



Analysing The Performance Impact of Parallelisation on ANN, SVM, RF and CNN Learning Models

Author: A.Varghese
B929164

ABSTRACT: This paper looked at increasing the performance of machine learning models by leveraging multiple cores to execute code in parallel. It detailed how each program was parallelised and the performance impact gained or lost, giving reasons for the increase or decrease. Intel vTune was used to profile each of the learning models to list the bottlenecks in the code and identify potential areas of parallelisation. OpenMP was used to execute segments of each program on threads resulting in a parallel version. For the NN and SVM a fast and slow parallel version was created to highlight the issues with inefficient parallelisation. Overall, it was found that single threaded performance, parallelisation code efficiency, amount of data to be executed in parallel and the number of threads available were the major factors in determining the speed up that could be gained from parallelisation. Intel vTune was effective at identifying program hotspots however, each hotspot may not be an ideal area for parallelisation and so it should be used in conjunction with a programmer's intuition and knowledge to create efficient parallel programs.

I. INTRODUCTION

On June 21st 1948 at 11 am, at the University of Manchester, on the Manchester Baby computer; the world's first electronically stored program was executed [1]. Written by Tom Kilburn it calculated the highest integer factor of 262,144 by starting with a trial divisor and performing repeated subtraction. It consisted of 17 assembly instructions and took 52 minutes to get the final answer of 131,072 [2]. Sixty-five years later Google released a tribute to the Manchester Baby describing the work as the 'birth of software' [3].

Since then, computer adoption has exploded with 47% of all households worldwide having a computer at home, going as high as 92% in countries such as the United Kingdom [4]. With this increased adoption, a wide selection of programs have been created, with estimates suggesting that 2.8 trillion lines of code has been written in the last 20 years alone [5].

Today there are programs that can be run on consumer hardware that use machine learning to mimic human tasks and behaviours. However, these programs are heavily demanding and to keep up with the demand, the silicon industry has created faster and faster processors to execute these programs.

In the past microprocessor designers shrunk the transistor size and squeezed more on per area of silicon. This allowed the processor to complete more operations at once, for example fetching the next instruction from memory while the current one was being executed. Each year, manufacturers increased the transistor count and the operating frequency giving rise to one of the guiding principles of computer architecture, Moore's Law. In 1965 Gordon Moore stated that the number of transistors on a chip will roughly double each year (he later refined this, in 1975, to every two years). What is often quoted as Moore's Law is Dave House's revision that computer performance will double every 18 months [6].

However, this shrinking and increase of transistors could not last forever and eventually the designers ran into the power wall, a limit on the amount of power a microprocessor chip could reasonably dissipate [7]. Nevertheless, the demand from software continued and chip designers needed a new strategy to ensure applications are executed within a reasonable amount of time.

Chip designers innovated by creating multi-core processors. By doubling the number of processing cores on a chip, one can maintain lower power while doubling the speed of many applications. In a sense, the core has become the new transistor with manufacturers adding increasing amounts of cores to processors.

Today a consumer can purchase a CPU such as an AMD Threadripper 5995WX with 64 cores and 128 threads. Although the number of cores per CPU has increased significantly their frequency has stagnated with the Threadripper having a base clock of 2.7 Ghz which is slower than the Intel Pentium 4 a single core CPU with a base clock of 3 Ghz released in 2002, twenty years ago.

Despite the year on year increase of cores, the vast majority of software written and used, only utilises one core. This leaves performance on the table that could be used to decrease execution time and handle tasks simultaneously. The benefits can easily be seen in machine learning applications which requires large amounts of data to be computed. If these applications utilised multiple cores to split their workload it could result in significantly improved efficiency.

This paper aims to investigate the performance impact that could be observed by taking four serial neural network models and parallelising them to utilise all available CPU cores a system has. How each model was analysed to identify areas for parallelisation and its implementation using OpenMP will be detailed throughout.

II. LEARNING MODELS

A neural network is a series of algorithms that endeavours to recognize relationships in a set of data [8]. Networks are typically made up of several layers with multiple nodes in each layer. A node is an area where computation happens, typically some type of weighted multiplication. The exact value of the weights is predetermined by a trial and error learning process on a labelled dataset.

They are able to identify and recognize patterns to help computers make intelligent decisions with limited human assistance. These networks are used everywhere from analysing real-time sales data to facial recognition [9].

There are many algorithms that allow for a computer to make these types of decision; this paper looks at the following four models:

- Artificial Neural Network (ANN)
- Support Vector Machine (SVM)
- Random Forest (RF)
- Convolutional Neural Network (CNN)

ARTIFICIAL NEURAL NETWORK

ANNs were the first learning models that drove machine learning research based on the model of a biological neuron.

It began with work done by McCulloch and Pitts who, in 1943, modelled a neuron as a switch that takes inputs from other neurons and depending on the total weighted input, is either activated or not [10]. The weight corresponds to the strength of a synapse, the neural contacts between nerve cells. These weights can be both positive (excitatory) and negative (inhibitory) [11].

Using this research, in 1958 Rosenblatt created the first supervised learning algorithm of binary classifiers named the perceptron [12]. As software was still in its infancy the Mark I Perceptron machine was realised fully in hardware using potentiometers and circuitry to adjust the weights.

As computers and software developed, today huge ANN can be programmed and executed in memory. These networks are not limited by physical circuit limitations and so can span hundreds of neurons across multiple layers allowing them to classify many patterns with greater accuracy.

WORKING PRINCIPLE

Figure 1 demonstrates the principle of an ANN that can identify shapes to be either a square, triangle or a circle.

The image's pixels are fed into a set of input neurons where a weighted multiplication is computed and sent to hidden layers represented by the lines connecting the input layer to the hidden layer.

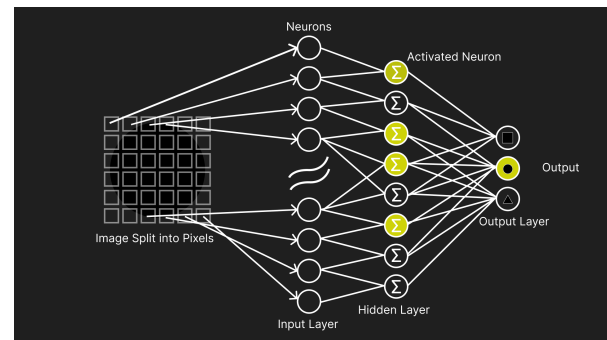


Figure 1: ANN Example (click on image to view larger)

Inside the hidden layers the sum of the weighted multiplication is calculated and depending on whether the threshold is met or not the neurons in the hidden layers are either activated or disabled.

The output from the activated neurons in the hidden layer are then passed to other layers or directly to the output layer depending on the complexity of the network.

The output layer computes the same calculation as the hidden layer, sums up the outputs from the previous layer. However, instead of passing the data forward, the neuron with the largest sum is the classification end result.

The exact weights are pre-calculated through training against a labelled dataset using forward and backward propagation. The predicted output is compared against the expected output and the system attempts to minimise the error difference between the two. There are many different optimisation algorithms that attempt to achieve this in the most efficient way [13].

SUPPORT VECTOR MACHINE

Support vector machines are learning machines for two-group classification problems [14] developed at Bell Laboratories by Corinna Cortes and Vladimir Vapnik in 1993. Similar to ANNs, the SVM is also used to classify groups by recognising patterns.

However, the SVM can only differentiate between two classifications unlike ANNs that can classify as many inputs as it was trained on. Despite this, due to the less computational method that SVM uses, it can make predictions faster than an equivalent ANN.

WORKING PRINCIPLE

The SVM uses regression analysis in n-dimensional space to classify new items into two groups. Here n refers to the number of features that the SVM uses to classify. It achieves this by finding the hyperplane between the two groups. Much like the ANNs the SVM must first be trained using a labelled dataset so that it can create the hyperplane. Once this is done, any new data will either fall on either side of the plane which will correspond to the classification result.

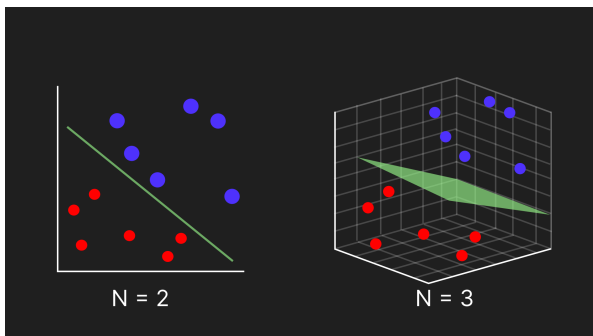


Figure 2: SVM Example (click on image to view larger)

Figure 2 shows two SVMs, one that uses two features to distinguish and one that uses three. The hyperplane for $n = 2$ is a line of best fit, and new data will either be above or below the line which will in turn classify it into the two groups, red or blue. When n is three the hyperplane is a three dimensional plane of best fit, it works similar to the two dimensional example. The number of features can be very large however it is difficult to visualise higher than three dimensional space.

The hyperplane is optimised to create the largest margin between the input training data. In other words, to create the largest gap between all the reference points.

RANDOM FOREST

Random Forest is a learning method for classification developed by Leo Breiman and Adele Cutler [15]. First published in 2001, later trademarked in 2006 [16], it is based on the concept of decision trees, where a large number of decision trees are created and combined to form a "forest" of trees.

The idea behind this method is that each tree in the forest will make a prediction, and the final prediction will be the average of the predictions made by all the trees in the forest.

It uses a technique called "The Random Subspace Method", created by Tin Kam Ho of Bell Laboratories in 1998 [17], which selects a random subset of features for each tree, rather than using all the available features to classify. This helps to reduce correlation between trees and improves the overall performance of the model as each individual tree will check the new data against a range of different identifiable features.

WORKING PRINCIPLE

A decision tree is created using a known dataset, which can then be used to make predictions on new sets of data. However, a single huge decision tree is prone to overfitting, which leads to poor performance and misclassification. As mentioned to overcome this problem, the Random Forest model uses the Random Subspace Method to select random samples of the data, to create multiple decision trees.

As shown in Figure 3, the final decision tree is created by combining the predictions of the smaller individual decision trees. This final decision tree is known as the "Random Forest".

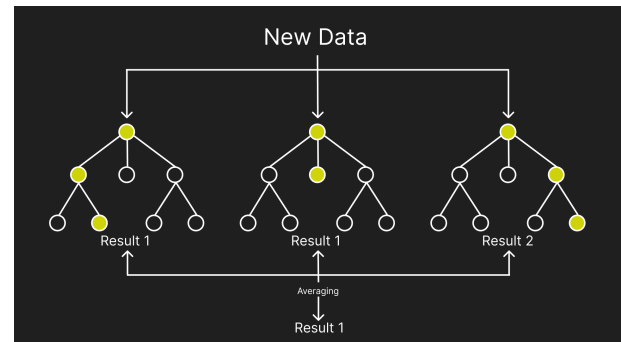


Figure 3: RF Example (click on image to view larger)

CONVOLUTIONAL NEURAL NETWORK

Convolutional Neural Networks (CNNs) are a class of deep learning neural networks designed to process data that has a grid-like topology, such as an image. It was first developed by Yann LeCun in 1998 [18].

CNNs are extremely useful for image recognition and classification tasks as they are able to identify features in images using convolutional layers, which scan the image and apply filters to extract features. The extracted features can be combined and analysed to classify the image.

WORKING PRINCIPLE

CNNs are an improvement over ANNs for image classification. In image classification, ANNs feed each pixel into a neuron. Whilst this is ok for low resolution images, in higher resolution images, it would require a large amount of neurons which would decimate performance and increase resource usage.

A CNN however, takes the input and passes it through many stages that identifies only the key features of an input image. The identified features can then be passed to an ANN to classify the image.

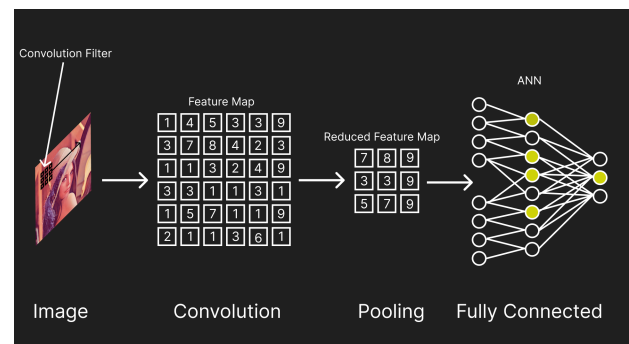


Figure 4: CNN Example (click on image to view larger)

A CNN can have many stages that are responsible for specific tasks; however there are three key stages in any CNN, convolution, pooling and the fully connected stage shown in Figure 4.

The convolutional layer is the core building block of a CNN. It applies a set of filters to the input image. These filters can identify specific features in the input such as, straight lines or more complex things like eyes etc. The identified features are then reduced to a feature map which is fed into a pooling stage.

The pooling stage helps to reduce the dimensionality of the data by downsampling the feature map. This reduces the size of the feature map matrix whilst retaining the most important data. This reduced matrix is then fed to the fully connected stage which is normally a ANN. The data from the down-sampled feature map is fed into each neuron after which the ANN will perform the classification as described in the Artificial Neural Network section.

III. PARALLELISATION STRATEGIES

All software programs, whether it be a simple hello world or an operating system, are nothing more than a series of instructions executed by a processor. These instructions can be organised together to create tasks which can take inputs and perform a specific operation. More formally, James McKenney defines a task in computing as a unit of execution or a unit of work [19]. Multiple tasks are stringed together to create a functioning program.

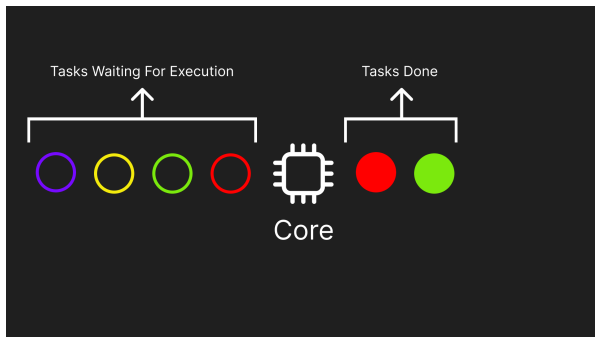


Figure 5: Sequential Execution (click on image to view larger)

In sequential execution all the tasks are completed in order on one core/processor shown in Figure 5. The speed of the process depends on the speed of the processor. A faster processor will execute the program faster and therefore complete the individual tasks quicker. However, due to issues mentioned previously, processor speeds can not infinitely increase and will hit a limit. This poses a problem, as programs become more demanding having more tasks that take longer, there needs to be a way to increase performance to match demand.

To solve this issue a programmer can leverage multiple cores and divide the tasks between the cores shown in Figure 6. Here there are two cores which can execute tasks in parallel, this way the throughput of the program is doubled and the execution time is reduced without the need for a core that runs twice as fast.

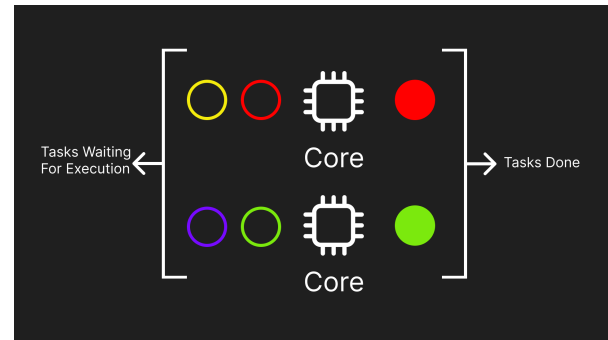


Figure 6: Parallel Execution (click on image to view larger)

This in theory this yields 2x the performance and should decrease execution time by a half; with more cores it should be expected the performance scales linearly with execution time being equal to

$$\frac{1}{n} * t$$

Where **n** is the number of cores and **t** is the execution time on one core.

However, in practice this is not the case due to a variety of factors such as, difference in individual core performance, context switching, access to shared resources, difference in compute needed for each task and the fact that not every task needs to be executed in parallel. Nonetheless, a programmer can still yield significant results by paralleling the areas that can be parallelised. However, depending on how much of a program can be parallelised the speedup that can be achieved will approach a limit in line with Amdahl's law [20] shown in Figure 7.

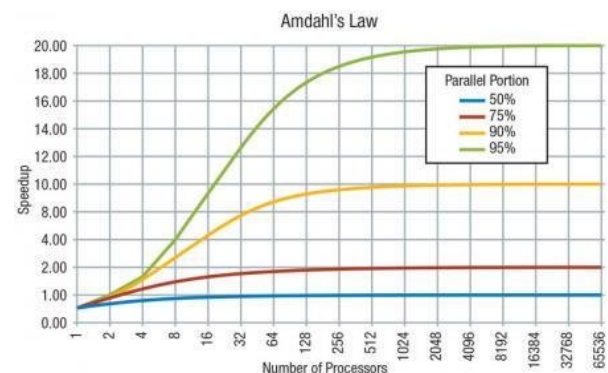


Figure 7: Graph of Amdahl's Law [Fig 1, 21]

From the graph, a program with 90% run in parallel, can only achieve a maximum speedup of 10x regardless of the number of cores used.

To achieve such an increase careful consideration must be employed to efficiently split the tasks between the cores minimising lock. There are many methods that aim to achieve this for specific algorithms; however, in a broad sense, programs can be broken down into **Task Decomposition** and **Data Decomposition**.

TASK DECOMPOSITION

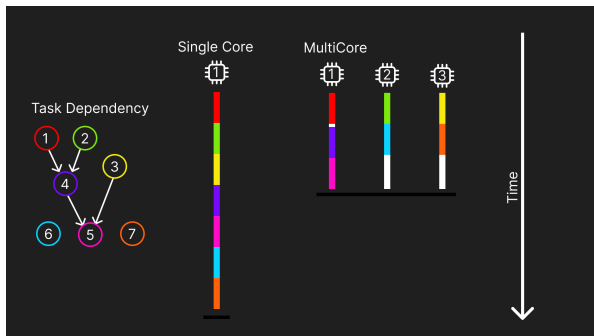


Figure 8: Task Decomposition (click on image to view larger)

Task decomposition is the first tactic in parallelisation. As all programs can be abstracted as a sequence of tasks where a task is just a unit of work, a task dependency graph can be created. Some tasks can be executed on their own but some will need data that is created by other tasks. This is illustrated in Figure 8, tasks that are dependent on other tasks are shown with an arrow.

In a sequential program each task is executed one after the other however, in multicore programs the tasks can be split between the cores and executed in parallel reducing the execution time. As some tasks will depend on the result from other tasks there may be a period of idle time on some cores as they wait for the data to be available before they can execute, this is shown by the white bars in Figure 8.

It is worth noting that the tasks do not need to take the same amount of compute time and can vary widely. However, if tasks are very short it may be worth executing them sequentially or bunching them together before assigning them to a core as task assignment itself does have inherent overhead.

DATA DECOMPOSITION

Data decomposition is a specific type of task decomposition that allows for the parallelisation of the same task executed multiple times on different data shown in Figure 9.

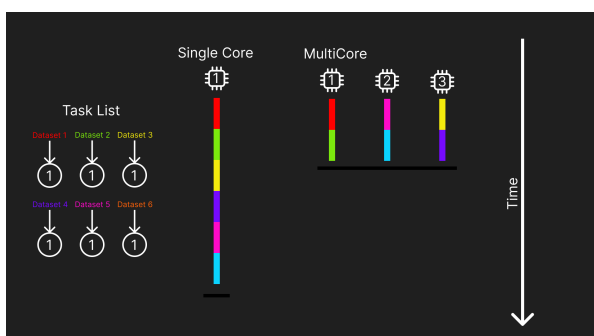


Figure 9: Data Decomposition (click on image to view larger)

Unlike task decomposition, here all cores execute the same instruction on different values from

memory. Figure 9 shows the cores executing the same task on different datasets; however, this could be one large dataset that is subdivided into different sections, then recombined.

One example of this would be in matrix multiplication. Each element in the output matrix is calculated using the same algorithm; however, it uses different input and output locations. A sequential program would work each element out in order but, in a parallel version, the rows in the output could be split among a set of cores to process in parallel.

A combination of these two strategies can be used to efficiently parallelise any program. First identify the different tasks the program executes then parallelise them. Once this is done, each task can be analysed to see if the data within those tasks can be decomposed and executed on a number of threads.

There is one major flaw with this method due to parallelisation being achieved with the use of threads. Threads are a way to execute a unit of work on a core in a fair manner so that all applications that are running can run concurrently. The creation of a thread and the switching a thread does between workloads will add overhead to the program's execution time.

This means that certain tasks will not benefit from being executed in parallel as the time to create a thread will outweigh the time it would take to execute sequentially.

To alleviate this issue each program should be analysed with a profile analyser to identify how long each section or task takes to execute. This data can be used to identify the areas (hotspots) that would benefit from parallelisation and the areas that will not. This paper will use Intel's VTune Profiler [22] to achieve this.

IV. SERIAL PROGRAM ANALYSIS

INTEL VTUNE

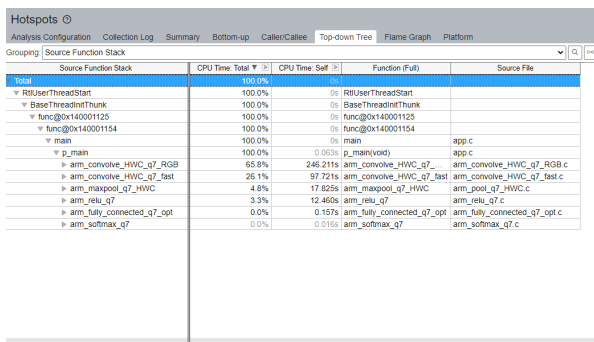
Intel vTune is a performance analysis tool that is used to optimise software applications and draw performance metrics. It allows developers to identify bottlenecks in their source code and see where improvements could be made. It can be used to profile C, C++, C#, Fortran, as well as Python, Java, and JavaScript applications.

vTune uses a variety of sampling and tracing techniques to collect detailed performance data from an application which can then be visualised using the vTune user interface. The user can select specific areas of the code to focus on, such as certain functions or loops, and view performance metrics such as CPU usage, memory access patterns, cache misses and hotspots.

Each of the learning models were subject to a hotspot analysis using vTune to identify the areas in which the CPU spent the majority of its computing. To get an accurate reading the original applications were modified. This mainly included the removal of all **printf** statements as this would add significant CPU time to

the application without actually contributing to the computation itself. Parts of computation that were done inside a **printf** statement were removed and executed before the statement. The modified source codes that were used to complete the vTune analysis can be found on the project GitHub [23].

The data obtained from vTune showed the breakdown of what functions caused bottlenecks. This allowed for identification of the areas that should be considered for parallelisation and what areas are better suited for sequential execution.



Hotspots

Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Flame Graph Platform

Grouping: Source Function Stack

| Source Function Stack | CPU Time: Total | CPU Time: Self | Function (Full) | Source File |
|----------------------------|-----------------|----------------|----------------------------|------------------------------|
| Total | 100.0% | 0s | | |
| RTUserThreadStart | 100.0% | 0s | RTUserThreadStart | |
| BaseThreadInitThunk | 100.0% | 0s | BaseThreadInitThunk | |
| func@0x140001125 | 100.0% | 0s | func@0x140001125 | |
| func@0x140001154 | 100.0% | 0s | func@0x140001154 | |
| main | 100.0% | 0s | main | main.c |
| p_main | 100.0% | 0.375s | p_main | main.c |
| arm_convolve_HWC_q7_RGB | 65.6% | 246.211s | arm_convolve_HWC_q7... | arm_convolve_HWC_q7_RGB.c |
| arm_convolve_HWC_q7_fast | 26.1% | 97.721s | arm_convolve_HWC_q7_fast | arm_convolve_HWC_q7_fast.c |
| arm_maxpool_q7_HWC | 4.8% | 17.625s | arm_maxpool_q7_HWC | arm_pool_q7_HWC.c |
| arm_relu_q7 | 3.3% | 12.460s | arm_relu_q7 | arm_relu_q7.c |
| arm_fully_connected_q7_opt | 0.0% | 0.157s | arm_fully_connected_q7_opt | arm_fully_connected_q7_opt.c |
| arm_softmax_q7 | 0.0% | 0.016s | arm_softmax_q7 | arm_softmax_q7.c |

Figure 10: vTune Top Down Tree Example on CNN

vTune will provide data, shown in Figure 10, detailing what functions contributed to the overall runtime as a percentage. It also displays the time in seconds taken to execute the function however, these can not be taken as definitive as it will vary from system to system. Furthermore, vTune itself will add some overhead to the program's execution.

Not everything in these tables will be relevant or actually editable for example, functions like **malloc** will take up a significant amount of time but can not be changed or edited to improve performance. This paper summarises the data vTune provided and highlights only the relevant areas. The full data can be found in the project github [23].

ARTIFICIAL NEURAL NETWORK

| Source Function Stack | CPU Time: Total | CPU Time: Self | Function (Full) | Source File |
|----------------------------|-----------------|----------------|----------------------------|-------------|
| Total | 100.0% | 0s | | |
| RTUserThreadStart | 100.0% | 0s | RTUserThreadStart | |
| BaseThreadInitThunk | 100.0% | 0s | BaseThreadInitThunk | |
| func@0x140001125 | 100.0% | 0s | func@0x140001125 | |
| func@0x140001154 | 100.0% | 0s | func@0x140001154 | |
| main | 100.0% | 0s | main | main.c |
| p_main | 100.0% | 0.155s | p_main | main.c |
| neural_net_run | 96.4% | 0.999s | neural_net_run | main.c |
| neuron_evaluate | 85.7% | 4.820s | neuron_evaluate | main.c |
| linked_list_get_next | 4.6% | 1.007s | linked_list_get_next | main.c |
| linked_list_start_iterator | 1.2% | 0.266s | linked_list_start_iterator | main.c |
| linked_list_length | 0.4% | 0.070s | linked_list_length | main.c |
| neural_net_add_neuron | 1.3% | 0.016s | neural_net_add_neuron | main.c |
| classify | 1.1% | 0.251s | classify | main.c |
| neural_net_add_layer | 0.2% | 0s | neural_net_add_layer | main.c |
| create_neural_net | 0.1% | 0s | create_neural_net | main.c |

Figure 11: ANN Hotspots (click on image to view larger)

The data for the ANN shows that 96.4% of the computation time is spent doing **neural_net_run** where 85.7% of that is from the **neuron_evaluate** function. The other functions make up for such a small proportion of the runtime that it is not worth parallelising them as the creation and destruction of threads will add more

overhead compared to the speedup that could be achieved.

SUPPORT VECTOR MACHINE

| Source Function Stack | CPU Time: Total | CPU Time: Self | Function (Full) | Source File |
|-----------------------|-----------------|----------------|---------------------|-------------|
| Total | 100.0% | 0s | | |
| RTUserThreadStart | 100.0% | 0s | RTUserThreadStart | |
| BaseThreadInitThunk | 100.0% | 0s | BaseThreadInitThunk | |
| func@0x140001125 | 100.0% | 0s | func@0x140001125 | |
| func@0x140001154 | 100.0% | 0s | func@0x140001154 | |
| main | 100.0% | 0s | main | main.c |
| p_main | 100.0% | 0.375s | p_main | main.c |
| svm_compute | 83.6% | 3.992s | svm_compute | main.c |
| classify | 7.2% | 0.343s | classify | main.c |
| clock_gettime | 1.3% | 0s | clock_gettime | main.c |

Figure 12: SVM Hotspots (click on image to view larger)

Figure 12 shows that for the SVM 83.6% of the execution time is due to one function, **svm_compute**, **classify** takes 2nd place but as this program only classifies between three objects parallelising that region will not be efficient. If the program was processing multiple results then parallelising this region could yield good results.

RANDOM FOREST

| Source Function Stack | CPU Time: Total | CPU Time: Self | Function (Full) | Source File |
|-----------------------|-----------------|----------------|-----------------------|-------------|
| Total | 100.0% | 0s | | |
| RTUserThreadStart | 100.0% | 0s | RTUserThreadStart | |
| BaseThreadInitThunk | 100.0% | 0s | BaseThreadInitThunk | |
| func@0x140001125 | 100.0% | 0s | func@0x140001125 | |
| func@0x140001154 | 100.0% | 0s | func@0x140001154 | |
| main | 100.0% | 0s | main | main.c |
| p_main | 100.0% | 0.313s | p_main | main.c |
| fit_model | 36.1% | 0.016s | fit_model | main.c |
| predict | 29.3% | 0.456s | predict | main.c |
| majority_vote_predict | 8.5% | 0.249s | majority_vote_predict | main.c |
| create_array_3d | 8.0% | 0.095s | create_array_3d | main.c |
| read_data_3d | 5.9% | 0.173s | read_data_3d | main.c |
| clock_gettime | 1.5% | 0s | clock_gettime | main.c |

Figure 13: RF Hotspots (click on image to view larger)

The RF model has two major areas of compute, **fit_model** and **predict** and, one minor area, **majority_vote_predict**. All these areas could benefit from parallelisation however the first two will contribute to the majority of the speedup.

CONVOLUTIONAL NEURAL NETWORK

| Source Function Stack | CPU Time: Total | CPU Time: Self | Function (Full) | Source File |
|----------------------------|-----------------|----------------|----------------------------|------------------------------|
| Total | 100.0% | 0s | | |
| RTUserThreadStart | 100.0% | 0s | RTUserThreadStart | |
| BaseThreadInitThunk | 100.0% | 0s | BaseThreadInitThunk | |
| func@0x140001125 | 100.0% | 0s | func@0x140001125 | |
| func@0x140001154 | 100.0% | 0s | func@0x140001154 | |
| main | 100.0% | 0s | main | main.c |
| p_main | 100.0% | 0.063s | p_main | main.c |
| arm_convolve_HWC_q7_RGB | 65.6% | 246.211s | arm_convolve_HWC_q7... | arm_convolve_HWC_q7_RGB.c |
| arm_convolve_HWC_q7_fast | 26.1% | 97.721s | arm_convolve_HWC_q7_fast | arm_convolve_HWC_q7_fast.c |
| arm_maxpool_q7_HWC | 4.8% | 17.625s | arm_maxpool_q7_HWC | arm_pool_q7_HWC.c |
| arm_relu_q7 | 3.3% | 12.460s | arm_relu_q7 | arm_relu_q7.c |
| arm_fully_connected_q7_opt | 0.0% | 0.157s | arm_fully_connected_q7_opt | arm_fully_connected_q7_opt.c |
| arm_softmax_q7 | 0.0% | 0.016s | arm_softmax_q7 | arm_softmax_q7.c |

Figure 14: CNN Hotspots (click on image to view larger)

Although the CNN runs multiple layers of convolution and pooling, from Figure 14 it is clear to see that only two functions make up 92% of the runtime:

- **arm_convolve_HWC_q7_RGB**
- **arm_convolve_HWC_q7_fast**

V. PROGRAM MODIFICATIONS

ARTIFICIAL NEURAL NETWORK

The data from vTune identified that **neuron_evaluate** is the function that the program spends the majority of time executing. Therefore, by parallelising this region a performance increase should be expected.

```
413 double _neuron_evaluate(neuron_t* neuron, double *data){
414
415     int i;
416     double result;
417     for(i = 0, result = 0 ; i < neuron->n_weights ; i++){
418
419         result += neuron->weights[i] * data[i];
420
421     }
422
423
424     return _activation_func(result + neuron->bias);
425 }
```

Figure 15: neuron_evaluate function
(click on image to view larger)

Figure 15 shows the **neuron_evaluate** function, the main bulk of execution is done in the for loop on line 417. Here, OpenMP could be used to split up the iterations of this loop between a team of threads that will individually evaluate the result. Each thread's individual result calculation can then be summed up to create the final answer; this is an example of data decomposition.

```
417 double _neuron_evaluate(neuron_t* neuron, double *data){
418
419     int i;
420     double result = 0;
421     #pragma omp parallel for reduction(+:result) private(i)
422     for(i = 0; i < neuron->n_weights ; i++)
423     {
424
425         result += neuron->weights[i] * data[i];
426
427     }
428
429
430     return _activation_func(result + neuron->bias);
431 }
```

Figure 16: neuron_evaluate function parallelised.
(click on image to view larger)

This for loop can be parallelised, as shown in Figure 16, by adding the following **#pragma** before the loop. Here the **reduction** clause ensures that each thread calculates the value of **result** individually then performs a sum of all the **result** variables that each thread has to give the final answer. **Private** is also used to give each thread their own copy of **i** so that one thread does not cause another thread to skip iterations.

Now that the function responsible for 85.7% of the execution time is parallelised, the overall execution time for this network should be substantially lower. However, if this program was run with this **#pragma** inserted, the data will show that the performance is substantially worse than the sequential version.

Conventional logic would imply that executing parts with multiple threads will result in better performance however, this neglects the fact that creation and destruction of threads has a performance cost.

Therefore, **executing parts with more threads will result in better performance only when the time to execute sequentially is greater than the time to create, setup and destroy threads.** The exact time to execute sequentially will depend on the single threaded performance which itself is dependent on the CPU and microarchitecture of the chip. **This means that the performance improvement of parallelism will not be the same across all machines, some will experience a higher boost than others and some may not experience any boost at all.**

Taking a deeper look at the for loop in Figure 15, it is clear to see that it runs for:

"i < neuron->n_weights"

In this example program, **neuron->n_weights** evaluates to four. Therefore, this loop is just four multiplications and four additions to make a total of eight instructions. It takes significantly more instructions to create, setup and destroy threads. So in this instance, this loop will run faster on one thread rather than multiple.

For demonstration purposes, this project created two parallel versions of this network, a fast and slow version. The slow version implements the **#pragma** shown in Figure 16, the fast version places the pragmas in more strategic places.

In a network where there are more neuron weights this pragma location would be valid and could result in performance increases; however, for this example code, it should be left out to maximise performance.

Inspecting the original code will reveal that the largest loop is within **main** itself.

```
330     for(i = 0 ; i < 150 ; i++){
331         double *result;
332         result = neural_net_run(neural_net, test_data[i] + 1, 4);
333         int class = classify(result, 3);
334         printf("If %lf %lf -> %d\n", *result, result[1], result[2], class);
335
336         free(result);
337     }
```

Figure 17: Largest loop in ANN (click on image to view larger)

The loop, shown in Figure 17, runs 150 times and executes many instructions, if any performance can be gained it will be here.

Parallelising this loop is not as simple as before due to it utilising the neural network that has been created earlier in the code. This network is a shared resource and implemented using linked lists. This adds an additional level of complexity as even if the shared network was protected using critical regions, due to it being a linked list, each thread will update the

list iterators to different memory locations. This means that when the next threads attempt to use the network the iterators will point to different neurons and will lead to an incorrect result.

The solution to this would be to drastically change the network's implementation or to give each thread their own neural network to work on.

```

202 double results[150][3];
203
204 // The classification of each row of data
205 int classification[150];
206
207 #pragma omp parallel
208 {
209     // Each thread has its own neural net to classify it
210     neural_net_t *neural_net = create_neural_net();
211
212     neural_net_add_layer(neural_net);
213     double w_0_0[] = { 0.18284023, -0.8529724, 0.9295352, 0.27745247;
214     neural_net_add_neuron(neural_net, w_0_0, 4, -0.5623624);
215
216     double w_0_1[] = {-0.17889775, -0.42715448, 0.5908963, 0.45988828;
217     neural_net_add_neuron(neural_net, w_0_1, 4, -0.05714933);
218
219     double w_0_2[] = {-0.48771185, -0.3112061, -0.3137951, 0.5289104 };
220     neural_net_add_neuron(neural_net, w_0_2, 4, -0.053803623);
221
222     double w_0_3[] = { 0.24778583, 0.9340235, -1.4063437, -1.3005494
223     neural_net_add_neuron(neural_net, w_0_3, 4, 0.27202815);
224
225     double w_0_4[] = { 0.7213577, -0.12556425, -1.1483048, -0.083784 };
226     neural_net_add_neuron(neural_net, w_0_4, 4, 0.41853805);
227
228     double w_0_5[] = { 0.85532144, 0.47561175, -0.17761162, -1.3145427;
229     neural_net_add_neuron(neural_net, w_0_5, 4, 0.28362706);
230
231     double w_0_6[] = { 0.12644108, -0.6311569, 0.03045397, -0.53840036;
232     neural_net_add_neuron(neural_net, w_0_6, 4, 0.025358625);
233
234     double w_0_7[] = { 0.21844819, 0.30629316, -0.961491, -0.46816182;
235     neural_net_add_neuron(neural_net, w_0_7, 4, 0.6091357);
236
237     neural_net_add_layer(neural_net);
238     double w_1_0[] = {-1.2635437, -1.316526, 0.28543112, 1.3884338, 0.45282924,
239     1.0371606, 0.12748991, 0.6325952 };
240     neural_net_add_neuron(neural_net, w_1_0, 8, -0.43309692);
241
242     double w_1_1[] = { 0.497385, -0.632777, 0.38966534, -1.2256211, -0.19634834,
243     0.25377455, 0.6375568, -0.5464157 };
244     neural_net_add_neuron(neural_net, w_1_1, 8, -0.14760242);
245
246     double w_1_2[] = { 0.3835234, 0.10146759, -0.32170305, -1.1918751, -1.1514843,
247     -0.8962826, -0.69583875, -0.74578637;
248     neural_net_add_neuron(neural_net, w_1_2, 8, 0.14179966);
249
250     #pragma omp for private(i)
251     for(i = 0 ; i < 150 ; i++)
252     {
253         // Run the neural net
254         double *result;
255         result = neural_net_run(neural_net, test_data[i] + 1, 4);
256         classification[i] = classify(result, 3);
257
258         // Copy the result to the result array
259         memcpy(results[i], result, sizeof(double) * 3);
260         free(result);
261     }
262
263     // Print the results
264     for(i = 0 ; i < 150 ; i++){
265         printf("X%f X%f X%f -> %d\n", results[i][0], results[i][1], results[i][2], classification[i]);
266     }
267 }

```

Figure 18: Thread safe parallelisation of ANN
(click on image to view larger)

Figure 18 shows this implementation in practice. There are two variables created to store the result and classification of the 150 input data samples on line 292 and 295. They are created outside the parallel section so that the results are accessible after the team of threads complete their calculations.

The original printf statements have been moved into their own loop. If they were inside the parallel section the performance will suffer as the output console is a shared resource and will need to be protected to prevent threads from overwriting each other. By having them in their own loop, a critical section, that uses semaphores, can be avoided.

Inside the parallel section, line 297 - 335, each thread creates its own version of the network. This local network is the one the *neural_net_run* function uses which is inside the for loop and split amongst the threads. The result is then copied into the external results array.

In this version of the program instead of one thread executing all 150 samples, the samples are split amongst a team of threads and executed together. This implementation does have one drawback, as each thread is executing on their own network it will increase the total memory usage by the program.

On a system with limited RAM, parts of the memory could be passed into swap memory which will add memory latency delays to the application. This is particularly a problem with large amounts of threads or large networks with hundreds of neurons.

SUPPORT VECTOR MACHINE

Using the data from vTune, the SVM code is bottlenecked by the *svm_compute* function, pictured below in Figure 19.

```

63 float svm_compute(float sample[], int n_svs, float svcs[][2], float alphas[], float bias){
64
65     int i = 0;
66     float acc_sum = 0;
67
68     for(i = 0 ; i < n_svs ; i++){
69
70         acc_sum += ((sample[0] * svcs[i][0] + sample[1]*svcs[i][1])*alphas[i]);
71
72     }
73
74     return acc_sum + bias;
75 }

```

Figure 19: Original svm_compute function
(click on image to view larger)

Like before, the majority of the work done in this function is within the for loop. Using OpenMP the for loop could be parallelised and split amongst threads shown in Figure 20.

```

66 float svm_compute(float sample[], int n_svs, float svcs[][2], float alphas[], float bias){
67
68     int i = 0;
69     float acc_sum = 0;
70
71     #pragma omp parallel for private(i) reduction(+:acc_sum)
72     for(i = 0 ; i < n_svs ; i++){
73
74         acc_sum += ((sample[0] * svcs[i][0] + sample[1]*svcs[i][1])*alphas[i]);
75
76     }
77
78     return acc_sum + bias;
79 }

```

Figure 20: svm_compute parallelised
(click on image to view larger)

Here the reduction clause is used to calculate the *acc_sum* value safely and *i* is private between threads. Although this function makes up 83.6% of the execution time, this parallel version of the SVM program will suffer horrible performance for the same reasons as the ANN program.

Here the loop runs for *n_svs* times which is a maximum of sixteen but usually two. This with the fact that the computation is just a few multiplications and additions means that threading this part of the program will not yield good performance.

Like previously, for demonstration purposes, this project created two parallel versions of this network, a fast and a slow version. The slow version implements the *#pragma* shown in Figure 20, the fast version places the pragmas in more strategic areas.

A better area for parallelisation will be the for loop in main, shown below.

```
281 float results[3];
282 for(int i = 0 ; i < 150 ; i++){
283
284     results[0] = svm_compute(samples[i], 2, set_vers_svs, set_vers_alphas, set_vers_bias);
285     results[1] = svm_compute(samples[i], 2, set_virg_svs, set_virg_alphas, set_virg_bias);
286     results[2] = svm_compute(samples[i], 16, versl_virg_svs, versl_virg_alphas, versl_virg_bias);
287
288     printf("Xid: ", i);
289     printf("Xsf, ", results[0]);
290     printf("Xsf, ", results[1]);
291     printf("Xsf\n", results[2]);
292     printf("final class -> %d\n", classify(results));
293
294 }
```

Figure 21: SVM main for loop (click on image to view larger)

This loop runs for 150 times and executes multiple instructions in each iteration. Parallelising this will benefit more than parallelising the smaller loops in Figure 19.

```
285 float results[150][3];
286 final_classes_t final_class[150];
287 #pragma omp parallel for
288 for(int i = 0 ; i < 150 ; i++){
289
290     results[i][0] = svm_compute(samples[i], 2, set_vers_svs, set_vers_alphas, set_vers_bias);
291     results[i][1] = svm_compute(samples[i], 2, set_virg_svs, set_virg_alphas, set_virg_bias);
292     results[i][2] = svm_compute(samples[i], 16, versl_virg_svs, versl_virg_alphas, versl_virg_bias);
293     final_class[i] = classify(results[i]);
294
295 }
296
297 // for loop to print results
298 for(int i = 0 ; i < 150 ; i++){
299 {
300     printf("Xid: ", i);
301     printf("Xsf, ", results[i][0]);
302     printf("Xsf, ", results[i][1]);
303     printf("Xsf\n", results[i][2]);
304     printf("final class -> %d\n", final_class[i]);
305 }
```

Figure 22: SVM main loop parallelised (click on image to view larger)

Figure 22 shows how parallelisation was achieved with this loop. There is a new variable to hold the classification and the old results array has been made three dimensional so each thread can store the result without needing to use critical sections that would hinder performance.

Once again, the print statements have been moved outside the parallel section for the same reasons as previously mentioned in the Artificial Neural Network section.

The parallel section now only contains the computation. The results are stored into arrays created outside and because it is indexed by the for loop, data race and overwriting issues can be avoided. There is no need to specify any private variables as *i*, the loop iterator, is created inside the parallel region and therefore private by default.

This version of the program is the fast variant and should perform better than the slow implementation.

RANDOM FOREST

The vTune data suggest that the functions that create the bottlenecks in the RF program are, **fit_model**, **predict**, **majority_vote_predict**. As mentioned before, although these may be the bottlenecks it is not always efficient to parallelise these sections.

The RF program has many loops that could be parallelised however, most only run for **n_estimators** which evaluates to ten times. Therefore, although these

loops could be executed in parallel it is often not worth doing so.

The best loop to parallelise in this program is the one in main that runs 75 times, shown below.

```
416 for(int j = 0 ; j < 75 ; j++)
417 {
418     printf("Xid -> ", j);
419
420     for(i=0; i < 4; i++){
421         printf("Xf, ", test_data[j][i]);
422     }
423
424     printf(" -> ");
425
426     for(i=0; i < n_estimators; i++)
427     {
428         predictions[i] = predict(rf[i], test_data[j]);
429         printf("Xf, ", predictions[i]);
430     }
431     printf("-> %f \n", majority_vote_predict(predictions, n_estimators));
432 }
```

Figure 23: Main Loop in RF (click on image to view larger)

Using the same strategies as the previous two programs the loop can be parallelised as shown in Figure 24.

```
412 double accuracy;
413 float sample[6];
414 float predictions[75][10];
415 char send_string[300];
416
417 float answers[75];
418
419 #pragma omp parallel for
420 for(int j = 0 ; j < 75 ; j++)
421 {
422     // #pragma omp parallel for
423     for(int i=0; i < n_estimators; i++)
424     {
425         predictions[j][i] = predict(rf[i], test_data[j]);
426     }
427
428     answers[j] = majority_vote_predict(predictions[j], n_estimators);
429
430 }
431
432 // For loop that takes the prints out of the parallel section
433 for(int j = 0 ; j < 75 ; j++)
434 {
435     printf("Xid -> ", j);
436
437     for(i=0; i < 4; i++)
438     {
439         printf("Xf, ", test_data[j][i]);
440     }
441
442     printf(" -> ");
443
444     for(i=0; i < n_estimators; i++)
445     {
446         printf("Xf, ", predictions[j][i]);
447     }
448
449     printf("-> %f \n", answers[j]);
450 }
451 }
```

Figure 24: Parallel version of RF (click on image to view larger)

Here the predictions array has been made two dimensional and a new array for the answers has been created so that the threads can store their results from computation.

The old print statements have been moved outside and into their own loop to not hinder performance.

The parallel sanction is left with just the computation of the predictions and the predictions answers.

Examples of how the other loops could be parallelised have been included but commented out to maximise performance.

CONVOLUTIONAL NEURAL NETWORK

For the CNN, the major bottlenecks were the following functions:

- **arm_convolve_HWC_q7_RGB**
- **arm_convolve_HWC_q7_fast**

Both these functions execute large for loops pictured below.

```

241 for (i = 0; i < ch_im_out; i++)
242 {
243     for (j = 0; j < dim_im_out; j++)
244     {
245         for (k = 0; k < dim_im_out; k++)
246         {
247             conv_out = (bias[i] << bias_shift) + NW_ROUND(out_shift);
248             for (m = 0; m < dim_kernel; m++)
249             {
250                 for (n = 0; n < dim_kernel; n++)
251                 {
252                     /* If-for implementation */
253                     in_row = stride * j + m - padding;
254                     in_col = stride * k + n - padding;
255                     if (in_row >= 0 && in_col >= 0 && in_row < dim_in && in_col < dim_in)
256                     {
257                         for (l = 0; l < ch_im_in; l++)
258                         {
259                             conv_out +=
260                                 in_in[in_row * dim_in + in_col] * ch_im_in +
261                                 l * wt[l * ch_im_in * dim_kernel * dim_kernel + (m * dim_kernel +
262                                     n) * ch_im_in + l];
263                         }
264                     }
265                 }
266             }
267             im_out[i + (j * dim_im_out + k) * ch_im_out] = (q7_t) SSAT((conv_out >> out_shift), 0);
268         }
269     }
270 }

```

Figure 25: arm_convolve_HWC_q7_RGB
(click on image to view larger)

```

271 for (i = 0; i < ch_im_out; i++)
272 {
273     for (j = 0; j < dim_im_out; j++)
274     {
275         for (k = 0; k < dim_im_out; k++)
276         {
277             conv_out = (bias[i] << bias_shift) + NW_ROUND(out_shift);
278             for (m = 0; m < dim_kernel; m++)
279             {
280                 for (n = 0; n < dim_kernel; n++)
281                 {
282                     /* If-for implementation */
283                     in_row = stride * j + m - padding;
284                     in_col = stride * k + n - padding;
285                     if (in_row >= 0 && in_col >= 0 && in_row < dim_in && in_col < dim_in)
286                     {
287                         for (l = 0; l < ch_im_in; l++)
288                         {
289                             conv_out +=
290                                 in_in[in_row * dim_in + in_col] * ch_im_in +
291                                 l * wt[l * ch_im_in * dim_kernel * dim_kernel + (m * dim_kernel +
292                                     n) * ch_im_in + l];
293                         }
294                     }
295                 }
296             }
297             im_out[i + (j * dim_im_out + k) * ch_im_out] = (q7_t) SSAT((conv_out >> out_shift), 0);
298         }
299     }
300 }

```

Figure 26: arm_convolve_HWC_q7_fast
(click on image to view larger)

Unlike the previous examples parallelising these loops would be beneficial due to the number of iterations they run for. They can both be parallelised as follows:

```

240 #pragma omp parallel for collapse(3) private(i,j,k,l,m,n,in_row,in_col) reduction(+:conv_out)
241 for (i = 0; i < ch_im_out; i++)
242 {
243     for (j = 0; j < dim_im_out; j++)
244     {
245         for (k = 0; k < dim_im_out; k++)
246         {
247             conv_out = (bias[i] << bias_shift) + NW_ROUND(out_shift);
248             for (m = 0; m < dim_kernel; m++)
249             {
250                 for (n = 0; n < dim_kernel; n++)
251                 {
252                     /* If-for implementation */
253                     in_row = stride * j + m - padding;
254                     in_col = stride * k + n - padding;
255                     if (in_row >= 0 && in_col >= 0 && in_row < dim_in && in_col < dim_in)
256                     {
257                         for (l = 0; l < ch_im_in; l++)
258                         {
259                             conv_out +=
260                                 in_in[in_row * dim_in + in_col] * ch_im_in +
261                                 l * wt[l * ch_im_in * dim_kernel * dim_kernel + (m * dim_kernel +
262                                     n) * ch_im_in + l];
263                         }
264                     }
265                 }
266             }
267             im_out[i + (j * dim_im_out + k) * ch_im_out] = (q7_t) SSAT((conv_out >> out_shift), 0);
268         }
269     }
270 }
271 }

```

Figure 27: arm_convolve_HWC_q7_RGB Parallel
(click on image to view larger)

```

272 #pragma omp parallel for collapse(3) private(i,j,k,l,m,n,in_row,in_col) reduction(+:conv_out)
273 for (i = 0; i < ch_im_out; i++)
274 {
275     for (j = 0; j < dim_im_out; j++)
276     {
277         for (k = 0; k < dim_im_out; k++)
278         {
279             conv_out = (bias[i] << bias_shift) + NW_ROUND(out_shift);
280             for (m = 0; m < dim_kernel; m++)
281             {
282                 for (n = 0; n < dim_kernel; n++)
283                 {
284                     /* If-for implementation */
285                     in_row = stride * j + m - padding;
286                     in_col = stride * k + n - padding;
287                     if (in_row >= 0 && in_col >= 0 && in_row < dim_in && in_col < dim_in)
288                     {
289                         for (l = 0; l < ch_im_in; l++)
290                         {
291                             conv_out +=
292                                 in_in[in_row * dim_in + in_col] * ch_im_in +
293                                 l * wt[l * ch_im_in * dim_kernel * dim_kernel + (m * dim_kernel +
294                                     n) * ch_im_in + l];
295                         }
296                     }
297                 }
298             }
299             im_out[i + (j * dim_im_out + k) * ch_im_out] = (q7_t) SSAT((conv_out >> out_shift), 0);
300         }
301     }
302 }
303 }

```

Figure 28: arm_convolve_HWC_q7_fast Parallel
(click on image to view larger)

Here both loops utilise the OpenMP collapse clause. It takes the number of nested loops and combines the iterations into one super loop that is then divided amongst the threads to execute. Without it, OpenMP will only divide up the iterations of the first loop between the threads which would leave performance on the table.

Additionally, as these loops utilise and modify external variables, each thread should be given private copies of these variables to avoid affecting other threads calculations. The final result, **conv_out**, should be passed into the reduction clause so that the cumulative sum is calculated without errors.

VI. EXPERIMENT SET-UP

The programs were compiled and executed on a total of three machines, one PC, one laptop and a Raspberry Pi 4. The specifications of the machine are listed below.

Table 1: PC Specifications

| | |
|------------------|--|
| CPU | Intel Core i7-8700k (12 Cores) 4.3GHz |
| Memory | 16GB DDR4 3000MHz |
| Operating System | Windows 10 Version 10.0.19044 Build 19044 |
| Compilers Used | gcc 11.2.0 (MinGW-W64 x86_64) g++ 11.2.0 (MinGW-W64 x86_64) |

Table 2: Laptop Specifications

| | |
|------------------|--|
| CPU | Intel Core i5-1135G7 (8 Cores) 4.20 GHz |
| Memory | 16GB DDR4 3200 MHz |
| Operating System | Windows 11 Version 10.0.22623 Build 22623 |
| Compilers Used | gcc 10.3.0 (MSYS2 Project) g++ 10.3.0 (MSYS2 Project) |



Table 3: Raspberry Pi 4 Specifications

| | |
|------------------|--|
| CPU | Broadcom BCM2711 (4 Cores) 1.5GHz |
| Memory | 8GB LPDDR4 3200MHz |
| Operating System | Ubuntu Server 22.10 (Kinetic Kudu) |
| Compilers Used | gcc 12.2.0 (Ubuntu 12.2.0-3ubuntu1) g++ 12.2.0 (Ubuntu 12.2.0-3ubuntu1) |

The applications were compiled using the makefiles provided by the project. All files are available on the Project GitHub [23].

All power limiting settings were disabled, where possible, on both the laptop and the PC to not limit the CPU frequency during runs. The laptop was plugged into power throughout all tests.

VII. METHODOLOGY

The project files were downloaded then compiled according to the instructions specified in each program ReadMe files.

Once the binary files were generated all versions of the program, serial and parallel, were executed 100 times each sequentially after each other to ensure that test conditions were consistent between all runs.

Each program appends its runtime to a comma separated file after each run. Once all 100 runs were complete, the times were copied into an Excel spreadsheet that calculated the average runtime of the application and standardised the speedup against the sequential programs average time.

This data was then used to create plots of the speedup (or slowdown) to help easily visualise the data, these are shown below.

VIII. RESULTS

PC SPEED UP

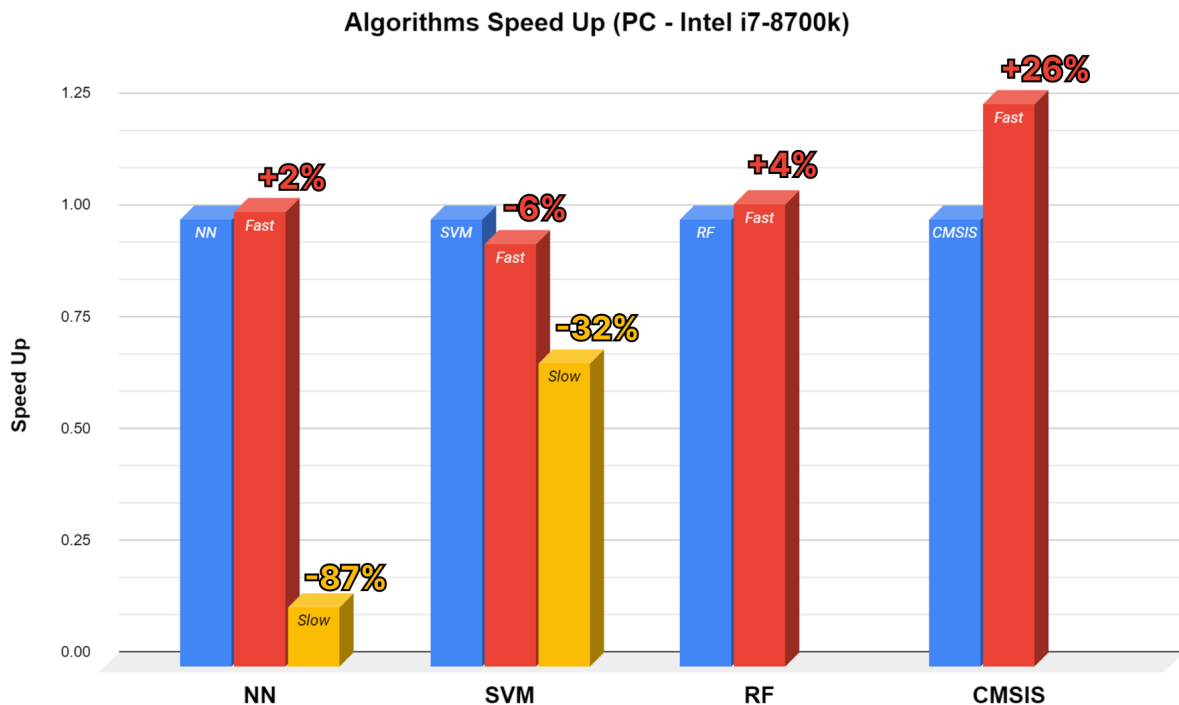


Figure 29: PC Speed Up Results (click on image to view larger)



RASPBERRY PI 4 SPEED UP

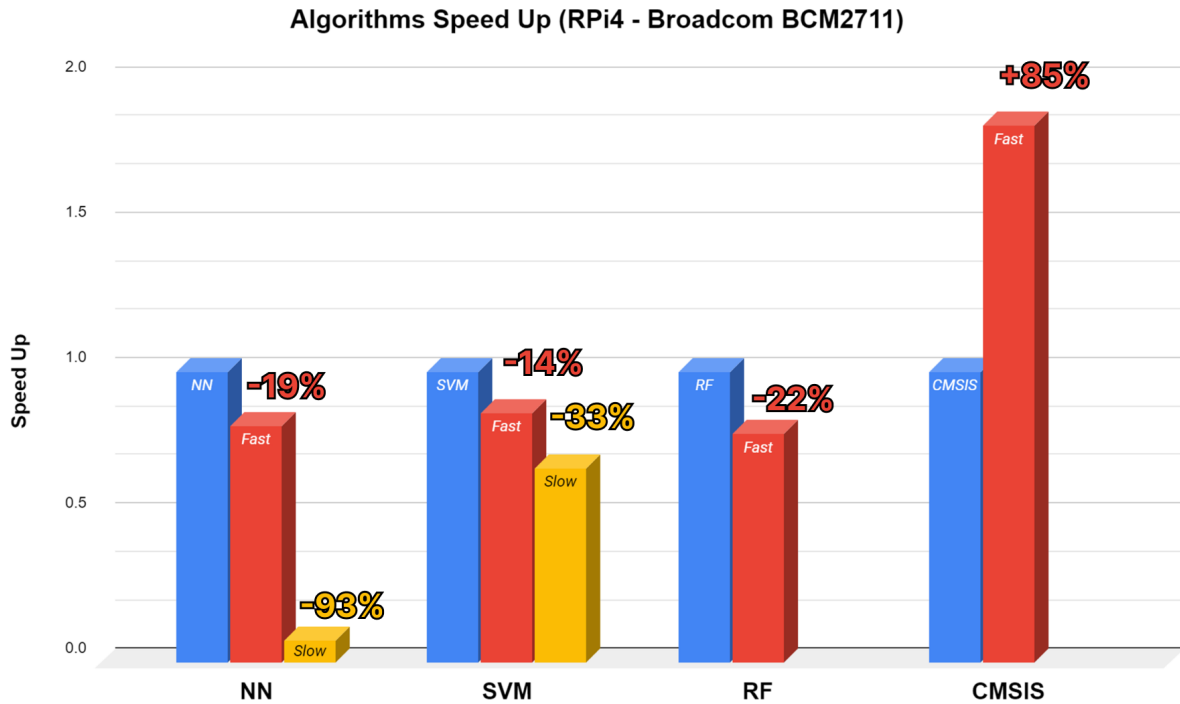


Figure 30: Raspberry Pi 4 Speed Up Results (click on image to view larger)

LAPTOP SPEED UP

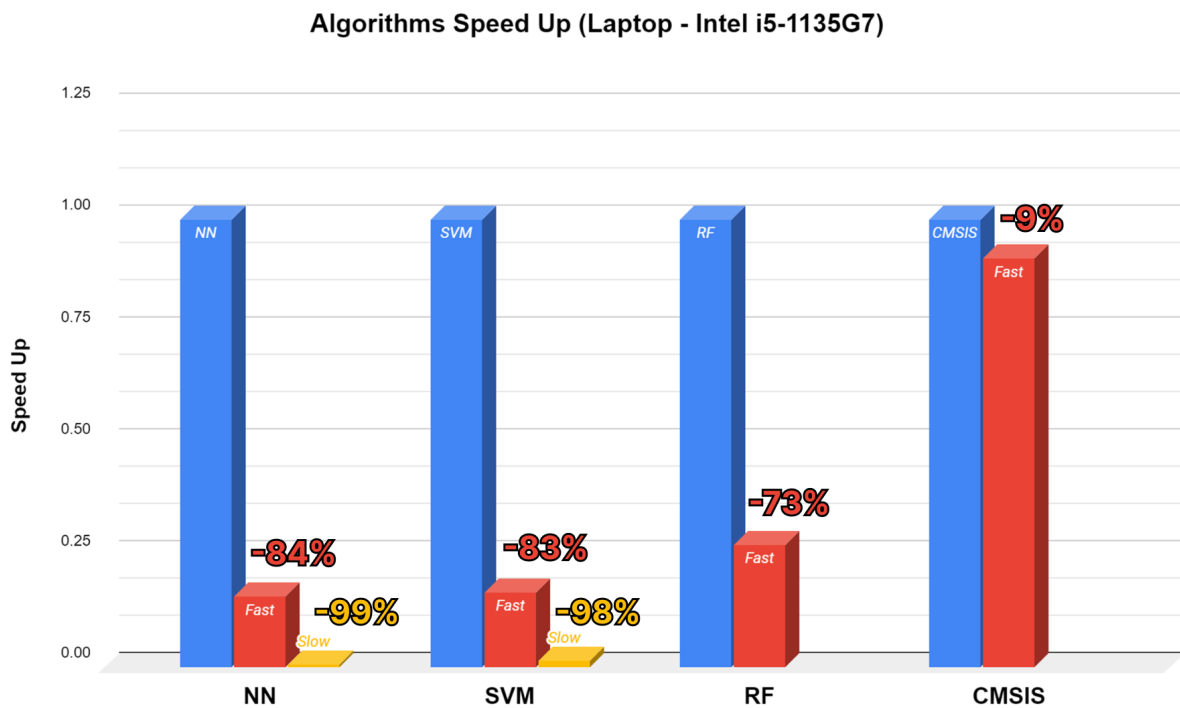


Figure 31: Laptop Speed Up Results (click on image to view larger)

IX. ANALYSIS

Across all machines the slow versions of the algorithms, shown by the yellow bar in Figures 29, 30 and 31, all performed significantly worse than the sequential versions, shown by the blue bar.

Despite these versions executing code in parallel, it is important to strategically pick what areas are executed on multiple cores and what areas to leave sequential to gain any performance improvement rather than attempting to parallelise every single area possible.

However, the SVM application did not gain any performance benefit from parallelisation on any machine despite placing pragmas in strategic areas to try and maximise the performance benefits.

The reason for this could be that the SVM's computation and classification algorithms are the simplest out of the four applications tested. They are composed of very simple multiplications and summations that would be executed very fast sequentially.

This means that the time taken to create threads and assign work to them would dwarf the time taken to perform the calculations sequentially leading to a longer execution time in parallel. If there were more data that needed classification or a more complex SVM algorithm was used, a performance speed up could be expected.

From this it is evident that, not only does the placement of parallel sections matter, but also the instructions and data that is executed within these sections in order to maximise performance.

The PC achieved a speed up on all other algorithms however for the NN and RF the speed increase was only 2% and 4% respectively whereas, the CMSIS CNN gained 26%.

On the other hand, the Raspberry Pi 4 only managed a speed up on the CNN which gained a staggering 85% increase. The NN and RF programs experienced a slow down of 19% and 22% respectively.

The NN and RF algorithms are like the SVM, they are simple multiplications and summations that are executed a few times. Therefore, they could be executed faster sequentially and would not actually benefit from parallelising.

Despite this fact, the PC did gain a small performance increase on these algorithms compared to the other machines. One reason for this could be because the PC has the largest core count out of any machine tested.

This means that the performance impact of creating and setting up threads would be mitigated by the twelve cores that are now available for execution allowing it to slightly outperform the sequential application.

Therefore, it could be said that the performance impact of parallelisation depends on the

number of cores available for execution as well as the efficiency of parallelisation of the code.

On a machine with few cores, it may not be worth executing in parallel; however, on a different machine with lots of cores even though there will be an initial decrease in performance due to creating more threads, this gap could be closed by executing more threads in parallel than the first machine.

One outlier was the results from the laptop. Using the reasoning explained, it is expected that the laptop will also gain a performance increase in some of the algorithms.

However, all four algorithms performed very poorly. They all ran faster sequentially than in parallel. The laptop has the newest CPU out of all the machines tested. This goes to show how important microarchitecture design is when considering the performance increase in parallel computing.

The single threaded performance of the i5-1135G7 in the laptop allowed it to execute the sequential program very quickly. As previously mentioned, parallelising code will result in better performance only when the time to execute sequentially is greater than the time to create, setup and destroy threads. Due to the superior single core performance of the laptop, none of the programs was able to gain a performance increase when executed in parallel.

X. CONCLUSION

In conclusion, it is adamant that the performance of parallelisation depends on a number of factors. Under certain conditions it has been shown that parallelisation will yield substantial improvement to execution time but for the same application, under different conditions, can decrease performance as well.

It was found that single threaded performance, parallelisation code efficiency, amount of data to be executed in parallel and the number of threads available were major factors in determining the speed up that could be gained from parallelisation.

Program profilers such as vTune were effective in identifying hotspots and bottlenecks in code; however, the identified regions may not always be suitable for direct parallelisation.

Profiles should be used in conjunction with a programmer's own intuition and knowledge of the application in order to create the most efficient parallel version.



XI. REFERENCES

- [1] M. Croarken, "The beginnings of the Manchester computer phenomenon: people and influences," *IEEE Annals of the History of Computing*, vol. 15, no. 3, pp. 9–16, 1993, doi: 10.1109/85.222835.
- [2] N. Enticknap, "Computing's Golden Jubilee," *Resurrection Computer Conservation Society*, vol. 20, no. 1, 1998, ISSN: 0958-7403.
- [3] S. Chapman, "Google celebrates 65 years since the birth of University of Manchester's Baby," *Prolific North*, Jun. 21, 2013. Available: <https://www.prolificnorth.co.uk/digital/featured/2013/06/google-celebrates-65-years-birth-university-manchester%E2%80%99s-baby>
- [4] OECD Digital Economy Outlook 2017. OECD, 2017. doi: 10.1787/9789264276284-en
- [5] J. McEnery, "How much computer code has been written? - Modern Stack - Medium," *Modern Stack*, Jul. 18, 2020. Available: <https://medium.com/modern-stack/how-much-computer-code-has-been-written-c8c03100f459>
- [6] B. Schauer, "Multicore Processors—A Necessity," *ProQuest Discovery Guides*, 2008. Available: <https://search.iczhiku.com/paper/U5ZCcgHesYukaFsA.pdf>
- [7] B. Leasure et al., "Power Wall," in *Encyclopedia of Parallel Computing*, Boston, MA: Springer US, 2011, pp. 1593–1608. doi: 10.1007/978-0-387-09766-4_499
- [8] J. Chen, "What Is a Neural Network?," *Investopedia*. Sep. 22, 2022. Available: <https://www.investopedia.com/terms/n/neuralnetwork.asp>
- [9] S. Cohen, *Artificial Intelligence and Deep Learning in Pathology*. Elsevier Health Sciences, 2020.
- [10] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943, doi: 10.1007/bf02478259.
- [11] A. Krogh, "What are artificial neural networks?," *Nature Biotechnology*, vol. 26, no. 2, pp. 195–197, Feb. 2008, doi: 10.1038/nbt1386.
- [12] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organisation in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.
- [13] F. Dario Baptista, S. Rodrigues, and F. Morgado-Dias, "Performance comparison of ANN training algorithms for classification," in *2013 IEEE 8th International Symposium on Intelligent Signal Processing*, Sep. 2013. doi: 10.1109/wisp.2013.6657493
- [14] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995, doi: 10.1007/bf00994018.
- [15] Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001, doi: 10.1023/A:1010933404324.
- [16] MINITAB, LLC, "Random Forests," Dec. 19, 2006. Available: <https://trademarks.justia.com/786/42/random-78642027.html>
- [17] Tin Kam Ho, "The random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998, doi: 10.1109/34.709601.
- [18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.
- [19] J. L. McKenney, *Simultaneous Multiprogramming of Electronic Computers*. University of California, Los Angeles, 1961.
- [20] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, 1967. doi: 10.1145/1465482.1465560
- [21] W. Chun Shiang, I. Abdul Aziz, N. S. Haron, J. Jaafar, N. N. Ismail, and M. Mehat, "THE HIGH PERFORMANCE LINPACK (HPL) BENCHMARK EVALUATION ON UTP HIGH PERFORMANCE CLUSTER COMPUTING," *Jurnal Teknologi*, vol. 78, no. 9–3, Sep. 2016, doi: 10.11113/jt.v78.9715.
- [22] Intel, "Intel® VTune™ Profiler," *Intel*, 2019. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>
- [23] A. Varghese, "22WSD530-Programming-Multi-many-core-Systems-Coursework-2," *GitHub*, 2023. <https://github.com/ajayvarghese2000/22WSD530-Programming-Multi-many-core-Systems-Coursework-2>.