John Olgin
olginj@oregonstate.edu
CS 475 – Spring 2019
Project #7B
Autocorrelation using CPU OpenMP, CPU SIMD, and GPU {OpenCL or CUDA}

Description

This project will be performing an autocorrelation procedure on an array of numbers from a text file. I'll be using four different methods of parallelization to achieve the wanted result: single-thread OpenMP, 16-thread OpenMP, SIMD and OpenCL. All four were developed on macOS Mojave Version 10.14 using Sublime Text. All testing was done on rabbit to maintain consistency between comparisons. Both OpenMP implementations were included in the same program.

The primary objective is to collect performance measurements and compare across the four different methods. To do this, I'll be using MegaShifts/s as my unit of measurement. This unit will be used for all four. It will be calculated as follows:

MegaShifts/s = size*size / (timeStart – timeEnd) / 1000000

Size is the size of the array determined by the input from the text file. The times are taken before the calculations occur and then immediately after. Dividing by 1 million will give me the resulting number of calculations in units with a size of one million.

Provided below are the performances of each method used to execute the autocorrelation. Also, there's a graph depicting the different sums in the array for index values 1-512. I will discuss the characteristics of this graph and explain the differences in the methods' performances below. Since the raw data for the sums array is rather lengthy, it will be omitted from this pdf.

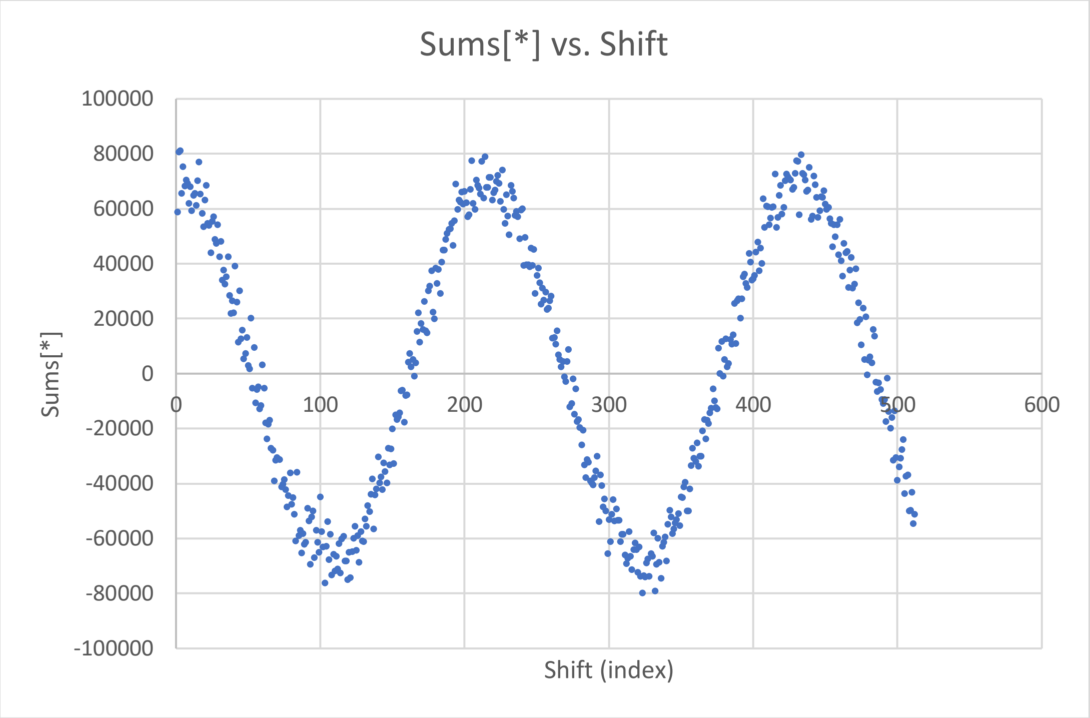
Performance by Method

Method	Performance (MegaShifts/s)
OpenMP–1 thread	138.03
OpenMP–16 threads	879.44
SIMD	1771.29
OpenCL	48904.18

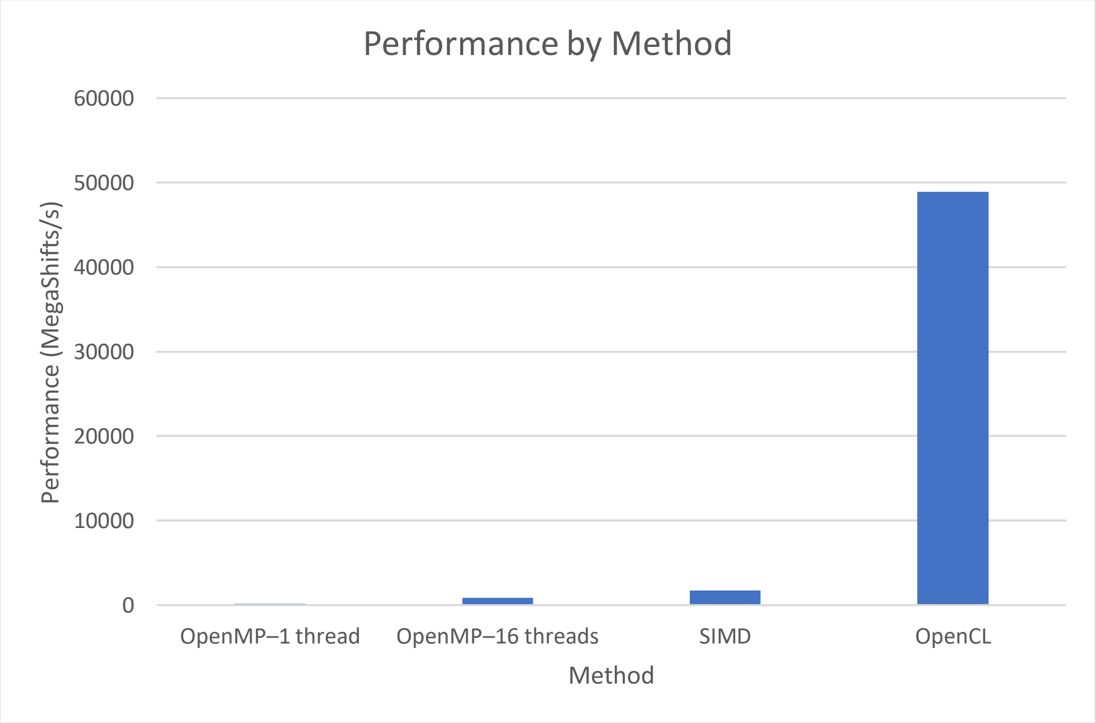
Graphs

Below

Graph 1



Graph 2



By looking at graph 1, the sine wave's period can be estimated to be around 219. This is the number of index values between the two maxima before the graph starts declining once again. However, it appears that this value may vary between each maximum. If we look at the first two spikes in the graph, the two maximum values have a period of about 211, so the period grew by about eight during the second period. The two values are still relatively close, so I expect that if we let the graph continue, it would peak again after about the same number of index values.

There was a large difference in performance between the methods used to perform the auto correlation. OpenCL was by far the fastest. The other three methods weren't even close. In fact, if you look at Graph 2, you can hardly tell they're even represented. However, the general pattern is an increase in performance from one method to another. In order from slowest to fastest, it's single-thread OpenMP, 16-thread OpenMP, SIMD and then OpenCL. While both the 16-thread OpenMP and SIMD methods performed multiple times faster than the method right before, OpenCL registered an enormous increase over the rest, dwarfing any increases in performance between the first three.

The changes in performance are intuitive. With a data size of over 32K and a single set of instructions, I can easily see where the program has ample room for parallelization. So, we'll look at the changes from method to method. First, my 16-thread OpenMP program had a speedup of about 6.3 over the single-thread OpenMP program. This was solely due to the dedication of 15 additional threads to perform the instructions on the given array. Next, the SIMD program performed about twice as well as the 16-thread OpenMP program. It also performed about 13 times better than the single-thread OpenMP program. I believe its primary advantage was the vectorization, where it executes the same instructions on four pieces of data simultaneously. Furthermore, the SIMD code is written in assembly language. This will inherently provide a boost in performance since assembly language is much more similar to machine code. Thus, less work has to be done to convert the assembly code than for C++, as used in OpenMP. Finally, the OpenCL program performed drastically better for several reasons. A primary reason is that it avoids looping through the entire array to perform the necessary instructions. Instead, it utilizes works groups that can simultaneously retrieve specific index values from an array with methods like get_global_id(). This eliminates the need for almost any sequential instructions and maximizes the level of parallelization that can be used on the dataset. The other 3 methods, despite their increased efficiencies, spent much more time waiting for threads to become available to perform instructions on the data than did OpenCL.