible to slightly change the system considering the output at time t as the input at time $t + 1$, creating a recurrent neural network. This may result in simplifications in the structure, since the network may exploit the previous scheduling information as key point for the new scheduling.

This structure goes really fast, hence it really operates for a percentage of the total time. Depending on the structure where this accelerator is inserted, there are different possibilities to increase the percentage of usage:

- The number of cores of the machine is really high: to use only one scheduler for all the cores, increasing the percentage of usage and providing a new schedule every clock cycle;

- The number of cores is not high: this structure, even in its simplest form, has a lot of computational power. It could be used for any other purpose that a computer is used for, as an additional execution unit or adapting any other kind of high computation workload to it. It could also be exploited for hardware security purposes.

Lastly, the integration with the environment represents a core point for the usability of new hardwares. From this point of view a big part of the study still has to be done, even though we do not believe this as a major issue. As shown in 1.1, the inputs to the system would be provided from a new structure. The information about the taskgraph can be provided directly from who is controlling the flow of the machine. Moreover, the creation of a system that writes the running queue to dispatch each task on a resource can be implementable. From this point of view the ways of proceeding may be different:

- In newly designed, special purpose systems, this can be integrated writing the new software. In fact, knowing that from the schedule on, the chain of operations are performed in hardware, the power would be moved to the cpu in one of the previous steps, instead of waiting for the dispatch.

- In already existing designs, a shell that takes the input the software already gives, adapts them and directly gives them to the hardware can be created. This would also take care of moving the result back to the dispatcher in order to use the standard protocol that operating systems use before giving the control to the CPU. If the dispatch instead should be done in hardware, at this point it would be possible to take advantage of pin-mux techniques, useful for the generalization of the problem as well. In fact, having pin-mux in input and in output, interacting with both the hardware and the software, the neural network could be used for scheduling and other purposes. Moreover, a simple multiplexer could be used to decide who is in charge of writing the running

queue for every application and, in that case, be fully compatible with the existing architectures.

Bibliography

[1] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[2] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, Nov 2007.

[3] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 64–75, June 2004.

[4] Jiadong Yang, Hua Xu, Li Pan, Peifa Jia, Fei Long, and Ming Jie. Task scheduling using bayesian optimization algorithm for heterogeneous computing environments. *Appl. Soft Comput.*, 11(4):3297–3310, June 2011.

[5] K. Ramamritham, J. A. Stankovic, and P. F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):184–194, April 1990.

[6] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[7] Alexandra Olteanu and Andreea Marin. Generation and evaluation of scheduling dags: How to provide similar evaluation conditions. *Computer Science Master Research*, 1(1), 2011.

[8] Z. Li, Y. Huang, and W. Lin. Fpga implementation of neuron block for artificial neural network. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2, Oct 2017.

[9] R. Ramanarayanan, S. Mathew, V. Erraguntla, R. Krishnamurthy, and S. Gueron. A 2.1ghz 6.5mw 64-bit unified popcount/bitscan datapath unit for

65nm high-performance microprocessor execution cores. In *21st International Conference on VLSI Design (VLSID 2008)*, pages 273–278, Jan 2008.