Abstract. Efficiency of operations in computer applications can have a significant impact on their performance. When intensive calculations are involved, and numerous memory access operations are required, they can cause latency which, in turn, can be reduced by a more efficient implementation. The C programming language offers a strategy for memory organization of arrays that can cause this kind of latency, under specific circumstances. This project presents a possible solution for mitigating this drawback. This is achieved by reproducing the behaviour of these operations in the Fortran programming language, which utilizes a different strategy for memory organization of arrays.

1 Introduction

The C and Fortran programming languages were created with different purposes in mind. The former aims at providing much greater control over memory addressing than the latter, which focuses on performance of mathematical calculations and vector operations [3]. For this reason, the Fortran language differs from the C language in respect to the way array allocation and access operations are performed, specifically multidimensional array operations [2, 5].

The goal of this project is to study the differences in performance between both array types, under different hardware conditions, and when implementing different algorithms that require them. In order to do so, an extension to the C language was developed, as well as the corresponding compiler, combining both array types into a single programming language.

2 State of the Art

In order to undertake a project of this nature, one must first understand the way both programming languages behave when dealing with multidimensional arrays, as well as the basic concepts behind the development of common code compilation utilities.

2.1 Arrays

In both programming languages, an array can be defined as a data structure which aggregates multiple entities of the same type. When necessary, arrays can
have more than one dimension, making it easier for the programmer to organize these entities in more convenient and logical ways \cite{2, 3, 8}. One common example of multidimensional array use is the implementation of matrix operations, or algorithms which resort to matrices.

One-dimensional arrays are handled in identical ways in both the C and Fortran languages. In the case of multidimensional arrays however, each language uses its own strategy to assemble and access their elements \cite{2, 5}.

2.2 The Fortran Language

Multidimensional arrays in Fortran, just like one-dimensional ones, are put together into a single block of memory, whose size in bytes is equal to the size of each element, multiplied by the product of the length of each of the array’s dimensions. This makes it so that, when accessing a given element of the array, i.e. when indexing the array, only the distance to the initial position in memory of the array needs to be determined \cite{24}.

For instance, when loading the value stored in position \((i, j, k)\) of a three-dimensional array of lengths \((X, Y, Z)\), the distance between the initial array position and the desired element’s position is given by:

\[
\delta = i + jX + kXY
\]

Thus, it is simply necessary to perform a series of sums and multiplications in order to calculate this distance. It is interesting to note that, unlike \(X\) and \(Y\), the last dimension size, \(Z\), is not needed to perform this calculation.

2.3 The C language

Unlike the Fortran language, the C language specifies a more elaborate structure for multidimensional arrays. In fact, according to its standard, such an array can be seen as a set of pointers to several other sub-arrays, whose elements may, in turn, point to other sub-arrays of pointers \cite{5}. This chain can be arbitrarily long depending on how many dimensions the array has as a whole. Taking a three-dimensional array as an example once more, it would be organized in C as a hierarchy of sub-arrays with three levels of indirection, one for each dimension. As an advantage, each individual sub-array does not need to have the same length as the others on the same level, which would not be possible according to the Fortran standard \cite{2, 5}.
3.1 Architecture Description

Following the structure of a typical code compilation tool, the developed compiler puts together a pipeline of components: lexical analysis, syntactic analysis, semantic analysis, and code generation [1]. However, the developed tool does not generate binary code, but traditional C language code. This code can then be processed and converted into binary code by a preexisting C language compiler of the programmer's choice, taking advantage of the optimizations it may perform.

Memory allocation is performed by calculating the product of the lengths of each dimension of the array. The resulting value is used to declare a one-dimensional array of the same type. Indexing operations consist in performing the previously mentioned index calculation specified by the Fortran standard.
Special attention was required for operations such as passing these new arrays as function arguments, since all but the last of their dimension lengths need to be known in each scope. Therefore, these values must be passed along with each argument array, following a similar syntax to the one chosen for allocation and indexing operations, as described in the following section.

4 Development Process

The compiler was developed in C, with the aid of Flex, an input processing and tokenizing tool, used for lexical analysis, as well as GNU Bison, a parsing tool, for syntactic analysis \cite{6}. Along with these tools, a lexical description file specifies which tokens the former should recognize, and a context-free grammar is used by the latter in order to parse sequences of these tokens. A C library was developed for the semantic analysis and code generation components.

Lexical analysis consists on recognizing many of the tokens that are part of the traditional C language, especially the ones that were chosen to be part of the set of operations involving the new kind of arrays. These are traditional C type name tokens, variable identifier tokens, square brackets, and semicolons for index separation, among a few others.

As for syntactic analysis, the most relevant grammar rules to note are array declaration and indexing rules, as well as function definition rules, which may include these new arrays in the argument list. Listings 1.1 and 1.2 exemplify the correct use of these operations in a program.

A declaration of one of these new arrays can be done in a similar fashion to traditional C array declarations, namely by indicating its type, followed by the variable name, and finally the lengths of its dimensions in square brackets. However, for the new arrays, these lengths must be separated by semicolons instead of enclosing each one in square brackets.

Also in a similar way to traditional C, indexing an array can be done by indicating the name of a previously declared array variable, followed by square brackets enclosing the index set. Again, each index must be separated from the next by a semicolon.

Function definitions are also of note, since the dimension lengths of each array passed as an argument to a function must be known inside its scope. Otherwise, it would be impossible to calculate the distance from the initial array position to the desired position. Therefore, as previously mentioned, all but the very last dimension length must be specified for each of the new arrays in a function argument list.
#include <stdio.h>

int main() {
    int ext_array[3; 5; 10];
    int trad_array[4][6][11];

    /* Initialization of array elements ...
    */

    printf("%d\n", ext_array[2; 4; 9] + trad_array[3][5][10]);

    return 0;
}

Listing 1.1: Example of array declaration and indexing operations.

#include <stdio.h>
#include <stdlib.h>

int foo(int arg1, int *arg2[30; 50;], double arg3) {
    /* Indexing the second argument: last dimension * length is not needed. */

    int *ptr = arg2[29; 49; 99];

    return ptr == NULL ? arg1 : arg3;
}

int main(int argc, char **argv) {
    int *arg_array[30; 50; 100];
    arg_array[29; 49; 99] = NULL;

    return foo(0, arg_array, 3.14);
}

Listing 1.2: Example of an array as a function argument.
5 Evaluation

5.1 Evaluation Method

Given the difference in memory organization and indexing operations in C and Fortran arrays, it was expected that the performance of each type of array would differ depending on how the Central Processing Unit (CPU) handles multiplications and memory address loading operations. Most Acorn RISC Machine (ARM) architectures, for instance, are known to take several CPU cycles in order to process a multiplication instruction. Therefore, the programs produced by the developed compiler needed to be run under a few varying test conditions, namely different CPU architectures and usage in different array-based algorithms.

Evaluation of the developed C language extension consisted in measuring and comparing the execution times of three different matrix-processing algorithms. Each algorithm was implemented in two different versions: one using only traditional C language arrays, and the other using only language extension arrays. The chosen algorithms were the sum of two matrices, the product of two matrices, and the Leibniz formula for the determinant of a square matrix.

These algorithms were run on three different CPU architectures, namely one ARMv6 and two Intel x86-64 processors. Moreover, both versions of each implemented algorithm were given input matrices of varying sizes to process. A total of ten execution times were measured for each set of test conditions, and the average of those times was then calculated for each set. Finally, these values were used to determine the performance ratio between both program versions under the same test conditions.

5.2 Discussion

It is important to note that the aforementioned algorithms operate differently on the input matrices. The first one iterates over the rows of both operand matrices and stores the sum of each pair of values in the corresponding position of a third matrix, which holds the result of the algorithm at the end. The matrix product algorithm traverses each matrix in different directions, iterating over the rows of the first operand matrix, and the columns of the second one. Finally, the matrix determinant algorithm accesses the elements of the input square matrix in a non-linear order, jumping between different permutations of column index values in each iteration.

The obtained results varied significantly among different sets, with the sum of matrices revealing the poorest performance results for the new arrays of the developed language extension. In fact, traditional C arrays proved to be about three times faster than the new ones. The reason for this is the principle of spatial locality, which is a common optimizing strategy in CPUs [9]. This optimization is based on the empirical principle that an address loading operation is likely to be soon followed by others to adjacent memory addresses. For this reason, cache memory is usually filled in advance with the contents of those adjacent
memory addresses in order to increase the performance of the following loading operations [9]. Since indexing the new kind of arrays requires an index calculation before an address can be loaded, significant amounts of latency are accumulated as a result of the several iterations the algorithm performs. Naturally, traditional C arrays do not generate this kind of latency. Its impact was especially noticeable in the set of tests run on the ARMv6 CPU, since the computational cost of multiplication operations is much heavier on an ARM architecture than on an Intel one.

As for the matrix product, results went slightly in favor of the new arrays on both Intel x86-64 CPUs. This is because of the much higher number of cache misses that are caused by traversing one of the matrices in a column-first order. In the case of the ARMv6 CPU however, the traditional arrays still proved to have slightly better performance than the new ones.

Finally, results for the Leibniz formula algorithm showed that the new arrays can perform better than traditional ones when the number of cache misses is high enough. Every single CPU architecture achieved a positive comparison ratio in favor of the new arrays, although the increase was only about 3.5% on average.

6 Conclusion

As a result of this project, a compiler for an extended version of the C programming language was developed, adding a new kind of arrays which can slightly improve a program's performance, given the right hardware conditions. However, their use is not advised for algorithms that heavily benefit from optimizations based on the principle of spatial locality.

Despite the efforts made, several improvements can be made to the developed tool, such as combining and packaging it with a traditional C compiler. This would eliminate the need of intermediate code generation, namely traditional C code. Furthermore, several features of traditional arrays are not yet supported by the new ones, such as including them as members of data structures or having function pointers as array elements.

Besides implementing the remaining features, it would be useful to include a compilation option that would convert every new array into a traditional one at compile time. This would make it easy for the programmer to switch between both alternatives without the need to change any code, since the use of the new arrays is not always recommended.

Finally, it would be interesting to test the impact of compiler optimizations on test results, such as operation strength reduction on multiplication operations, or the absence of the strict aliasing rule. Given the obtained results, it would also be interesting to expand this sort of study to other programming languages, such as C++.
Bibliography