Clash of the Cache

*Muhammad Mahmood, Roman Millem, Easton Culver*

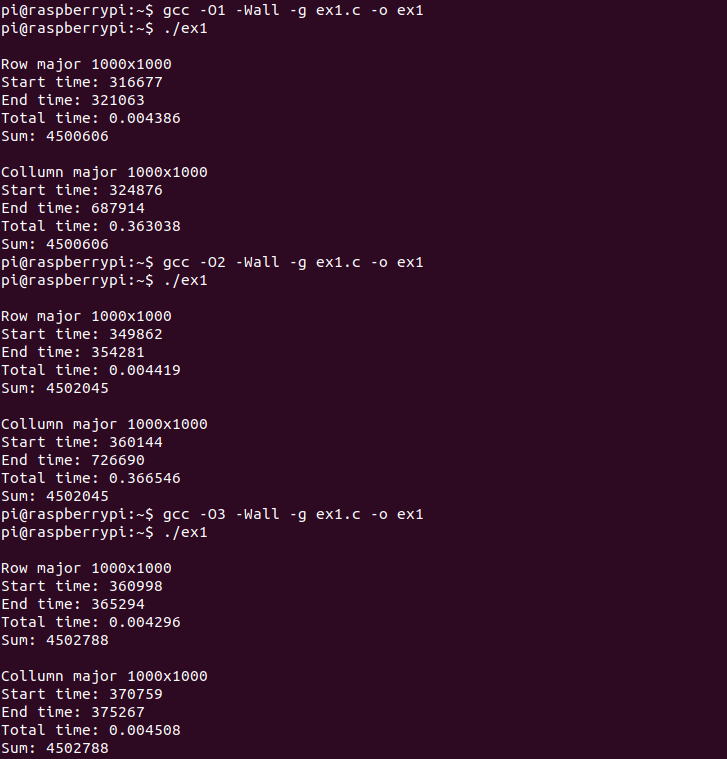
CECS525

Abstract

The main objective is to discover the optimization methods that can be used when interacting with cache memory. With utilizing multiple different iterations of row versus column major computing it could be seen which versions would create the shortest amount of time while parsing the data of a large number set. The end goal was to see how each one compared to the others given the exact same amount of numbers to deal with, as well as different access techniques to the stored data. The first of these examples was simply creating the individual row-major and column-major functions and seeing how they compared to each other with computer optimization still turned on. After this was measured, the computer optimization would be turned off and the same test would be run and recorded. The third iteration would be using the floop interchange within gcc and see how that would affect the results. The final trial was used with jumping the accesses by a total of 4, and then 20 memory spaces to see how it affected both the column and row major times.

Body

When first tested, the matrix summation program ran row major summation and column major summation with the same run time. This held true when tested on a 20x20, 200x200, and 1000x1000 matrix (note that testing with the specified 2000x2000 matrix resulted in a segmentation fault due to the matrix size). The row and column major algorithms were equally fast due to the default optimization in the gcc compiler. After continuous testing, this occurred at optimization level 3. When compiled with optimization level 1 and 2, row major executed significantly faster than the column major summation (shown in the screenshot below) and the run times were close to optimization level 0.



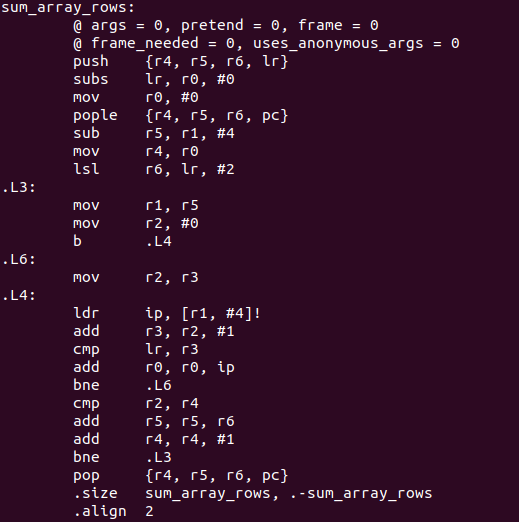
An assembly listing was generated using optimization level 3. The generated row major and column major summation algorithms are near identical, occasionally using different registers and pointers. This is because the compiler converted the column major algorithm to row major. Comparatively, the generated assembly listings using optimization level 0 are larger. The unoptimized code manages the stack frequently using stack and frame pointers. The ‘MUL’ instruction is also used in the unoptimized code which can significantly increase run time. Data is exchanged through many levels of register moves. The optimized code makes use of the pop and push instructions to manage the stack and uses condition codes to avoid many compares and branches.

When compiled using the -floop-interchange optimization, no changes were noticed. The gcc configuration file was not found in the virtual file system. So when applying this optimization, gcc was not configured with --with-ppl nor --with-cloog. Neither of these configurations were enabled with the given build of gcc. This was checked using the command ‘gcc -v’ to list the current version and configuration of the compiler. From further research, this optimization seems to be one of the most important optimizations when applied to this experiment. The -floop-interchange optimization is responsible for converting a column major algorithm to a row major algorithm. This results in a much faster run time as the matrix is accessed contiguously in memory for the row major matrix summation algorithm. The assembly listing generated using this optimization is identical to the assembly code generated from optimization level 0. From these results, it can be assumed that the -floop-interchange optimization was incorrectly implemented as a result of the inability to reconfigure the given gcc build.

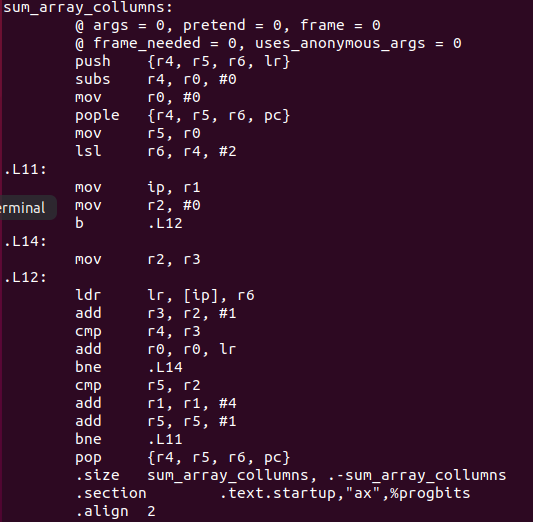
Both summation algorithms were then rewritten to access every fourth memory location and then rewritten again to access every 20th. Both of these changes resulted in a faster run time for the column major summation algorithm. The fourth location access algorithm resulted in a faster row major execution time than the column major.The modulo 20 access pattern resulted in a slight execution time increase for row major and a column major execution time that was faster than row major execution. The new algorithms work by first accessing the first location in the matrix. For the column major algorithm, the second location accessed is the fourth row, first column. When reaching the end of the first column, the next location accessed is four away from the previous. For example, for a 20x20 array, if the last location accessed is the first column, row 18, then the next location accessed will be column 2, row 2. For the first rewrite to access every fourth location, this process is looped 4 times incrementing the start position each time to ensure summation works correctly. The changes in run time may be due to multiple differences. Standard column major memory access is costly, so accessing a location offset decreases the run time because fewer noncontiguous memory jumps are made. The likelihood of a cache hit increases as the offset grows. Contiguous memory locations are more easily accessed using previously cached locations. The increase in row major execution time for the modulo 20 access pattern is due to a larger offset resulting in less contiguous memory access due to more frequent row jumps than the standard row major sequential access algorithm. The non-sequential access patterns can be thought of as a slight trend toward the opposite algorithm. Meaning that memory access in the column major algorithm trends more towards row major, and vice versa.

Source Code (Software)

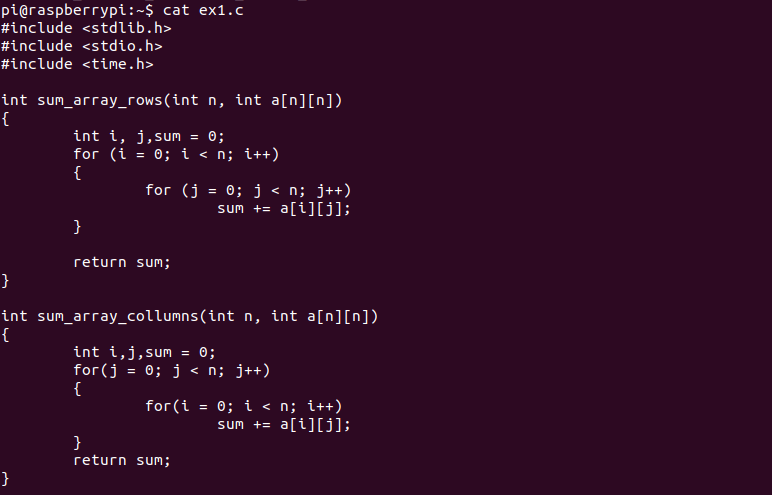
Exercise 1 Assembly file row major algorithm



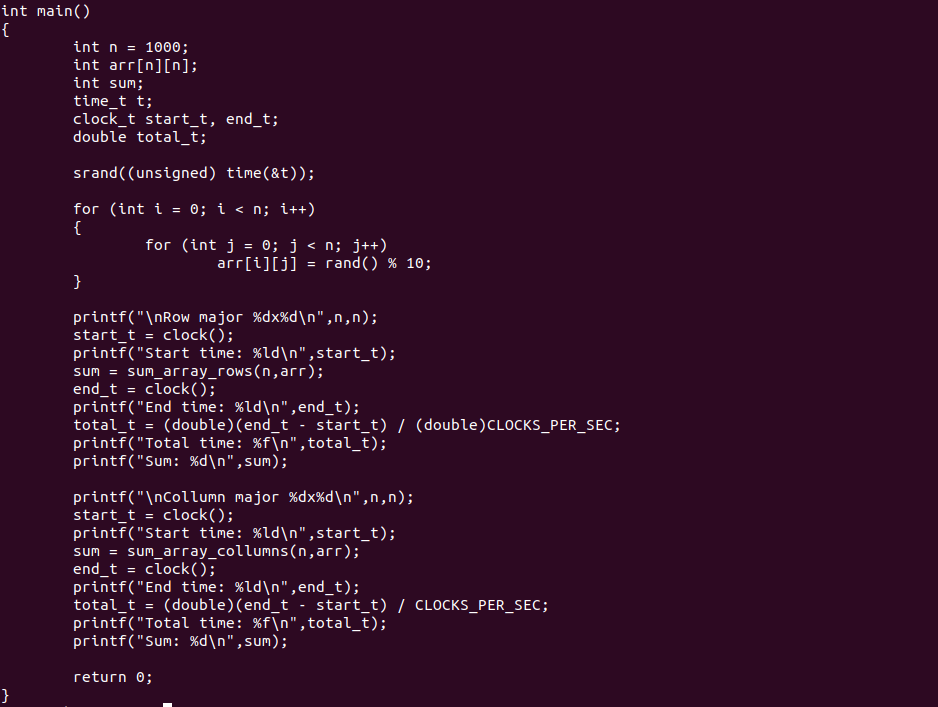
Exercise 1 Assembly file column major algorithm



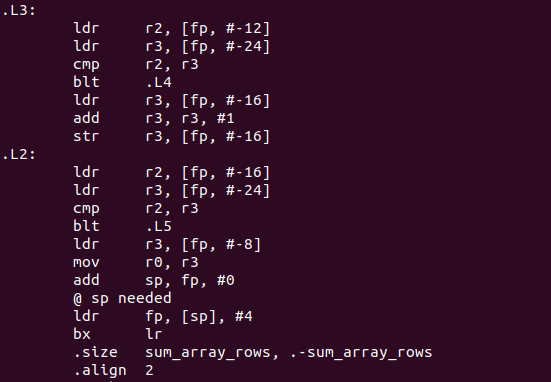
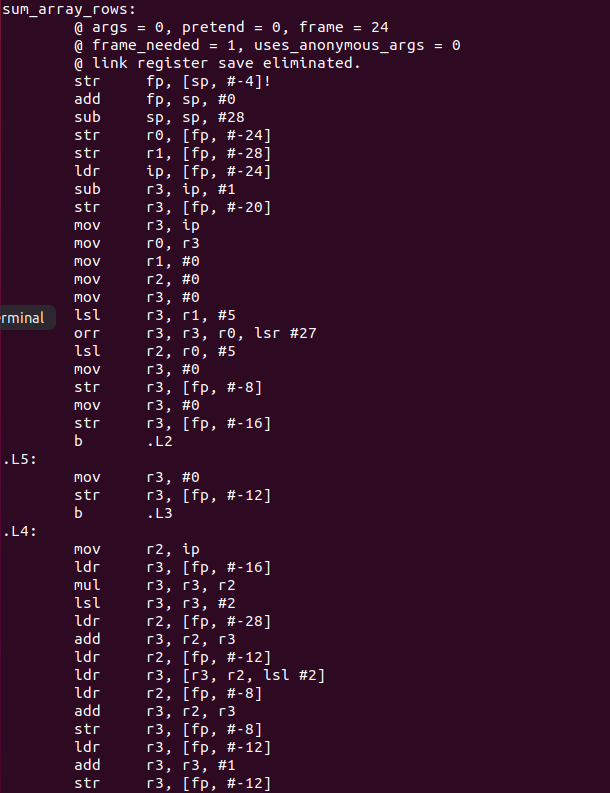
C file sequential access row and column major matrix summation

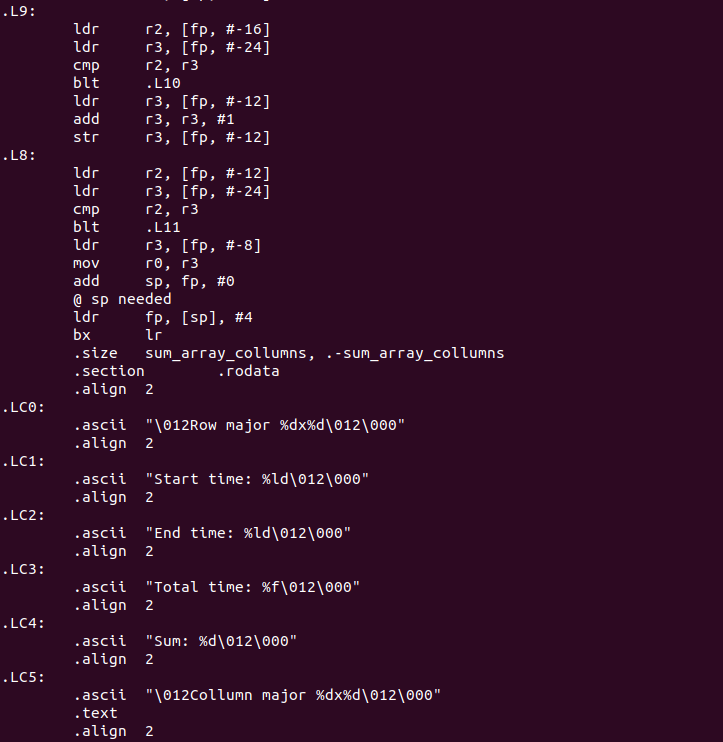
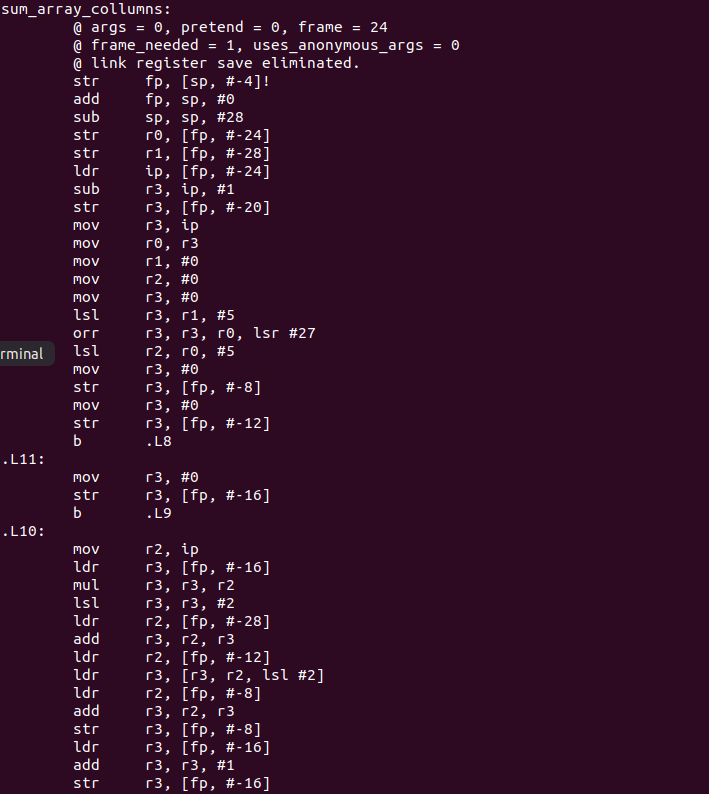


C file time benchmarking is included

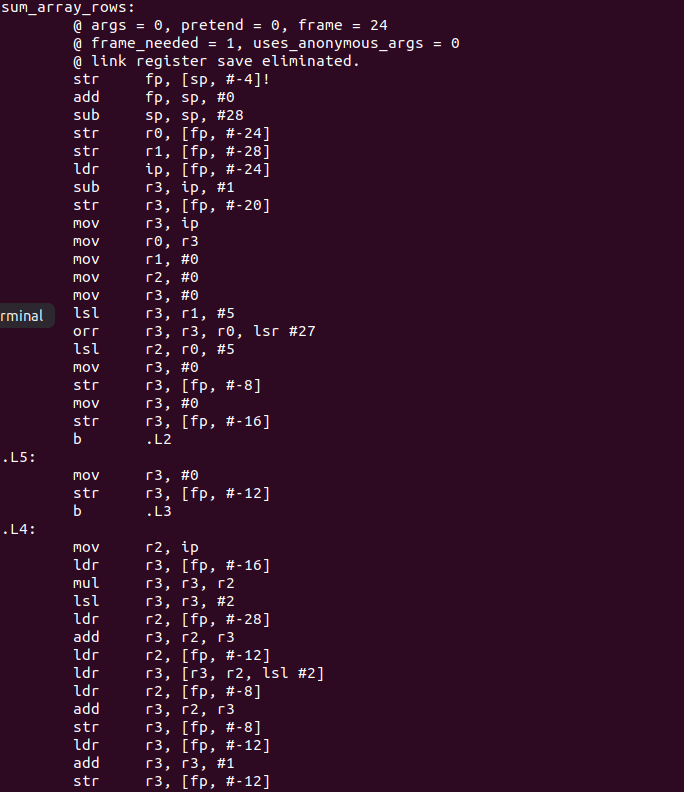


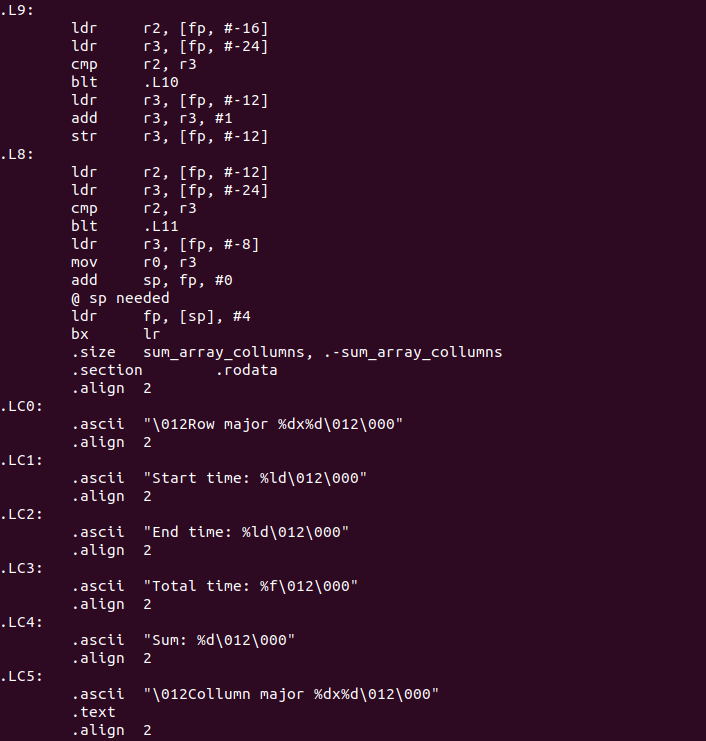
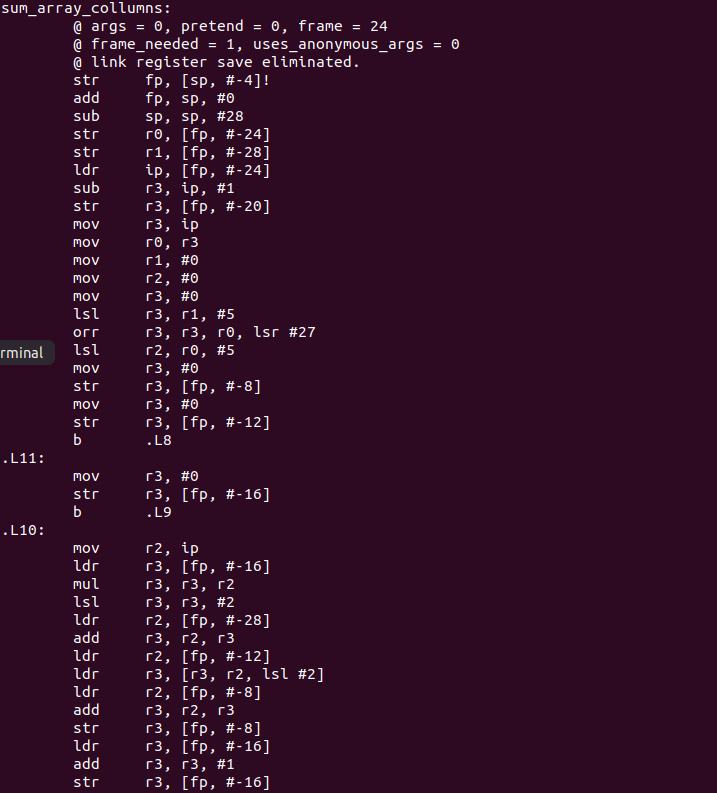
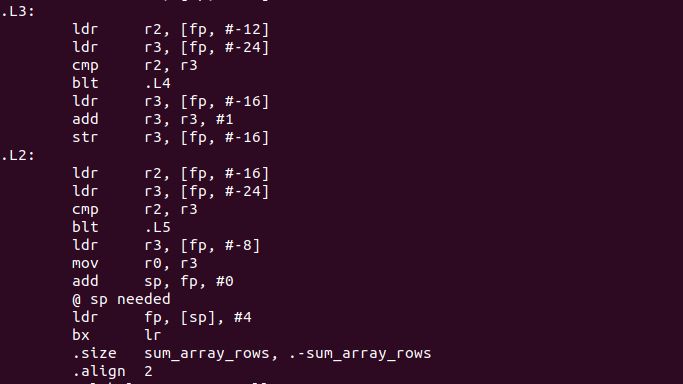
Exercise 2 Assembly file row major algorithm



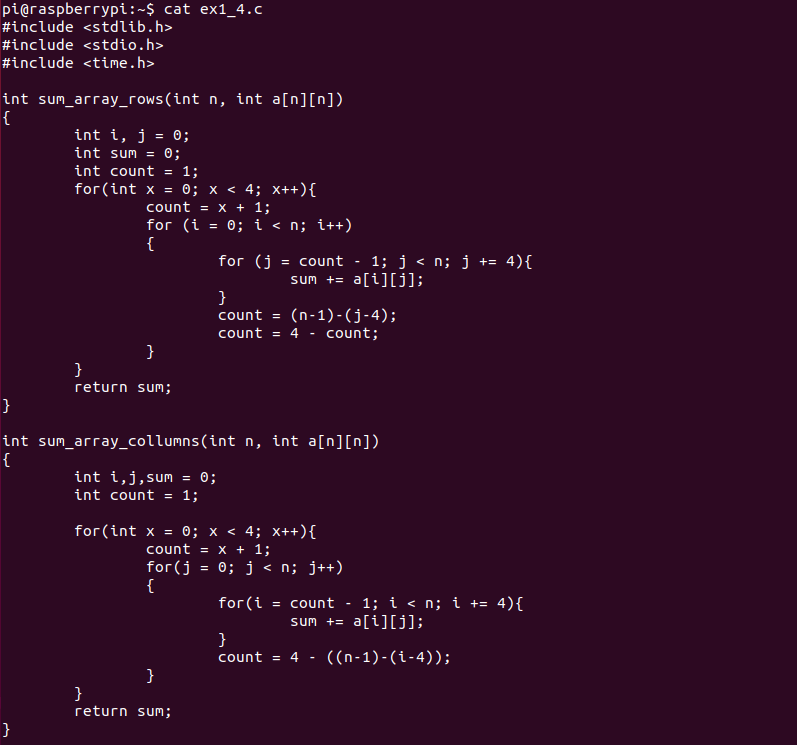
Exercise 2 Assembly file column major algorithm

Exercise 3 Assembly file row major algorithm

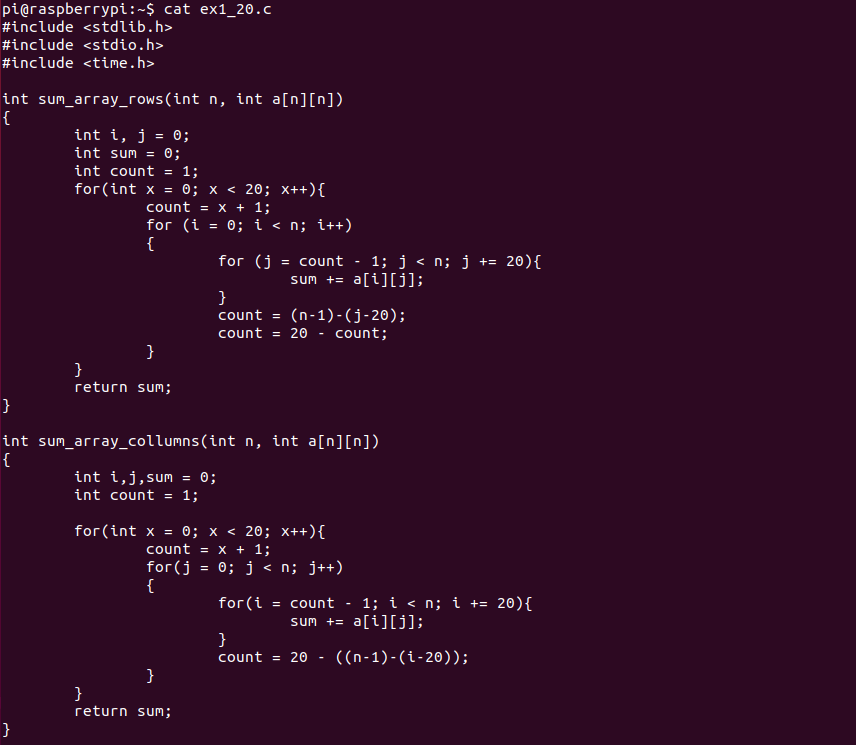




C file accessing every 4th value



C file accessing every 20th value



Schematics (Hardware)

N/A

Analysis

The initial code focuses on the concept of computations within a matrix utilizing row major or column major. The task of building a matrix with random numbers was simple further implementing two simple loops to look at indexes of the 2-d matrix. Further computatue the summation by going row to row or column to column, while it is faster to move from left to right for there are smaller jumps required in memory. Additionally, benchmarking is included to represent computation speeds of the calculations.

Without optimization the computations would take roughly .5 s, while with optimization the computations would take almost 100 times less. While the floop interchange optimization took the same time as the optimized solution.

The -floop-interchange optimization was likely applied incorrectly. The gcc gnu optimization document specifies two configuration options that must be added to enable -floop-interchange. We were unable to locate the gcc configuration file for the given build to add these options. So when the code was compiled using the -floop-interchange optimization, the resulting program was identical to optimization level 0.

Conclusion

With the main goal of this exercise being a deeper understanding and grasp of memory optimization on computers that have cache memory it was largely achieved. The first example not entirely showing how to manually optimize it, but that modern day computers on their own automatically optimize cache memory access so that in either case it should produce the same result. The next example showed just how much of an effect that not having optimization could have on the computing time. Then finally the last two examples showed different ways that could optimize the memory accesses in order to reduce the time it took for the process to complete. Since each of the individual parts of the entire process helped teach a different portion of memory optimization, the objective of learning was more than met.

References

Video Demo - <https://youtu.be/4B_O7eUSgwI>

<https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Optimize-Options.html>