Timing Neural Networks in C and Ada

Martin C. Carlisle, *Senior Member*
Department of Computer Science
United States Air Force Academy
USAFA, CO 80840
00+1-719-333-3590
carlislem@acm.org

Leemon C. Baird III
Department of Computer Science
United States Air Force Academy
USAFA, CO 80840
00+1-719-333-3590
leemon@leemon.com

ABSTRACT
In this paper, we describe a neural network program that was originally developed in C, then ported to Ada 2005. We explain several simple modifications to the Ada code that reduce the overhead from 76% to 0%. These modifications could provide significant performance gains to other applications, allowing them to combine the safety of Ada with the speed of C.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *Ada, C.*

C.4 [**Performance of Systems**]: Performance Attributes.

I.2.6 [**Artificial Intelligence**]: Learning – *connectionism and neural nets.*

General Terms
Performance, Languages.

Keywords
Ada, C, Neural Networks, Benchmarking.

1. INTRODUCTION
C is generally perceived as a highly portable and extremely efficient programming language. Ada, while known for its safety, often suffers from a perception that its use creates a significant performance penalty. Tennebo [1], e.g. while touting that Ada provides the efficiency of C with the elegance of Java, concludes that Ada is about 19% slower than C++. Other authors have published reports that are more favorable for Ada. Weiskirchner [2] reports additional overhead of 21%; however, analyzing his data in the appendices, one notes that Ada often outperforms C when the runtime checks are turned off. Corlan [3] reports identical times for C and Ada, although it appears he had to hand-tune his C code in order to obtain this result.

SIGada'07, November 4–9, 2007, Fairfax, Virginia, USA.
ACM 978-1-59593-876-3/07/0011.

We implemented a neural network program in C because we knew C would be widely available on high-performance computers. The C program required a lot of debugging for subtle pointer errors. Since there was still some concern that subtle errors might remain in the code and we wondered whether we were truly getting better performance by using C, we reimplemented the code in Ada.

In Section 2, we describe the neural network and why it requires such a complicated data structure. In Section 3, we explain how we modified the code in translating it to Ada. Section 4 describes how we made simple changes to the Ada implementation to eliminate the additional overhead. Section 5 provides conclusions and insights for future projects.

2. BACKGROUND
Neural networks are a powerful form of machine learning that is ideal for pattern recognition problems such as machine vision. In simple *supervised learning*, the system is repeatedly given an input (a list of real numbers) and then calculates an output (another such list). During learning, the output is compared to a known "desired value", and various parameters within the network are modified so the actual output more closely approximates the desired output. After several hours of training this way, the system will often learn the underlying patterns, and be able to answer questions it has never seen before.

Neural network algorithms tend to be fairly simple, and so result in short programs with simple data structures. Unfortunately, we were developing a convolutional neural network with a self-organizing algorithm, using second-order techniques to speed learning. This meant that both the algorithms and the data structures were fairly complex, with much pointer arithmetic in the original C implementation.

In a neural network, the network is designed to implement a function that takes several inputs and transform them into several outputs, all of which are real numbers. The network is structured as a set of *neurons*, each of which takes several real-valued *signals* as input, and returns one signal as output. The set of neurons and their connections form an acyclic graph, whose structure must be represented efficiently.

In addition, there are many *weights*, which are adjustable, real-valued parameters that are changed during learning. As these are adjusted, they cause the function computed by the network as a whole to eventually approximate the desired function. Each neuron uses several of the weights when computing its output. In a convolutional network, each weight is used by several different neurons. This causes the pattern of which neuron uses which weight to be very complicated.

Overall, the program must keep track of quite a few things. The network is made of some number of layers. Each layer contains some number of neurons, and different layers can have different numbers of neurons. Each neuron takes some number of inputs from the previous layer, and uses them to calculate an output that will be sent to the next layer. The neurons in each layer are viewed as forming a list of 2D images, with each neuron's output forming one pixel in one of the images. In the next layer, a given neuron receives inputs from a set of rectangular windows, one for each image. A given window will feed into several different neurons in the next layer. Some neurons in a given layer share identical sets of weights, and some neurons use different weights. As a result, the pattern of connections is complicated.

In addition to keeping track of this complicated topology, the network must also keep track of a host of different derivatives. It must calculate the first and second derivatives of the error in the output of the network with respect to each weight. It must also store an exponentially-weighted average of recent first and second derivatives for each weight.

Most of the computation time in the program is spent in basic arithmetic operations of adding, multiplying, and occasionally evaluating hyperbolic tangent functions. We originally wrote the software in highly-optimized C code that took full advantage of the language's pointer arithmetic, which achieves faster code through the use of unsafe (and often unintelligible) coding hacks. We then decided to translate this code into Ada, both as an aid to debugging, and to see how the speed of the code might compare.

3. CODE COMPARISON

Most of the program was a very straightforward translation from C to Ada, performed manually by the authors. However, in a couple of cases we diverged from the C code.

3.1 Pointer Arithmetic

The record for a single neuron in the network had the following description in C:

```
struct neuron_data {
   unsigned int number_inputs;
   double **input_signals;
   // array of pointers to input signals
   int *input_signals_indices;
   double *input_weights;
   // pointer to weights then bias
   double *output_signal;
   // pointer to output signal
};
```

In Ada, we instead chose to represent the record as follows:

```
type Integer_Array is array (Integer
    range <>) of Integer;
type Integer_Array_Ptr is access all
    Integer_Array;
type Neuron_Data is record
    Number_Inputs : Natural;
    Input_Signals : Integer_Array_Ptr;
    Input_Weights : Natural;
    Output_Signal : Natural;
end record;
```

In particular, rather than storing pointers into the signals and weights arrays, we instead stored the indices of these values. The reason for this choice was that when propagating the signal backwards, we need to know these indices. In C, we obtain them from doing pointer subtraction, as:

```
topology->weights_error_derivative[
   (topology->neurons[j].input_weights-
   topology->weights)+k] = …;
```

Here we subtract the pointer to the input_weight minus the pointer to the weights array to get the index of this particular weight, which we then use as an index into the weights_error_derivative array. In Ada, pointer subtraction is awkward to implement (requiring the use of an interface package, or unchecked conversions). Therefore, we simply stored the indices instead. So, when forward propagating, in C, we would use:

```
temp += (
   *(topology->neurons[j].input_signals[k]))
   *topology->neurons[j].input_weights[k];
```

But in Ada we see instead:

```
Temp := Temp +
   Topology.Signals(Topology.Neurons(J).
       Input_Signals(K))*
   Topology.Weights(Topology.Neurons(J).
       Input_Weights+K);
```

3.2 memcpy/memset

A second key difference was that the C code frequently used memcpy and memset. Marcus Kuhn [4] notes that, in Ada, made these routines "have been made redundant". So, rather than doing:

```
memcpy(input_p3,&topology->signals[0],
   input_size);
```

we instead simply use Ada's array slicing, as:

```
Input_p3 := Topology.signals(0 ..
   Input_Size-1);
```

And for initializing an array to all zero, we use:

```
Topology.Weights_Error_Derivative.all :=
   (others => 0.0);
```

instead of:

```
memset(topology->
   accumulate_weights_error_derivative,0,
   topology->number_weights*sizeof(double));
```

3.3 File I/O

C uses the routines fopen, fseek, fread, fclose to read data from an untyped binary stream. Since our input data consisted of a long sequence of bytes, we chose to use Ada.Direct_IO instantiated on a modular type corresponding to a byte.

```
type Byte is mod 256;
package Byte_IO is new Ada.Direct_IO(Byte);
```

4. MAKING ADA EFFICIENT

Our first test consisted of running the neural network for 400 iterations of forward and backward propagation. We used GCC version 4.1.2 on a Gateway tablet with a 1.86GHz Pentium M processor and 1GB of memory. For Ada, we used GNATPRO 6.0.1. In both cases we turned on optimization (level 2). Table 1 shows the first set of results we obtained.

Table 1: Timings for 400 iterations (first try)

Compiler	Time	Overhead
gcc (-O2)	29 secs	0%
gnat (–O2)	51 secs	75.9%
gnat (-O2 –gnatp)	38 secs	31%

The –gnatp flag turns off run-time checks. This seemed a fairer comparison, as the run-time checks are useful during debugging, but once the code has been debugged, they are no longer required. We were disappointed to discover that Ada ran significantly slower. Not only did we encounter a significant performance loss during the run, but the I/O phase before the iterations began took 8 seconds in the Ada program, and less than a second in the C program. To uncover where the extra time was being spent, we used the GNU Profiler (available for both GCC and GNAT by simply adding the –pg flag during compilation).

4.1 memset/memcpy

Surprisingly, we discovered a huge difference between the C and Ada code on initializing the derivative arrays to zero. The profiler reported the Ada program was spending more time on these routines, and we confirmed that doing 10,000 memsets in C took 9 seconds, while the array initializations in Ada required 16 seconds.

Initializing an array to the floating-point number 0.0 is a special case, as its binary representation is all zeroes. Therefore a memset with 0 can be used in C (memset can only be used when every byte is going to be the same, which will not be the case for most integer or floating-point values).

The GNAT Ada compiler doesn't recognize this special case, and instead generates a loop which writes the floating point number 0.0 to each element of the array. However, we can get a type-safe and fast clearing of the array as follows:

```
procedure Memset (
      Mem_At     : Chars_Ptr;
      With_Value : Integer;
      How_Many   : Size_T);
pragma Import (Intrinsic, Memset,
   "__builtin_memset");

function To_Chars_Ptr is
   new Ada.Unchecked_Conversion (
   System.Address, Chars_Ptr);

procedure Clear (A : in out
   Long_Float_Array) is
begin
   Memset(To_Chars_Ptr(
      A(A'first)'address),
      0,Size_T(8*A'Length));
end Clear;
pragma Inline(Clear);
```

Using this Clear procedure, we are guaranteed never to accidentally do a set with an incorrect size.

We similarly found a builtin_memcpy available in GCC. From this, we created the following safe copy procedure:

```
procedure Memcpy (
      Dest     : Chars_Ptr;
      Source   : Chars_Ptr;
      How_Many   : Size_T);
pragma Import (Intrinsic, Memcpy,
   "__builtin_memcpy");
procedure Copy (
   To : in out Long_Float_Array;
   From : in Long_Float_Array) is
begin
   pragma Assert(To'Length=From'Length);
   Memcpy(To_Chars_Ptr(
      To(To'first)'address),
      To_Chars_Ptr(From(From'first)
         'address),
      Size_T(8*From'Length));
end Copy;
pragma Inline(Copy);
```

Since the amount of overhead was much smaller, we increased the number of iterations to 1400 to make it more evident.

Table 2: Timings for 1400 iterations (change Clear/Copy)

Compiler	Time	Overhead
gcc (-O2)	100 secs	0%
gnat (–O2 –gnatp -gnatN) w/Clear	108 secs	8%
gnat (same) with Clear, Copy	107 secs	7%

Using the builtin_memset was a big improvement; however, the builtin_memcpy didn't make much of a difference. At this point, the overhead is fairly acceptable, but we still wanted to get the Ada code to perform as well as the C implementation.

4.2 Storing pointers

As noted in Section 3.1, we elected to store indices into arrays rather than pointer elements because we needed to keep track both of the element as well as its index. After profiling, we discovered (not surprisingly) that the loop containing the code at the end of Section 3.1 ran slightly slower. This is because each reference to

the signals array requires an addition to a pointer. To solve this, we decided to store both the indices and pointers directly to the elements. Once we did this, the Ada code ran in the same amount of time as the C code.

Table 3: Timings for 1400 iterations (all changes)

Compiler	Time	Overhead
gcc (-O2)	100 secs	0%
gnat (–O2 –gnatp -gnatN)	100 secs	0%

To make the comparison fair, we also added an array of indices to the C code (to avoid having to do the pointer subtraction). This showed no appreciable change in the timing of the C code.

4.3 File I/O

Since we generally run the program for many hours, the extra eight seconds of File I/O at the start was not a big concern. However, we nonetheless pursued the question of how to get the Ada code to do I/O faster. We first considered using Ada.Streams. The default Stream'Read in Ada would expect to read the bounds of an unconstrained array, then its elements. The bounds weren't present in our data files. Since we wanted to maintain the file format of the C program, and we didn't want to go to the trouble of implementing our own stream reader, we chose instead to use the Interface.C_Streams package provided by GNAT. Since we were already using GNAT-specific routines for clearing and copying arrays, this seemed like a reasonable design choice. It involved a very straightforward translation of the C I/O routines. Once completed, the I/O section took less than a second, as did the C version.

5. CONCLUSIONS

Although Ada often has a reputation of being less efficient than C, we discovered that with a small amount of work (less than one man-day of effort for 1226 non-comment non-blank lines of code), we were able to generate an Ada program that was just as efficient as C. Furthermore, we uncovered a subtle bug in the C program, where it was writing one byte past the end of an array.

Consequently, we conclude that Ada provides superior safety to C, while maintaining a similar level of performance.

6. REFERENCES

[1] Tennebo, Frode. (December, 2000). Elegance of Java and the Efficiency of C—It's Ada. *Linux Journal*. [Online]. Available: http://www.linuxjournal.com/article/4342.

[2] Weiskirchner, Marcus. (September, 2003). Comparison of the Execution Times of Ada, C and Java. [Online]. Available: http://www.aicas.com/info/EADS_benchmark_language_comparison.pdf.

[3] Corlan, A. D. "Language Benchmarks." [Online]. Available: http://dan.corlan.net/bench.html.

[4] Kuhn, Markus. "Markus Kuhn's Ada95 page." [Online]. Available: http://www.cl.cam.ac.uk/~mgk25/ada.html.