23WSD530 CW2: OpenMP

Contents

Background knowledge

What is parallelisation?

Concurrency is a prerequisite for parallelism. A concurrent program is one that has independent tasks that can be in-progress at the same time. For example, task 1 runs for some time, then is pre-empted while task 2 runs. Afterwards, the program can return to task 1 without running into issues. Parallelisation takes a concurrent program, identifies these tasks, and attempts to execute them on different threads. The simplest way to run these threads is by assigning a thread to each core. However, a single core can run multiple threads. The difference between a parallel and a concurrent application is that parallel applications appear to progress multiple tasks in real-time [1] . Parallel applications have the potential for much higher performance.

Traditionally, performance was improved by improving the hardware. If higher performance was needed, a higher frequency processor would be used. However, this changed once constraints on power consumption were identified. To reduce power consumption without sacrificing processing power, multiple lower-frequency cores are used instead of one higher-frequency core. This is because lowering the frequency allows the voltage required to be lowered since switching happens slower. [*equation I*] shows the relationship between capacitance, voltage, and frequency. Therefore, despite higher capacitance resulting from the extra cores, using multiple cores allows the power consumption to be significantly reduced. With power consumption as an important constraint, this makes power efficiency a function of how well the application can utilise different cores.

$$P = CV^2f \qquad [I]$$

OpenMP

In the software stack, OpenMP mediates between the application layer and the operating system layer. It provides a high-level interface between the programmer and Posix threads (pthreads). At a high level, it allows programmers to parallelise their applications by assigning workload to different threads. OpenMP relies on directives to instruct the compiler on how to create and handle the pthreads.

A major challenge to parallelisation is known as the race-condition. This occurs when multiple threads attempt to access the same variable at the same time. For example, thread 1 reads variable x = 1 and so does thread 2. Both threads attempt to increment x, so the correct answer would be x=3. However, because both threads read x=1 they both increment their copy of x by 1 resulting in x=2. To avoid this, the threads need to coordinate. OpenMP provides synchronisation constructs to accomplish this.

The synchronisation constructs are #pragma critical, #pragma barrier, and #pragma atomic. #pragma critical and #pragma atomic are mutual exclusion clauses. Code enclosed within these directives will only be executed by one thread at a time. #pragma atomic uses hardware synchronisation which makes it more efficient for basic operations like ++. This prevents the race-condition. However, it forces that section of the code to be executed sequentially. #pragma barrier is analogous to joining pthreads. All threads must reach the barrier before any thread progresses past it. However, these constructs are computationally expensive. To optimise performance, the application should be split up into tasks that are as independent from each other as possible.

To avoid using synchronisation constructs, it is important to correctly manage the data environment. OpenMP has clauses that can be added to its directives to manage data access. At the start of a parallel region, the compiler needs to know which variables are shared between the threads and which can be copied into the private stack of each thread. By default, all variables are set to shared (this can be changed by the *default()* clause). However, shared variables can only be accessed by one thread at a time and should be reduced as much as possible. The data environment clauses are *shared()*, *private()*, *firstprivate()*, and *lastprivate()*.

Declaring a variable as private is the same as declaring it within the parallel region; each thread gets its own independent, *uninitialised* copy. Firstprivate variables are similar to variables are analogous to passing a variable by value to a function; each thread gets its own independent, *initialised* copy of the variable. The scope of private and firstprivate variables is within the threads – the threads cannot change its value outside of them. Lastprivate is designed to overcome this issue if needed. Lastprivate variables store their value at the last loop iteration to the original variable outside of the threads.

Parallelising programs is often done by parallelising loops. To do this, OpenMP provides a *#pragma for* directive that tells the compiler to split the following loop across threads [2]. There are several methods of splitting the loop. The method used is determined by the scheduler. There are 3 main schedules used by OpenMP: static, dynamic, and guided. The default scheduler is static, which splits up the loops across the threads in *chunk* sizes. This is known as block scheduling. The default *chunk* size is ***ceiling****(num_iterations /num_threads)*. This mode is useful for when the number of iterations is known at compile time. For example, in a loop iterating 1000 times, the default scheduler gives one thread iterations 0-249, then 250-499 to another, and so on.

Dynamic scheduling operates on a first-come-first-served basis [2]. If a thread completes its *chunk* before the others, dynamic scheduling allows it to adopt more work. This is useful when there is work imbalance between the threads. This may be due to the program itself or how background tasks are dealt with by the OS (some background tasks may be overloading a specific core/thread). Guided scheduling is a form of dynamic scheduling that starts with large chunks that progressively get smaller [2]. A disadvantage of dynamic and guided scheduling is that runtime computations are required to split the loop amongst the threads, resulting in a higher overhead than static scheduling.

Finally, OpenMP provides a SIMD clause to allow the user to attempt to vectorise simple operations [3]. It is important to note that some OpenMP directives have an implicit barrier which can be disabled. The exception is #pragma omp parallel which defines a parallel region that cannot be left until all threads are finished.

Theoretical speed-up

Amdahl's law can be used to calculate the maximum theoretical speed-up of an application. This law states that the speed-up [2, p. 546], S, is:

$$S = \frac{1}{(1-\alpha) + \frac{\alpha}{k}}$$

Where:

α = proportion of the program that can be parallelised, and

k= number of threads used

While this may be a good estimate for programs where α is known, it should be expected that overheads will reduce the speed-up. Furthermore, calculating α may be difficult to do. Doing so requires knowledge of exactly where the execution time is spent and how much of the instructions executed at that portion is parallelisable.

Methodology

To begin parallelisation, the algorithms are first analysed for bottlenecks. This was done using Intel vTune's Hotspot analysis. The algorithms were modified to run for 500 times per execution to provide sufficient data for this analysis. Furthermore, it was observed that there was significant variance in the data, likely dependent on background tasks ran by the OS. To mitigate this, a python script was used to run the elongated algorithms a further 100 times and collect execution times. Outliers of this data are then removed using the IQR method as the nature of the distribution is unknown. Furthermore, the python script utilised the *openpyxl* library to automatically write this data to an excel workbook where it is automatically cleaned and analysed. This streamlined the data collection phase of this project.

Once bottlenecks are identified as potentials for parallelisation, the algorithm is then understood and broken down into concurrent tasks. Once the concurrent tasks are identified, the data environment for each task is examined and modified to reduce the need for synchronisation between threads. Where possible, threads are given an independent copy of variables. In some cases, this meant modifying the implementation of the algorithm. In such cases, the modifications are detailed and explained. This usually entailed avoiding pointers to avoid threads attempting to access the same memory address.

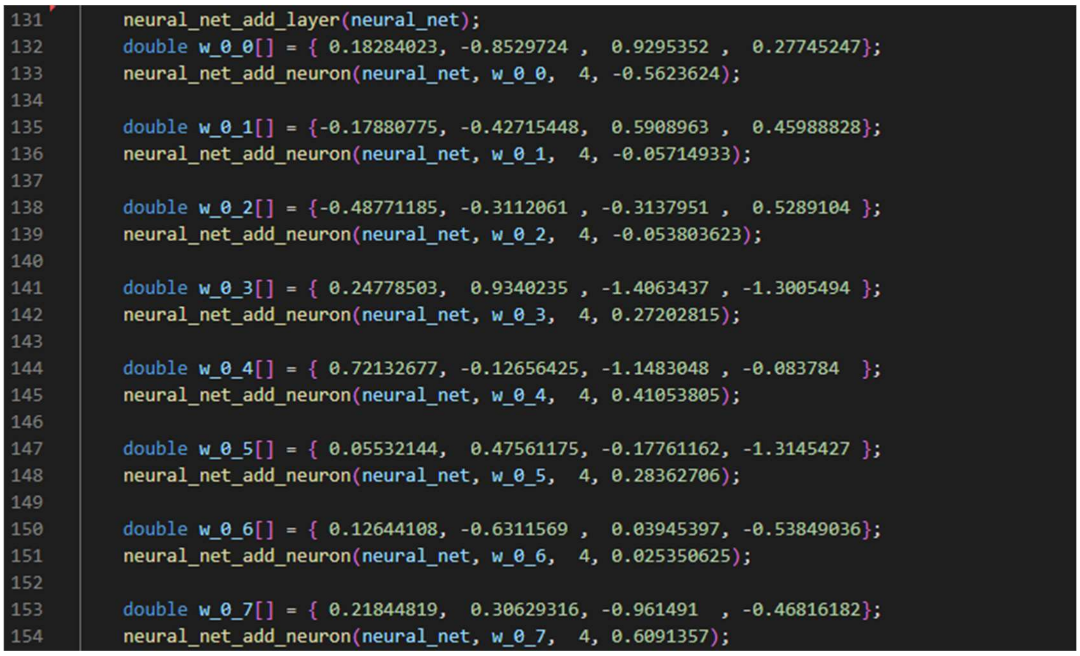
Note: use of Amdahl's law has been avoided due to the difficulty of working out α.

Machine Learning Algorithms

Artificial Neural Network (ANN)

Description

This algorithm starts off with creating a neural network *neural_net*. This network has 1 hidden layer and an output layer [Figure 1]4. Layer 1 has 8 neurons, and the output has 3. The neurons of layer 1 have 4 weights, and a bias, indicating an input size of 4. This is confirmed by examining the *neural_net_run()* function, line 268 [Figure 3], where len = 4. Therefore, each neuron in layer 1 has a set of 5 weights, and layer 2 has 9. This is illustrated in [Figure 2]. The 3 neurons in the output layer correspond to the possible 3 classification categories. The largest of the 3 outputs determines the final classification.



```c
    neural_net_add_layer(neural_net);
    double w_0_0[] = { 0.18284023, -0.8529724 ,  0.9295352 ,  0.27745247};
    neural_net_add_neuron(neural_net, w_0_0,  4, -0.5623624);

    double w_0_1[] = {-0.17880775, -0.42715448,  0.5908963 ,  0.45988828};
    neural_net_add_neuron(neural_net, w_0_1,  4, -0.05714933);

    double w_0_2[] = {-0.48771185, -0.3112061 , -0.3137951 ,  0.5289104 };
    neural_net_add_neuron(neural_net, w_0_2,  4, -0.053803623);

    double w_0_3[] = { 0.24778503,  0.9340235 , -1.4063437 , -1.3005494 };
    neural_net_add_neuron(neural_net, w_0_3,  4, 0.27202815);

    double w_0_4[] = { 0.72132677, -0.12656425, -1.1483048 , -0.083784  };
    neural_net_add_neuron(neural_net, w_0_4,  4, 0.41053805);

    double w_0_5[] = { 0.05532144,  0.47561175, -0.17761162, -1.3145427 };
    neural_net_add_neuron(neural_net, w_0_5,  4, 0.28362706);

    double w_0_6[] = { 0.12644108, -0.6311569 ,  0.03945397, -0.53849036};
    neural_net_add_neuron(neural_net, w_0_6,  4, 0.025350625);

    double w_0_7[] = { 0.21844819,  0.30629316, -0.961491   , -0.46816182};
    neural_net_add_neuron(neural_net, w_0_7,  4, 0.6091357);
```

Figure 1: Creation of the hidden layer [NeuralNet/Base/main.c]

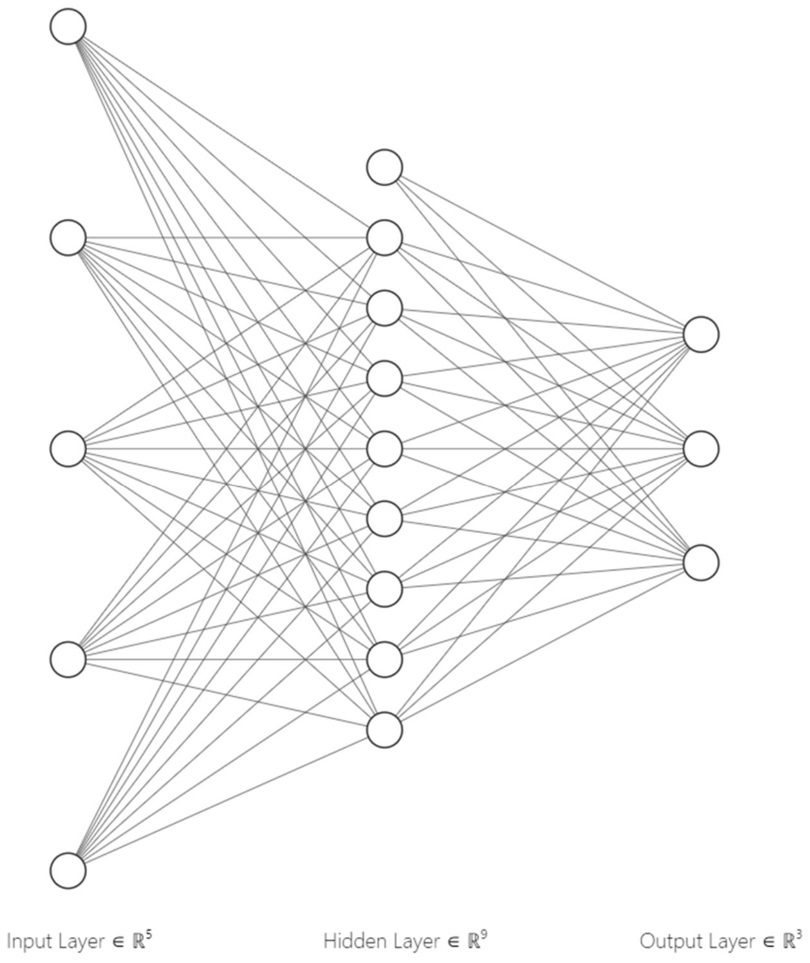
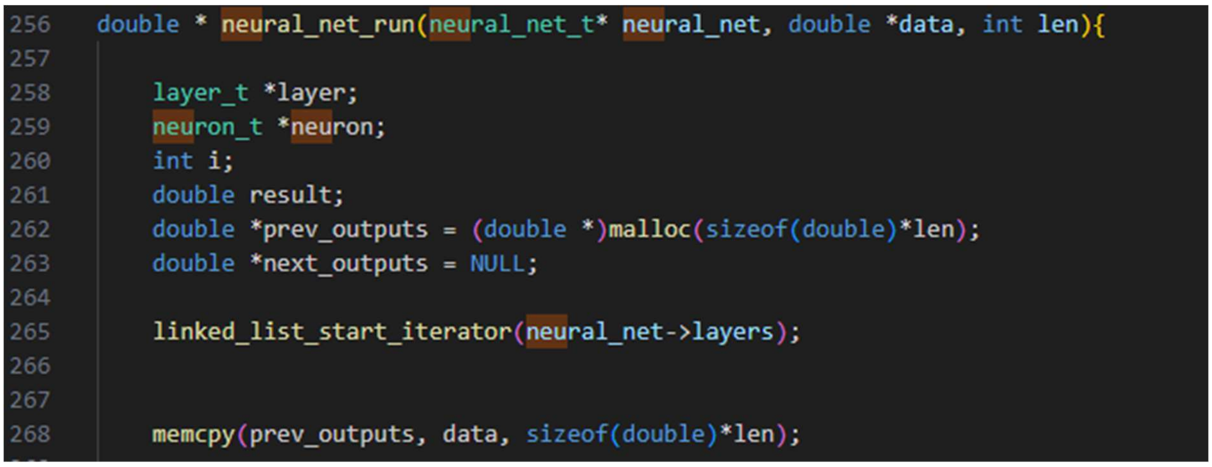


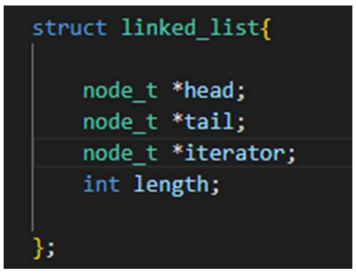
Figure 2: Illustration of the neural network implemented in NeuralNet/base/main.c4



```c
double * neural_net_run(neural_net_t* neural_net, double *data, int len){

    layer_t *layer;
    neuron_t *neuron;
    int i;
    double result;
    double *prev_outputs = (double *)malloc(sizeof(double)*len);
    double *next_outputs = NULL;

    linked_list_start_iterator(neural_net->layers);


    memcpy(prev_outputs, data, sizeof(double)*len);
```

Figure 3: neural_net_run() from NeuralNet/base/main.c

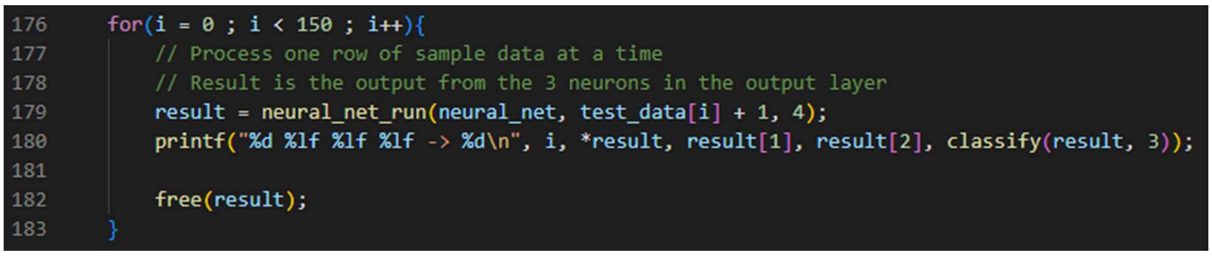
The algorithm is implemented in a linked list format. The neural network is a linked list of layers, and the layers are linked lists of neurons. Doing so allows flexibility in modifying the network if needed. The linked list contains 3 pointers – head, tail, and iterator – and an integer storing its size [Figure 4]. Once the neural network is constructed, the code responsible for running the network is a loop in *main()* that calls *neural_net_run()* [Figure 4].



```
struct linked_list{

    node_t *head;
    node_t *tail;
    node_t *iterator;
    int length;

};
```

Figure 4: NeuralNet linked-list

The neural network is passed to this function along with one row of data at a time and the number of inputs. This function then resets the iterator by pointing it to the head of the list of layers. The same is done for the list of neurons within each layer. Once the iterators are reset, the dot product of the inputs and weights for each neuron is computed and added to the bias. The activation function is then called to compute the final output of the neuron. The iterator for the list of neurons is then incremented. This is repeated for all neurons and then each layer.



```
    for(i = 0 ; i < 150 ; i++){
        // Process one row of sample data at a time
        // Result is the output from the 3 neurons in the output layer
        result = neural_net_run(neural_net, test_data[i] + 1, 4);
        printf("%d %lf %lf %lf -> %d\n", i, *result, result[1], result[2], classify(result, 3));

        free(result);
    }
```

Figure 5: Code to process the samples

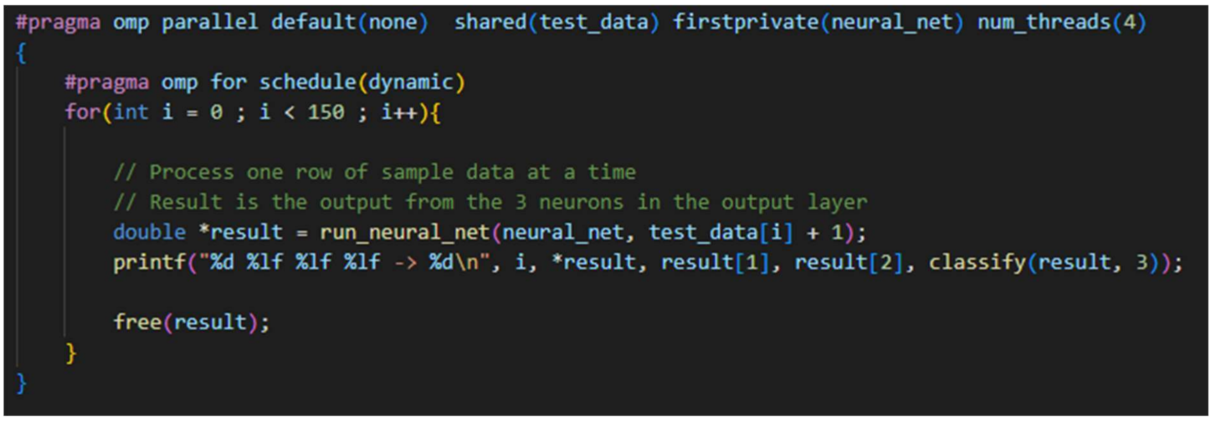
Data environment

Now that the implementation has been discussed, the data environment can be examined. The relevant variables are *neural_net*, *test_data*, and *result*. The test data is constant and remains unchanged and each iteration is independent of the others. Each thread could receive its own copy to avoid the need for synchronisation. The result variable is independent of other iterations and could therefore be made private to each thread. On the other hand, the neural network – despite being fixed once constructed – would be unsuitable for sharing. This is because the network relies on an iterator to store the memory address of the next layer or neuron in the list. If the for loop was to be parallelised, each thread would attempt to access the same iterator. Furthermore, the threads would need to compete to read the value of the neurons. This makes this implementation non-concurrent and unsuitable for parallelism.

Alternative sequential implementation

To overcome this issue, a different implementation of the neural network is required. The method chosen was to use the same approach but without using linked lists. This means that the neural network is a struct that contains 2 layer structs that contains neurons as structs. This implementation defines struct types bespoke to each layer. Bespoke functions were defined to compute the output of each layer. This implementation can be found in *NeuralNet/nonll/ann_nonll.c*.

OpenMP implementation



```c
#pragma omp parallel default(none)  shared(test_data) firstprivate(neural_net) num_threads(4)
{
    #pragma omp for schedule(dynamic)
    for(int i = 0 ; i < 150 ; i++){

        // Process one row of sample data at a time
        // Result is the output from the 3 neurons in the output layer
        double *result = run_neural_net(neural_net, test_data[i] + 1);
        printf("%d %lf %lf %lf -> %d\n", i, *result, result[1], result[2], classify(result, 3));

        free(result);
    }
}
```

Figure 6: OpenMP implementation of the modified NeuralNet

The OpenMP implementation can be seen in [Figure 6]. The #pragma for directive is nested within a #pragma omp parallel directive. The data environment is managed by the *firstprivate()* and *shared()* clause. This gives each thread its own initialised copy of *neural_net* and shares the test data. The number of threads is determined by the - *num_threads()* clause. Finally, the *schedule()* clause defines the scheduler. The number of threads and the scheduler used are varied for comparison [Figure 7].

Verification

The output of part 1 algorithms can be 0, 1, or 2. The sample data is sorted such that samples 0-49 output a 0, 50-99 output a 1, and 100-149 output a 2. The outputs of all applications (base and modified) can be found in *data/ANN/output*. Correct output can be observed for all applications.

Results

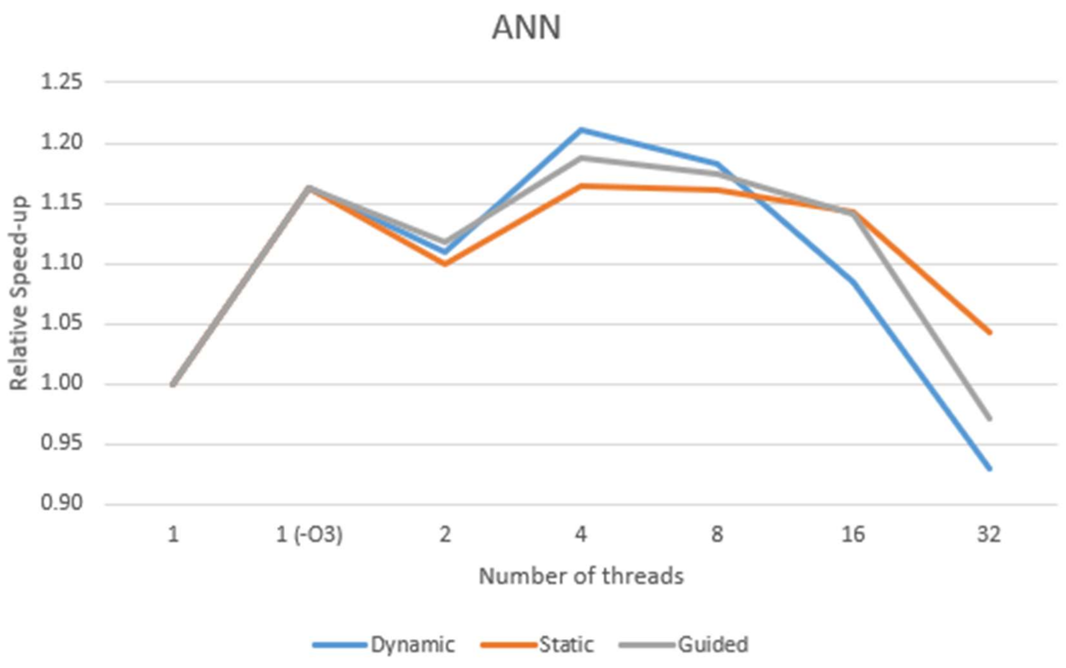


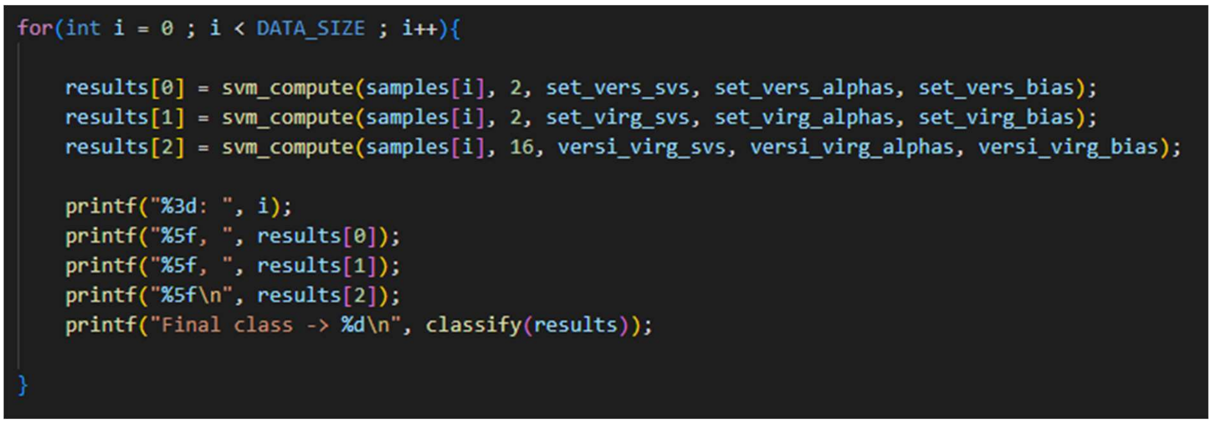
Figure 7: NeuralNet speed-up comparison

[Figure 7] shows the results for this algorithm. It compares the base implementation, its compiler-optimised equivalent (-O3), and parallel versions running from 2-32 threads using dynamic, static, and guided scheduling. The parallel program running 4 threads outperformed all other implementations. Dynamic scheduling was found to be the most effective for 2 and 4 threads. However, it was significantly worse than the alternatives for 8-32 threads. Guided scheduling outperformed static scheduling for threads 2,4, and 8. It was equivalent for 16 threads and significantly worse for 32. Furthermore, it was observed that declaring the test data as *shared* significantly improved the speed-up.

Support Vector Machine (SVM)

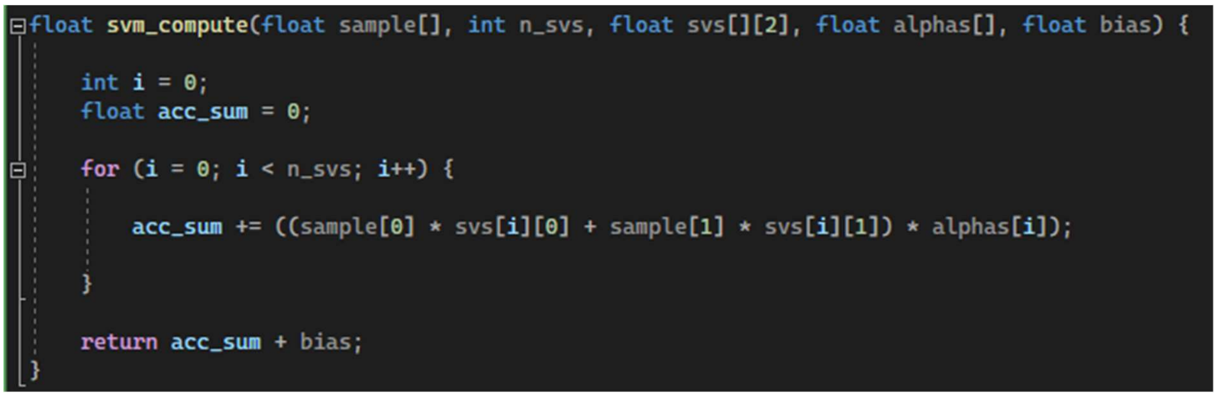
Description

This implementation of SVM takes a row of samples and computes 3 possible classifications using *svm_compute()* function [Figure 8]. *Svm_compute()* is a straightforward function. It performs basic mathematical operations on the sample and accumulates the results. From [Figure 9], the loop ranges from 2 to 16 iterations.



```c
for(int i = 0 ; i < DATA_SIZE ; i++){

    results[0] = svm_compute(samples[i], 2, set_vers_svs, set_vers_alphas, set_vers_bias);
    results[1] = svm_compute(samples[i], 2, set_virg_svs, set_virg_alphas, set_virg_bias);
    results[2] = svm_compute(samples[i], 16, versi_virg_svs, versi_virg_alphas, versi_virg_bias);

    printf("%3d: ", i);
    printf("%5f, ", results[0]);
    printf("%5f, ", results[1]);
    printf("%5f\n", results[2]);
    printf("Final class -> %d\n", classify(results));

}
```

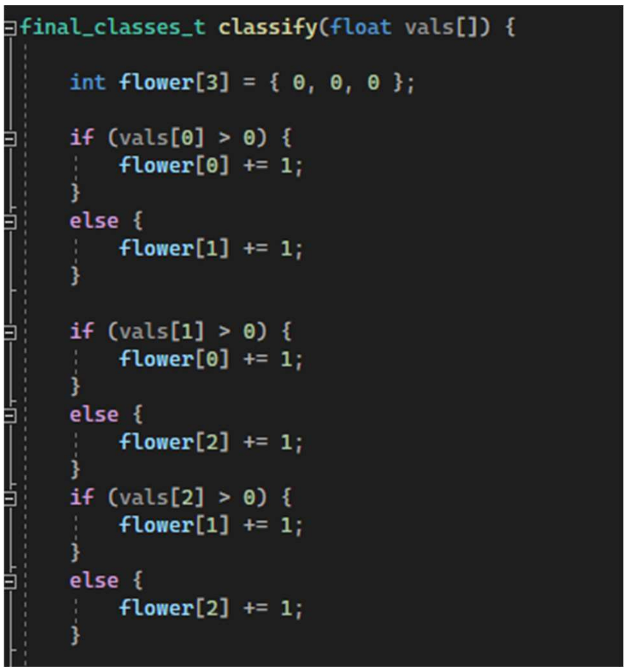
Figure 8: SVM - Processing the samples via loop



```c
float svm_compute(float sample[], int n_svs, float svs[][2], float alphas[], float bias) {

    int i = 0;
    float acc_sum = 0;

    for (i = 0; i < n_svs; i++) {

        acc_sum += ((sample[0] * svs[i][0] + sample[1] * svs[i][1]) * alphas[i]);

    }

    return acc_sum + bias;
}
```

Figure 9: SVM - svm_compute function

The *classify()* function is even simpler [Figure 10]. It takes the *results* variable which represents the SVM computations for each row of samples and determines the final classification. This is known as majority voting.



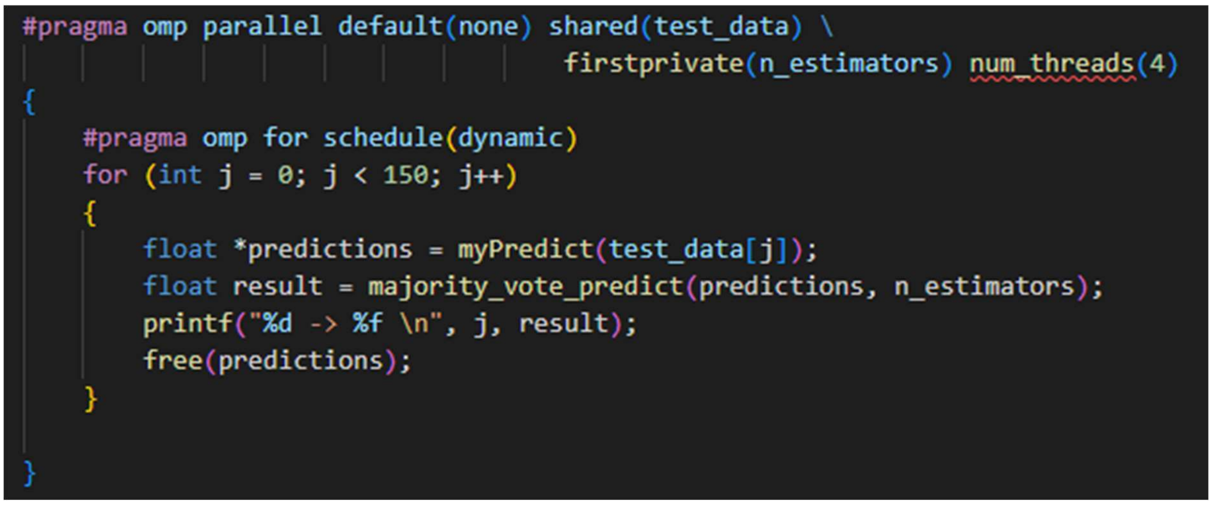
```c
final_classes_t classify(float vals[]) {

    int flower[3] = { 0, 0, 0 };

    if (vals[0] > 0) {
        flower[0] += 1;
    }
    else {
        flower[1] += 1;
    }

    if (vals[1] > 0) {
        flower[0] += 1;
    }
    else {
        flower[2] += 1;
    }
    if (vals[2] > 0) {
        flower[1] += 1;
    }
    else {
        flower[2] += 1;
    }
```

Figure 10: SVM - majority voting function

Data environment

The variables passed to *svm_compute()* are a row of samples, an integer (*n_svs*), an array of svs values, an array of alphas, and a bias. The first key observation is that, except for *printf*, computations within each loop iteration is independent of the others. This means that all variables can be enclosed within a *firstprivate()* clause, giving each thread an independent copy of the variables.

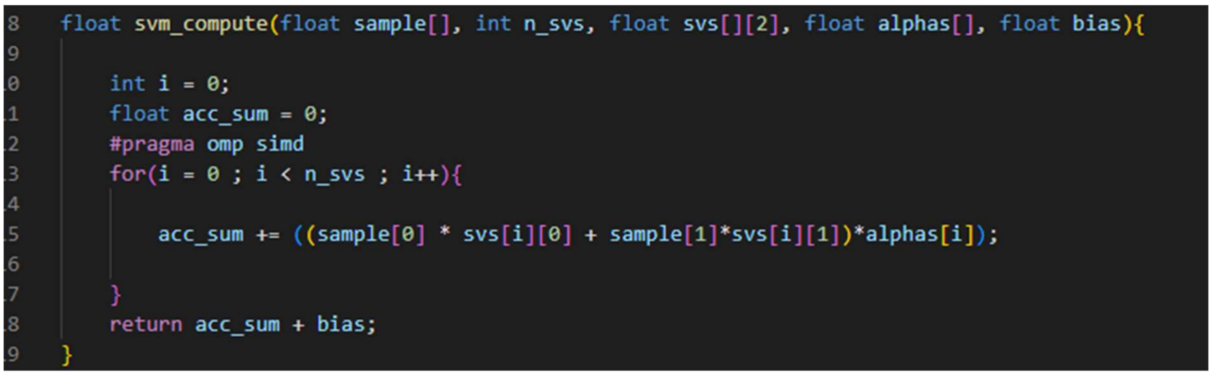
OpenMP implementation



```
#pragma omp parallel default(none) shared(test_data) \
                        firstprivate(n_estimators) num_threads(4)
{
    #pragma omp for schedule(dynamic)
    for (int j = 0; j < 150; j++)
    {
        float *predictions = myPredict(test_data[j]);
        float result = majority_vote_predict(predictions, n_estimators);
        printf("%d -> %f \n", j, result);
        free(predictions);
    }

}
```

Figure 11: OpenMP implementation of SVM

OpenMP was used to parallelise the main loop. As with ANN, performance was found to be better if the test data is shared between the threads. Furthermore, the *simd* clause [Figure 12] was used in *svm_compute* to attempt to speed up the computation where possible.



```
float svm_compute(float sample[], int n_svs, float svs[][2], float alphas[], float bias){

    int i = 0;
    float acc_sum = 0;
    #pragma omp simd
    for(i = 0 ; i < n_svs ; i++){

        acc_sum += ((sample[0] * svs[i][0] + sample[1]*svs[i][1])*alphas[i]);

    }
    return acc_sum + bias;
}
```

Figure 12: SVM - simd clause

It can be seen that the data within the *main()* loop need not be shared. This gives a good opportunity for parallelisation, as each thread would have independent variables.

Results

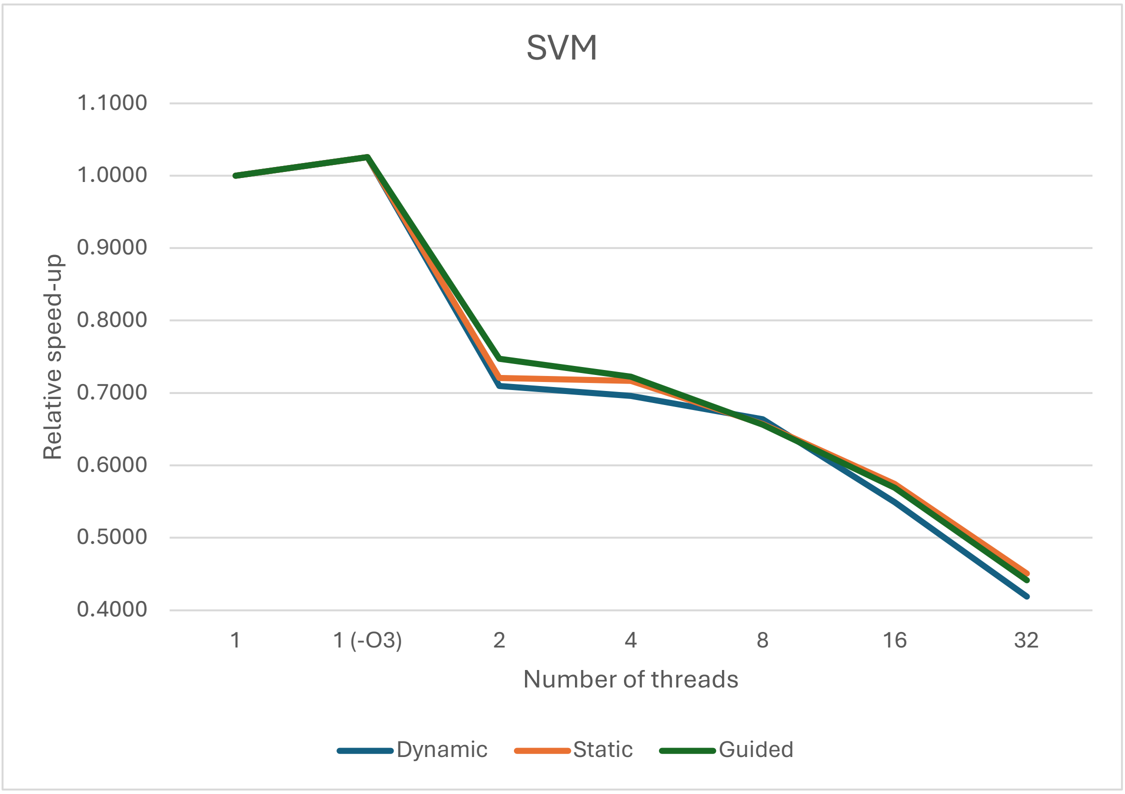


Figure 13: Speed-up comparison of SVM

[Figure 13Figure 7] shows the results for this algorithm. It compares the base implementation, its compiler-optimised equivalent (-O3), and parallel versions running from 2-32 threads using dynamic, static, and guided scheduling. The base algorithm with compiler optimisation flag -O3 performed the best, at 1.025x speed-up. The parallel versions of this program significantly reduced performance. The best performance from a parallel version was achieved by 2-threaded guided scheduling, at 0.75x speed-up. Performance is seen to decrease as the number of threads used increases.

Random Forest (RF)

Description

The Random Forest algorithm is a decision tree-based classifier. It combines the output of multiple decision trees to classify input data. Each decision tree classifies the input based on its logic which is different to the others. The final classification takes the output of all decision trees and determines a final classification mased on a majority voting method. This method takes the most common classification as the most likely and final. This is illustrated in [Figure 14].

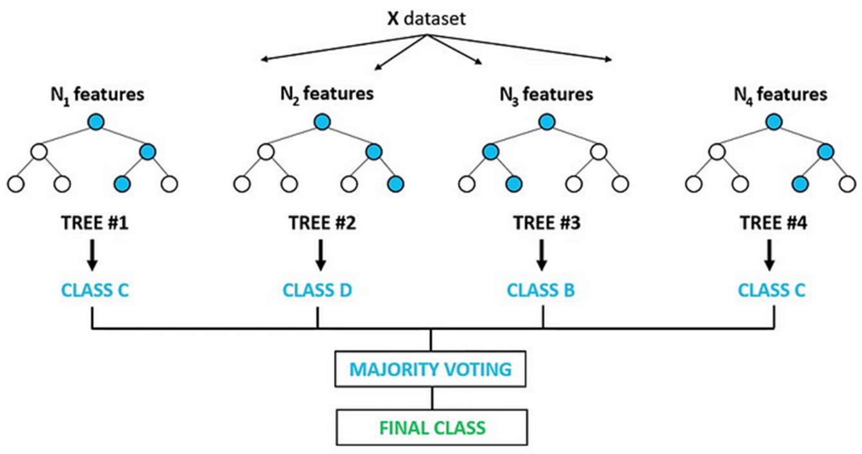
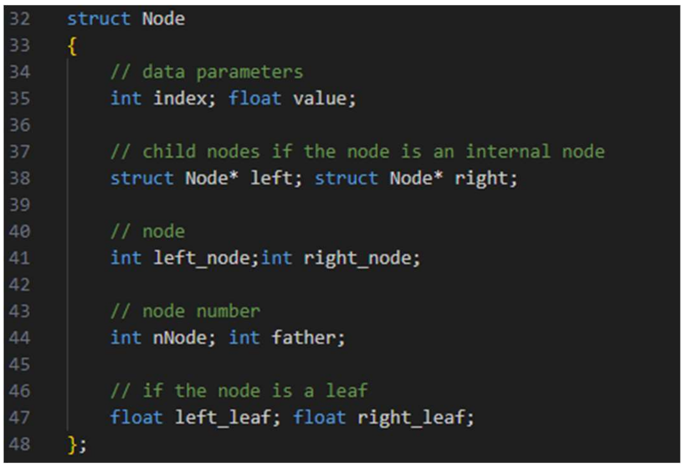


Figure 14: Illustration of a Random Forest, taken from [5]

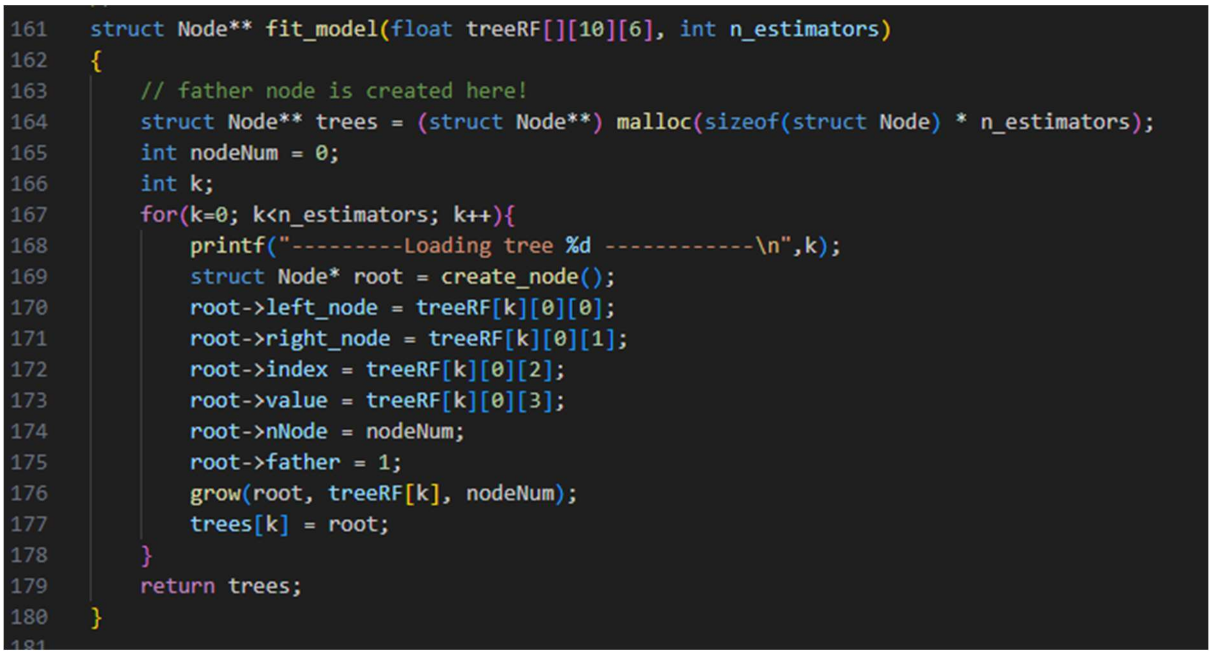
The base implementation of this algorithm implements the Random Forest as a linked list of nodes. Each node has an index, a value, pointers to adjacent nodes, flags determining if nodes exist to the left or right, a node number, and leaf values (if the node has no child nodes). The struct used for this can be seen in [Figure 15]. The forest is built from the first node of each decision tree (root) downwards via the *fit_model()*. The rest of the decision tree is built by recursively calling the *grow()* [**Error! Reference source not found.**] function based on its arguments. It is important to note that the data for



```
struct Node
{
    // data parameters
    int index; float value;

    // child nodes if the node is an internal node
    struct Node* left; struct Node* right;

    // node
    int left_node;int right_node;

    // node number
    int nNode; int father;

    // if the node is a leaf
    float left_leaf; float right_leaf;
};
```

Figure 15: RandomForest - Node struct

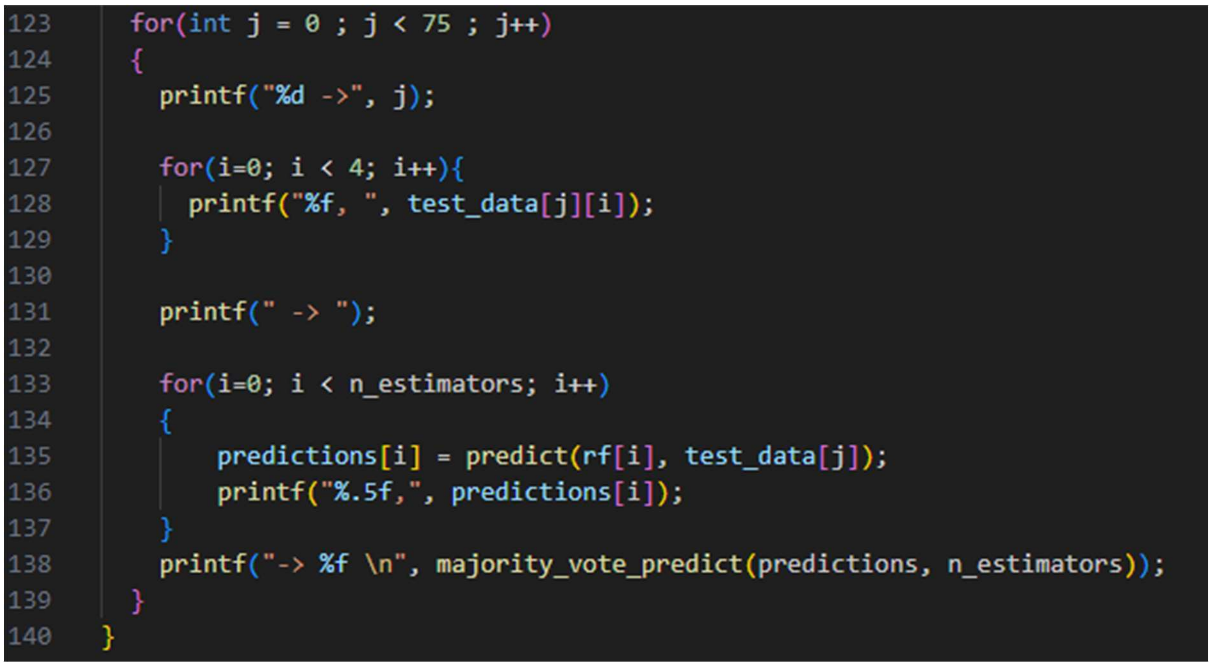
each node are defined in a 3D array *motherFucking3dVec[10][10][6]*. This array also defines parameters that determine the connectivity of each node. These are stored in the inner-most array dimension. For example, the variable at index [0][4][1] determines if the 5th node ([4]) of the first tree ([0]) has a right node ([1]). Consequently, the structure of all 10 decision trees is determined by parsing the values within this 3D array. This array is passed to the *fit_model()* and *grow()* functions as *treeRF*. Once the forest is built, it can be used to classify input data.



```c
struct Node** fit_model(float treeRF[][10][6], int n_estimators)
{
    // father node is created here!
    struct Node** trees = (struct Node**) malloc(sizeof(struct Node) * n_estimators);
    int nodeNum = 0;
    int k;
    for(k=0; k<n_estimators; k++){
        printf("---------Loading tree %d ------------\n",k);
        struct Node* root = create_node();
        root->left_node = treeRF[k][0][0];
        root->right_node = treeRF[k][0][1];
        root->index = treeRF[k][0][2];
        root->value = treeRF[k][0][3];
        root->nNode = nodeNum;
        root->father = 1;
        grow(root, treeRF[k], nodeNum);
        trees[k] = root;
    }
    return trees;
}
```

Figure 16: RandomForest - creating the model

To classify the data, a *predict()* function iteratively passes one row of input data to all 10 trees [Figure 17]. Each node contains an index and a value. The index specifies the index of the input row to be compared to this value. If it is larger, the input data moves down the tree to the node on the right. Otherwise, it moves down to the left. The predictions from each tree is passed onto *majority_vote_predict()* for majority voting. The output is printed to the console via *printf*.



```c
    for(int j = 0 ; j < 75 ; j++)
    {
      printf("%d ->", j);

      for(i=0; i < 4; i++){
        printf("%f, ", test_data[j][i]);
      }

      printf(" -> ");

      for(i=0; i < n_estimators; i++)
      {
          predictions[i] = predict(rf[i], test_data[j]);
          printf("%.5f,", predictions[i]);
      }
      printf("-> %f \n", majority_vote_predict(predictions, n_estimators));
    }
  }
```

Figure 17: RandomForest - processing samples via main loop

Data environment

The relevant variables are *rf, test_data*, *n_estimators*, and *predictions*. *n_estimators* is an unchanged integer declared at the beginning of the program. As with the other algorithms, *test_data* also remains unchanged and computationally loop-independent. This makes *n_estimators* and *test_data* suitable candidates for *firstprivate()*. *predictions* contains the predictions for each row of input data and is created and consumed within the same loop iteration. Since the variable is uninitialised, it is a suitable candidate for *private()*. The Random Forest nodal network is passed as an array of pointers to decision trees. Furthermore, the nodes within the decision trees are connected to adjacent nodes via pointers. This means that all threads would need to access the same memory addresses to access the forest network. This means that *rf* is shared by the threads.

Modifications

As with ANN, overcoming this issue means avoiding sharing pointers between threads. Since the functionality of a decision tree relies on simple conditional statements, they can be implemented by propagating simple *if else* statements. This means that each decision tree can be replaced by a function that follows the same logic of checking a certain input index against a certain value. An example of this can be seen in [Figure 18] for decision tree with index 0. Once all trees have been implemented in such way, a different function is used to call all decision trees and hold their output in an array. A pointer of this array is returned by the function to be stored in *predictions* [Figure 19]. This allowed for an almost like-for-like swapping of *predict()* with *myPredict()*.

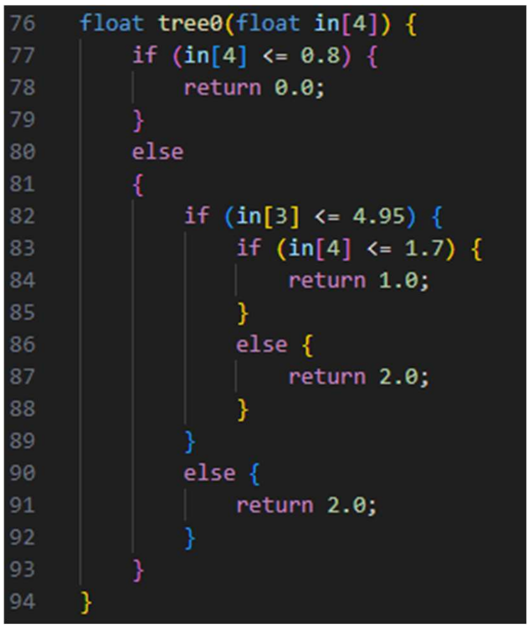
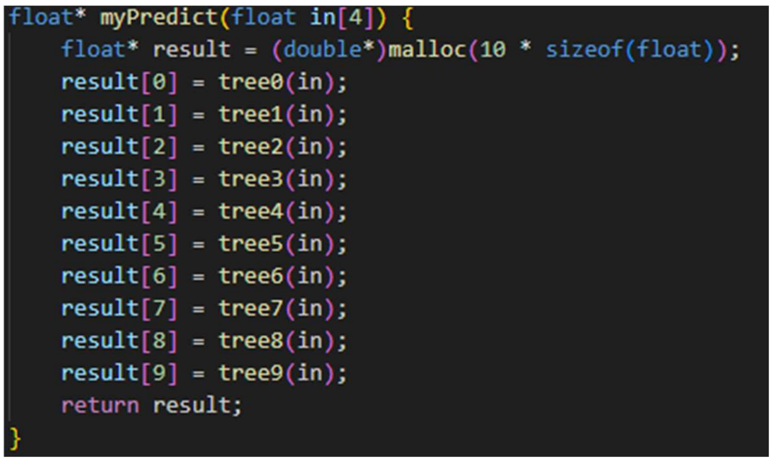

```
float tree0(float in[4]) {
    if (in[4] <= 0.8) {
        return 0.0;
    }
    else
    {
        if (in[3] <= 4.95) {
            if (in[4] <= 1.7) {
                return 1.0;
            }
            else {
                return 2.0;
            }
        }
        else {
            return 2.0;
        }
    }
}
```

Figure 18: Example of decision tree implemented as a function



```c
float* myPredict(float in[4]) {
    float* result = (double*)malloc(10 * sizeof(float));
    result[0] = tree0(in);
    result[1] = tree1(in);
    result[2] = tree2(in);
    result[3] = tree3(in);
    result[4] = tree4(in);
    result[5] = tree5(in);
    result[6] = tree6(in);
    result[7] = tree7(in);
    result[8] = tree8(in);
    result[9] = tree9(in);
    return result;
}
```

Figure 19: Creating a Random Forest using decision trees as functions

Results

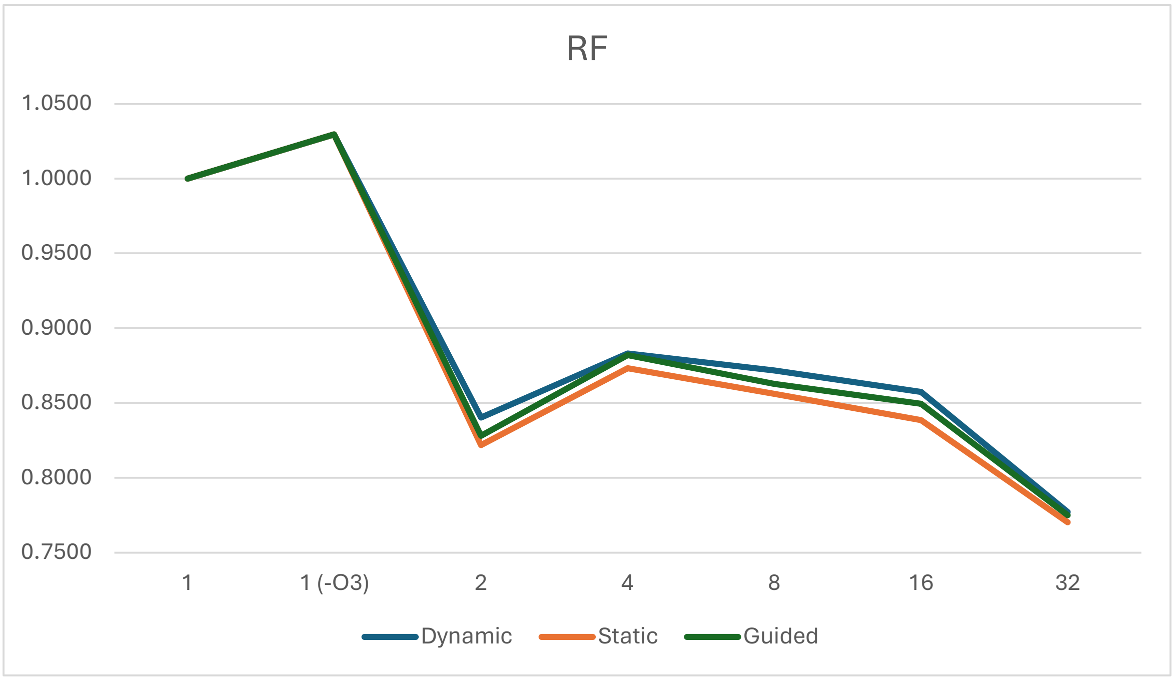


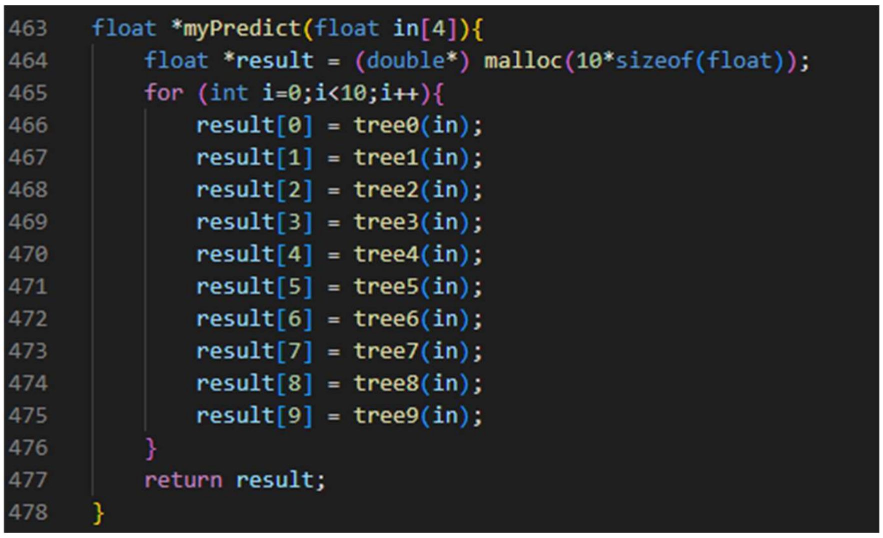
Figure 20: RandomForest speed-up comparison

[Figure 20] shows the results of this algorithm. It compares the base implementation, its compiler-optimised equivalent (-O3), and parallel versions running from 2-32 threads using dynamic, static, and guided scheduling. The base algorithm with compiler optimisation flag -O3 performed the best, at 1.03x speed-up. The parallel versions of this program significantly reduced performance. The best performance from a parallel version was achieved by 4-threaded dynamic scheduling, at 0.88x speed-up. Performance is seen to decrease as the number of threads used increased above 4.

Discussion of MLA results

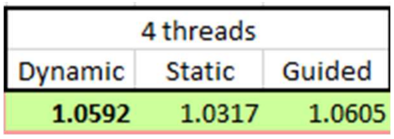
ANN was the only algorithm that achieved a speed-up by parallelising the modified version. The 4-threaded dynamic variant performing the best indicates that the best number of threads is equal to the number of logical cores. The dynamic scheduling suggests that there is imbalance in the workload of the cores. This could be due to the OS, rather than the application (as the workload remains relatively constant between loop iterations). Furthermore, it was seen that using compiler optimisation has improved both sequential and parallel versions of the code (see data/performance.xlsx).

No significant speed-up was observed for RF, despite the alternative implementation. This was hypothesised to be due to printf occupying most of the computational load. To verify this, the printf statements were significantly reduced and the program was re-tested. This showed a speed-up of over 7x, however, the parallel versions still did not outperform the sequential version. To further investigate this, the computational ratio cost between *myPredict()* and printf was modified. The *myPredict()* function was modified to run 10 times per loop iteration [Figure 21]. The result was that 4-threaded guided scheduling improved performance by a marginal 7% [Figure 22]. It is important to note that printf accesses a shared resource that needs synchronisation. This gives it low affinity for parallelisation.



```
float *myPredict(float in[4]){
    float *result = (double*) malloc(10*sizeof(float));
    for (int i=0;i<10;i++){
        result[0] = tree0(in);
        result[1] = tree1(in);
        result[2] = tree2(in);
        result[3] = tree3(in);
        result[4] = tree4(in);
        result[5] = tree5(in);
        result[6] = tree6(in);
        result[7] = tree7(in);
        result[8] = tree8(in);
        result[9] = tree9(in);
    }
    return result;
}
```

Figure 21: RF - myPredict ran 10x per loop iteration



4 threads		
Dynamic	Static	Guided
1.0592	1.0317	1.0605

Figure 22: Speed-up achieved by reducing the ratio of printf to other computations

A similar scenario was seen with SVM. However, SVM's non-printf computations were even less than RF's. It is believed that this is the reason why adding threads reduced performance.

CMSIS-NN

Description

CMSIS-NN is a 3-layered convolutional neural network. It takes an RGB image of dimension 32x32x3 and performs 3 iterations of convolution followed by pooling. The initial convolution layer outputs a 32x32x32 tensor which is reduced to 16x16x32 by the pooling layer. The second and third layers operate similarly. The result is a 4x4x32 tensor which is passed to the activation layer for classification. This is illustrated in []. Objects in the input image are classified in 10 categories: plane, car, bird, cat, deer, dog, frog, horse, ship, and truck. This is used to assess the accuracy of the neural network.

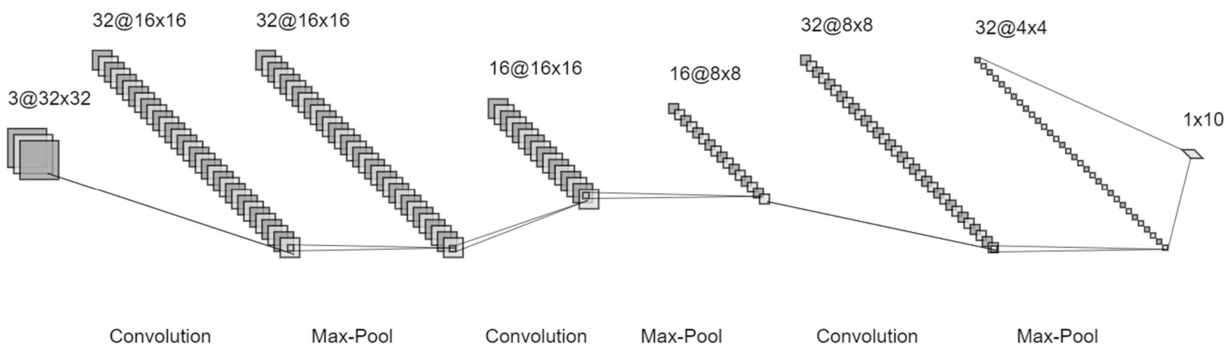
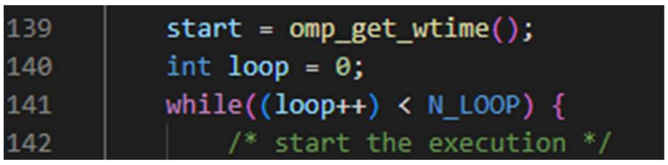


Figure 23: Illustration of CMSIS-NN

Initial performance profile

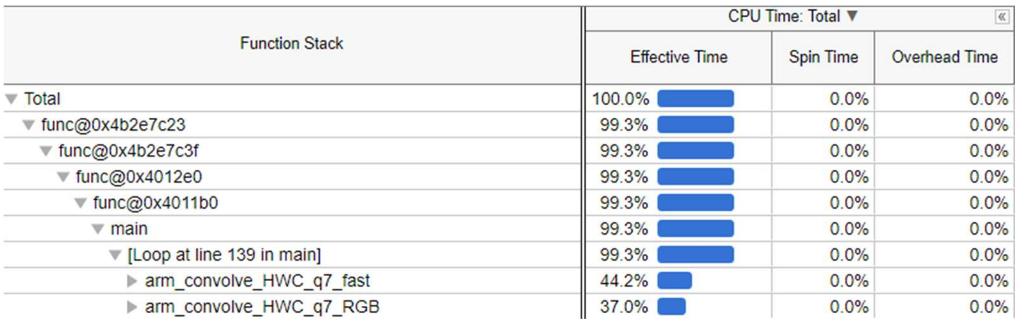
[ref spec] states that 97.7% of the execution time is accounted for by the 3 convolution layers (96.8ms). Unfortunately, this timeframe is too short for Intel vTune to profile the code. To avoid this, the entire program was placed within a loop. The execution time is then divided by *N_LOOP* to obtain the average execution time of the program. For this program, *N_LOOP* was equated to 5000.



```
    start = omp_get_wtime();
    int loop = 0;
    while((loop++) < N_LOOP) {
        /* start the execution */
```

Figure 24: looping the whole program

The output of vTune [ref] shows that 81.2% of the execution time is accounted for by the convolution functions. The difference between [ref] and [ref] is believed to be due to the output shape of the program. [ref spec] outputs a 4x4x64 whereas the example used for [ref] outputs a 4x4x32. The functions *arm_convolve_HWC_q7_RGB()* and *_q7_fast()* perform layer 1 and layers 2&3 respectively. A flowchart of the *_RGB()* variant can be found in [ref appendix]. Finally, it is important to note that the layers have a dependency on the previous layer. This restricts parallelism to the operations within the layers.



Function Stack	CPU Time: Total ▼		
	Effective Time	Spin Time	Overhead Time
▼ Total	100.0%	0.0%	0.0%
▼ func@0x4b2e7c23	99.3%	0.0%	0.0%
▼ func@0x4b2e7c3f	99.3%	0.0%	0.0%
▼ func@0x4012e0	99.3%	0.0%	0.0%
▼ func@0x4011b0	99.3%	0.0%	0.0%
▼ main	99.3%	0.0%	0.0%
▼ [Loop at line 139 in main]	99.3%	0.0%	0.0%
▶ arm_convolve_HWC_q7_fast	44.2%	0.0%	0.0%
▶ arm_convolve_HWC_q7_RGB	37.0%	0.0%	0.0%

Figure 25: CMSIS-NN intel vTune profile

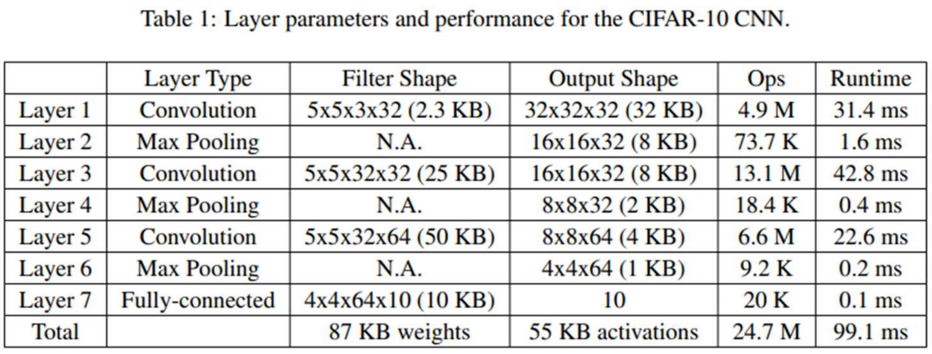


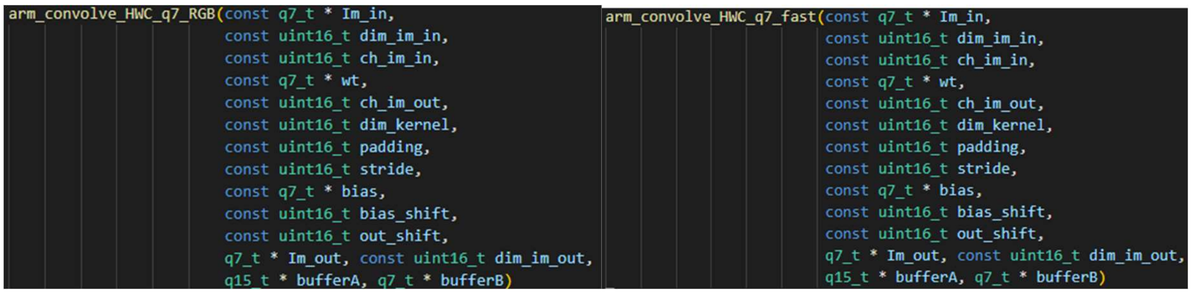
Table 1: Layer parameters and performance for the CIFAR-10 CNN.

	Layer Type	Filter Shape	Output Shape	Ops	Runtime
Layer 1	Convolution	5x5x3x32 (2.3 KB)	32x32x32 (32 KB)	4.9 M	31.4 ms
Layer 2	Max Pooling	N.A.	16x16x32 (8 KB)	73.7 K	1.6 ms
Layer 3	Convolution	5x5x32x32 (25 KB)	16x16x32 (8 KB)	13.1 M	42.8 ms
Layer 4	Max Pooling	N.A.	8x8x32 (2 KB)	18.4 K	0.4 ms
Layer 5	Convolution	5x5x32x64 (50 KB)	8x8x64 (4 KB)	6.6 M	22.6 ms
Layer 6	Max Pooling	N.A.	4x4x64 (1 KB)	9.2 K	0.2 ms
Layer 7	Fully-connected	4x4x64x10 (10 KB)	10	20 K	0.1 ms
Total		87 KB weights	55 KB activations	24.7 M	99.1 ms

Figure 26: From [6]

Data environment:

Both functions have the same prototype [ref]. The code can be found in [appendix].



```
arm_convolve_HWC_q7_RGB(const q7_t * Im_in,
                        const uint16_t dim_im_in,
                        const uint16_t ch_im_in,
                        const q7_t * wt,
                        const uint16_t ch_im_out,
                        const uint16_t dim_kernel,
                        const uint16_t padding,
                        const uint16_t stride,
                        const q7_t * bias,
                        const uint16_t bias_shift,
                        const uint16_t out_shift,
                        q7_t * Im_out, const uint16_t dim_im_out,
                        q15_t * bufferA, q7_t * bufferB)
```

```
arm_convolve_HWC_q7_fast(const q7_t * Im_in,
                         const uint16_t dim_im_in,
                         const uint16_t ch_im_in,
                         const q7_t * wt,
                         const uint16_t ch_im_out,
                         const uint16_t dim_kernel,
                         const uint16_t padding,
                         const uint16_t stride,
                         const q7_t * bias,
                         const uint16_t bias_shift,
                         const uint16_t out_shift,
                         q7_t * Im_out, const uint16_t dim_im_out,
                         q15_t * bufferA, q7_t * bufferB)
```

Figure 27: Convolution function prototypes

`const` variables are unchanged by the function. Furthermore, they are not copied then modified by the loops. This means that they can be cast as *firstprivate*. The *firstprivate()* clause gives each thread a copy of the enclosed variables on its stack. This avoids the need for synchronisation clauses for accessing these variables. On the other hand, the remaining variables *Im_out*, *bufferA,* and *bufferB* are not `const`. *Im_out* is the output variable and is written by the function. Therefore, it is enclosed within the *shared()* clause. The other arguments are unused by the function and are ignored.

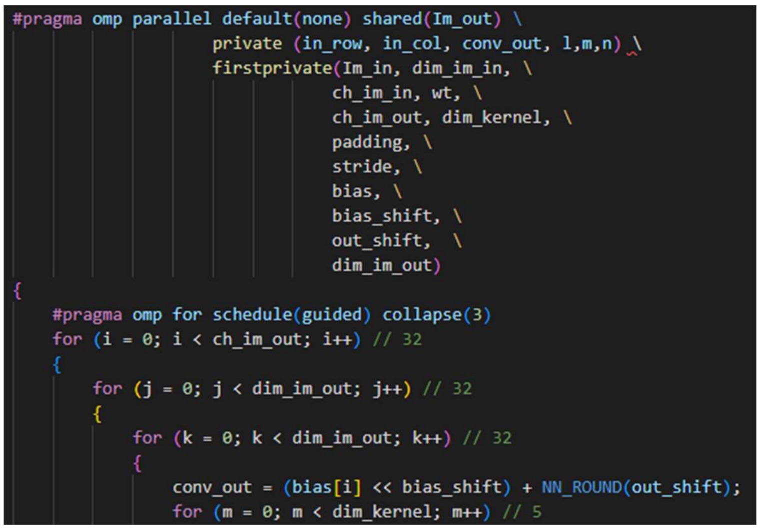
The remaining variables are declared within the function []. The variables in the first row are iterators for the nested loops. Iterators for loops shared by the threads need to be *shared* to allow all threads to increment the loop. The remaining variables are cast as *private* to give the threads separate, independent copies.

```
int  i, j, k, l, m, n;
int        conv_out;
int in_row, in_col;
```

Figure 28: CMSIS-NN conv. func. local variables

OMP features

To parallelise the functions, `#pragma omp parallel` is used followed by a `#pragma omp for collapse(3)` [ref]. The `collapse(3)` clause merges the outer 3 nested loops together. This is the same as enclosing *i,j,* and *k* within *shared()* clause. This was chosen as loops *i* and *j* do not perform any operations independent from loop *k*. Therefore, each thread operates at the level of loop *k*.



```
#pragma omp parallel default(none) shared(Im_out) \
                    private (in_row, in_col, conv_out, l,m,n) \
                    firstprivate(Im_in, dim_im_in, \
                                ch_im_in, wt, \
                                ch_im_out, dim_kernel, \
                                padding, \
                                stride, \
                                bias, \
                                bias_shift, \
                                out_shift,  \
                                dim_im_out)
{
    #pragma omp for schedule(guided) collapse(3)
    for (i = 0; i < ch_im_out; i++) // 32
    {
        for (j = 0; j < dim_im_out; j++) // 32
        {
            for (k = 0; k < dim_im_out; k++) // 32
            {
                conv_out = (bias[i] << bias_shift) + NN_ROUND(out_shift);
                for (m = 0; m < dim_kernel; m++) // 5
```

Figure 29: CMSIS-NN OpenMP implementation

A comparison of schedulers can be found in [ref].

Results

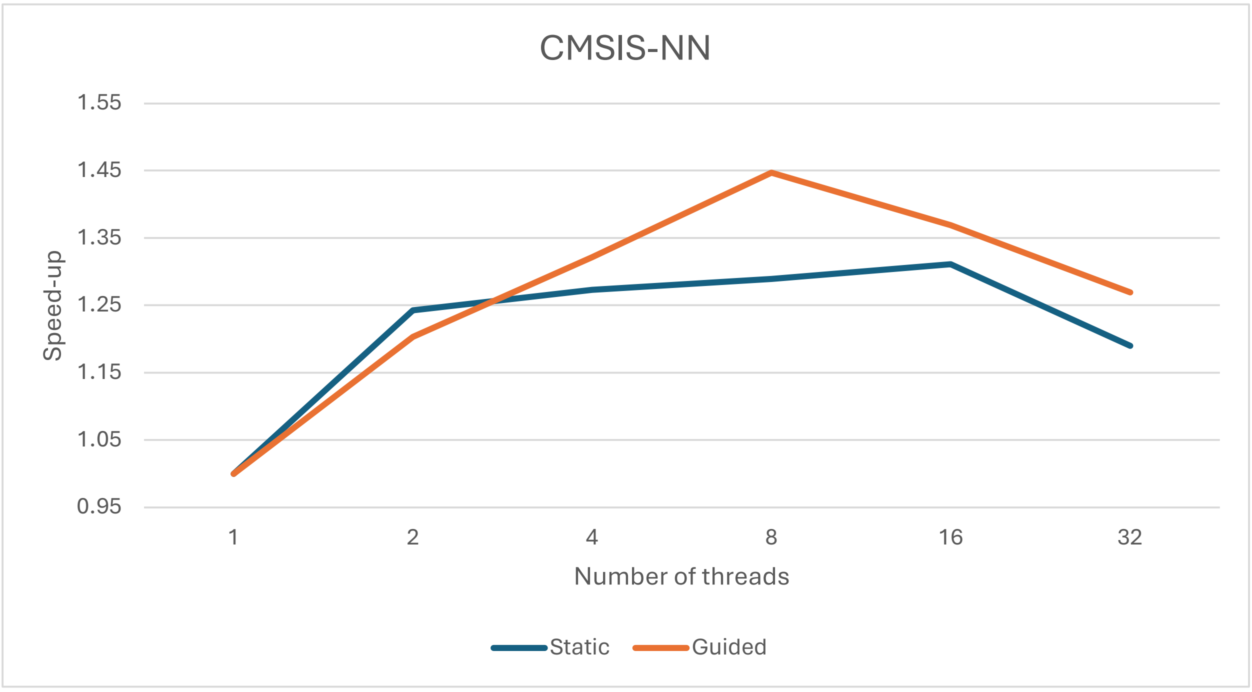
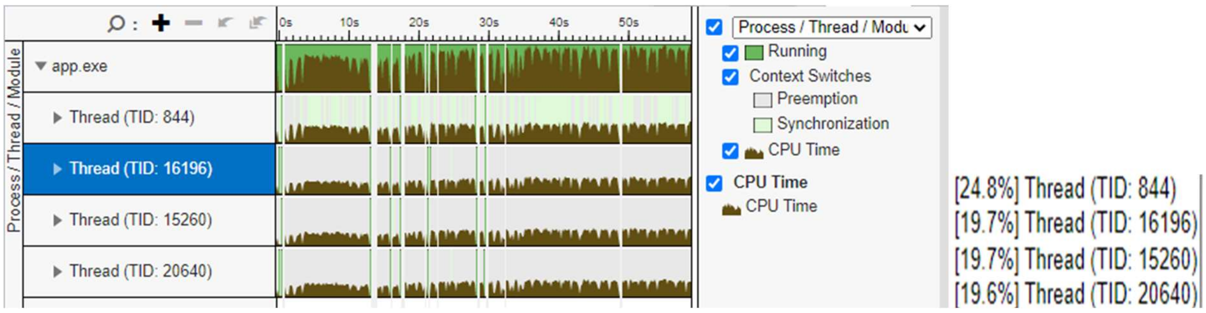


Figure 30: CMSIS-NN speed-up comparison

Significant speed-up was achieved by parallelising the convolution functions. The best performance came from 8-threaded guided scheduling, at 1.45x speed-up. Performance decreased as the threads increased to 16 and 32. Guided scheduling maintained better performance than static for 4-32 threads.

Discussion

As expected, this algorithm had a high affinity for parallelisation. This is because most of the computation happens within the convolution loops, which require a large number of iterations. Intel vTune analysis of the 4-threaded parallel program can be seen in [ref]. This shows that the workload was roughly balance across all threads, with the main thread being slightly higher (due to sequential code). While different compiler optimisation levels were not considered, all results come from the highest level of optimisation. Therefore, it can be concluded that this speed-up should be roughly as high as it can be using the same hardware.



[24.8%] Thread (TID: 844)
[19.7%] Thread (TID: 16196)
[19.7%] Thread (TID: 15260)
[19.6%] Thread (TID: 20640)

Figure 31: CMSIS-NN workload across threads

The optimisations applied to this program were limited to the convolution and pooling layers. This is because they are responsible for over 90% of the computation. Further, marginal improvements may be found by parallelising the final layer, though it would be difficult to detect. Furthermore, data for dynamic scheduling is missing. While it is expected to be similar to guided, this is not known for sure.

Output Verification

The output of the program was verified to be correct. This can be found in data/CMSIS/CMSIS_NN_4.txt.

Conclusion

This report explored the impact of parallelising simple and complex machine learning algorithms. The performance of simple algorithms was difficult to improve via parallelisation. This was mainly due to threads sharing access to printf modules. This was demonstrated by reducing the printf statements in the Random Forest implementation and parallelising it. This achieved a speed-up which supports this claim.

Furthermore, the base implementations of ANN and RF relied on pointers to construct the neural and decision tree networks. This proved to be ineffective for parallelisation as all threads received a copy of pointers to the same memory address. To avoid this, these implementations were modified to avoid the use of linked lists and pointers. The modified ANN algorithm is the same as the base implementation except that the list is static and is contiguous in memory. This allowed for each thread to have its own copy of the entire network.

A similar approach was taken for RF. However, instead of implementing the forest as an array of decision trees made up of nodes, the modifications focused on the functionality of this algorithm. Instead of re-constructing the nodal network using static structures, the decision trees were implemented as a sequence of if-else statements. This resulted in creating a bespoke function for every decision tree. The forest then become a function that called all decision tree-functions and stored their outputs. However, a speed-up was still not achieved for RF. As mentioned above, this was found to be due to the computational load being heavily coming from printf statements, which cannot be fully executed in parallel due to access to the kernel being mutually exclusive.

Somewhat counterintuitively, a better speed-up was achieved by setting the test data to shared. This is believed to be because the data is a 2D array, which means that pointers to 1D arrays are used. While the output of the program is correct regardless, the performance reduction is believed to be from the way the compiler splits up the loop. The performance reduction is believed to stem from cache-misses. Cache misses occur when the copy of a variable in the cache is changed in its main memory address – or in a different cache. This causes the processor to re-fetch the variable from memory. This is more likely to occur if different threads are operating on similar indices of the test data. It is believed that setting the test data to shared tells the compiler to keep the iterations separate. This avoids different threads copying test data used by other threads to their cache and invalidating the cache line.

On the other hand, the more complex algorithm CMSIS-NN benefited greatly from parallelisation. This is because most of the computation was found within the convolution loops. Parallelising this algorithm was straightforward and resulted in a 1.45x speed-up at 8 threads. Furthermore, this algorithm had minimal printf statements.

In conclusion, this report shows that speed-up cannot be simply obtained by parallelising programs. The algorithm parallelised needs to be considered and the data environment needs to be properly managed to avoid race conditions and cache-line misses. Furthermore, this report highlights major bottlenecks caused by excessive use of printf statements.

References

[1] OpenCSF, "Parallelism vs. Concurrency," in *Computer Systems Fundamentals*.

[2] IBM, "#pragma omp for," [Online]. Available: https://www.ibm.com/docs/it/xl-c-and-cpp-linux/16.1.0?topic=parallelization-pragma-omp. [Accessed January 2024].

[3] IBM, "#pragma omp simd," [Online]. Available: https://www.ibm.com/docs/it/xl-c-and-cpp-linux/16.1.0?topic=pdop-pragma-omp-simd. [Accessed January 2024].

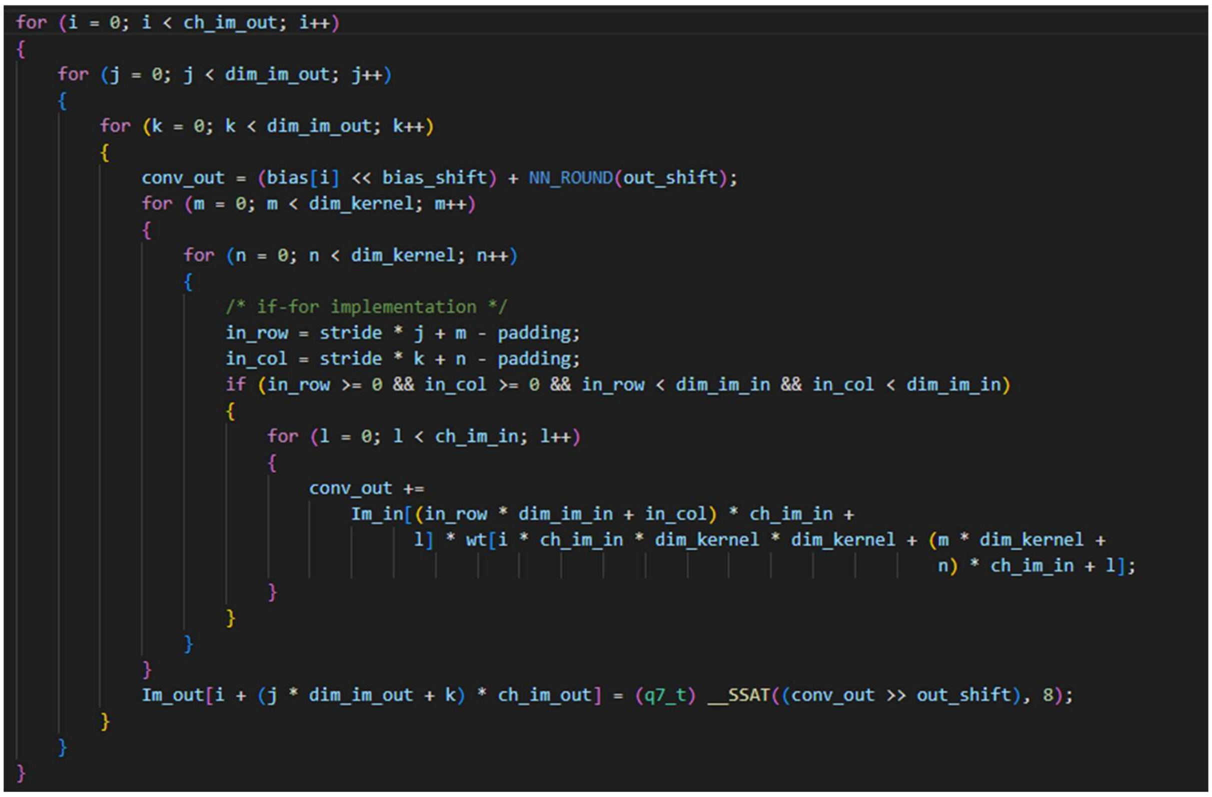
[4] R. Bryant, Computer Systems: A Programmer's Perspective, Third Edition ed., Pearson, 2015.

[5] Medium, "Introduction Random Forest Classification By Example," [Online]. Available: https://medium.com/@mrmaster907/introduction-random-forest-classification-by-example-6983d95c7b91. [Accessed January 2024].

[6] L. Lai, N. Suda and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *arXiv.*

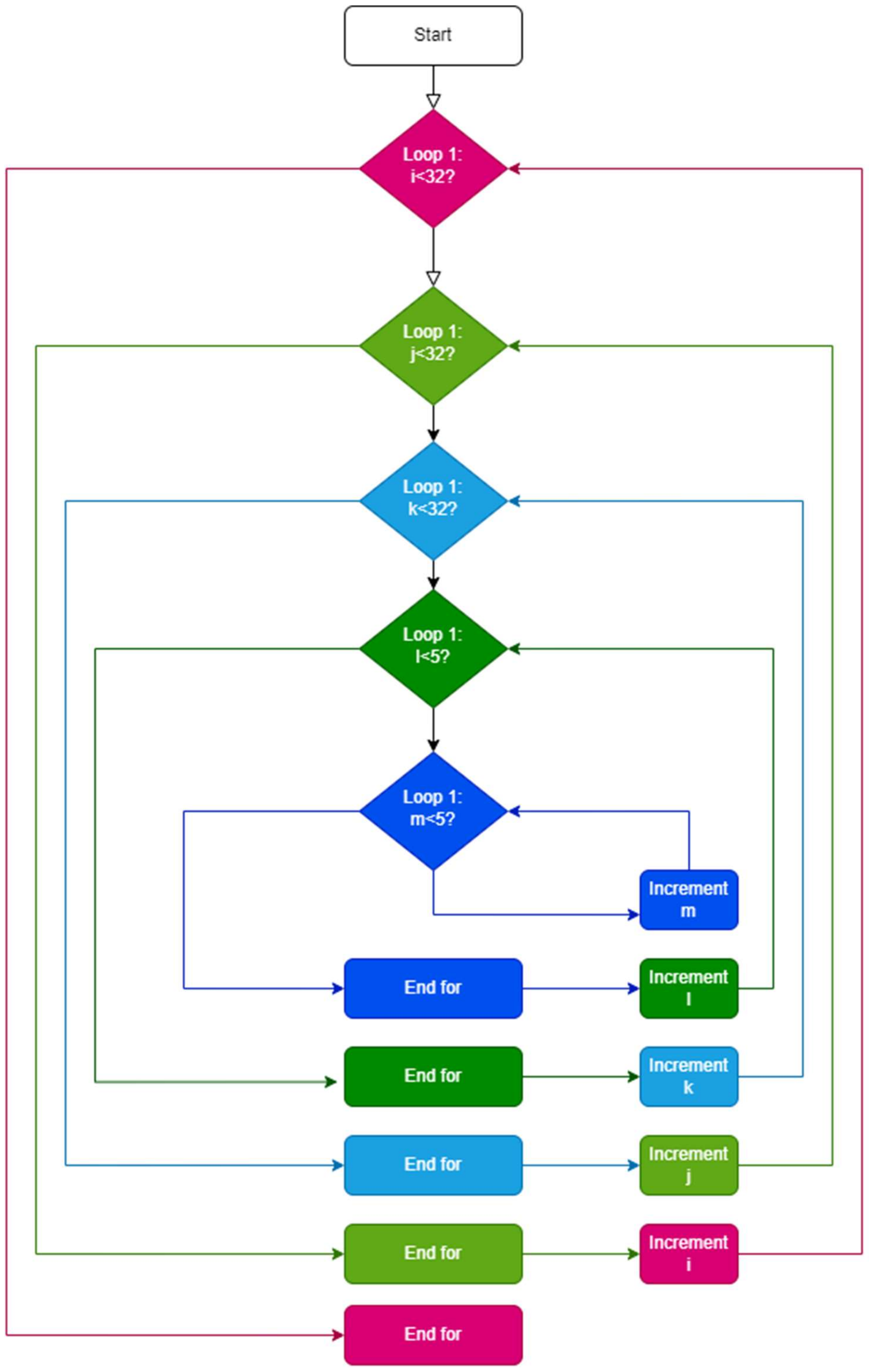
Appendix

Appendix A – *arm_convolve_HWC_q7_RGB()* and *_fast()*



```
for (i = 0; i < ch_im_out; i++)
{
    for (j = 0; j < dim_im_out; j++)
    {
        for (k = 0; k < dim_im_out; k++)
        {
            conv_out = (bias[i] << bias_shift) + NN_ROUND(out_shift);
            for (m = 0; m < dim_kernel; m++)
            {
                for (n = 0; n < dim_kernel; n++)
                {
                    /* if-for implementation */
                    in_row = stride * j + m - padding;
                    in_col = stride * k + n - padding;
                    if (in_row >= 0 && in_col >= 0 && in_row < dim_im_in && in_col < dim_im_in)
                    {
                        for (l = 0; l < ch_im_in; l++)
                        {
                            conv_out +=
                                Im_in[(in_row * dim_im_in + in_col) * ch_im_in +
                                      l] * wt[i * ch_im_in * dim_kernel * dim_kernel + (m * dim_kernel +
                                                                                        n) * ch_im_in + l];
                        }
                    }
                }
            }
            Im_out[i + (j * dim_im_out + k) * ch_im_out] = (q7_t) __SSAT((conv_out >> out_shift), 8);
        }
    }
}
```

Appendix B – flow chart of *arm_convolve_HWC_q7_RGB()*


Start
Loop 1:
i<32?
Loop 1:
j<32?
Loop 1:
k<32?
Loop 1:
l<5?
Loop 1:
m<5?
Increment
m
End for
Increment
l
End for
Increment
k
End for
Increment
j
End for
Increment
i
End for