Name: Ramon Arambula

Lab Topic: Performance Optimization (Memory Hierarchy) (Lab #:6)

<u>Part 1 - Data storage in memory</u>

Question #1:

Describe how a two-dimensional array is stored in one-dimensional computer memory.

Answer:

In 1D memory, the computer stores the leading dimension (N), and the secondary dimension(M) in memory. Therefore it can specify in one dimension the step size/ bit size that corresponds to the first dimension.

Question #2:

Describe how a three-dimensional array is stored in one-dimensional computer memory.

Answer:

A 3D array is stored similar to a 2D array, but it just adds a depth layer, which would correspond to a block of memory, and search that block of memory with N * M dimensions.

Question #3:
(3) Copy and paste the output of your program into your lab report, and be sure that the source code and Makefile is included in your compressed folder to submit.
Answer:
Printing 2D Array where output is [Val: Address]
    1: c991d018 2: c991d01c 3: c991d020 4: c991d024 5: c991d028

    6: c991d02c 7: c991d030 8: c991d034 9: c991d038 10: c991d03c

    11: c991d040 12: c991d044 13: c991d048 14: c991d04c 15: c991d050
How 2d memory is stored in computer's 1d memory storage
    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Printing 3D Array where output is
Depth #
    [Val: Address]
Depth 0:
    1: c991d058 2: c991d05c 3: c991d060 4: c991d064 5: c991d068 6: c991d06c
7: c991d070
    8: c991d074 9: c991d078 10: c991d07c 11: c991d080 12: c991d084 13:
c991d088 14: c991d08c
    15: c991d090 16: c991d094 17: c991d098 18: c991d09c 19: c991d0a0 20:
c991d0a4 21: c991d0a8
    22: c991d0ac 23: c991d0b0 24: c991d0b4 25: c991d0b8 26: c991d0bc 27:
c991d0c0 28: c991d0c4
    29: c991d0c8 30: c991d0cc 31: c991d0d0 32: c991d0d4 33: c991d0d8 34:
c991d0dc 35: c991d0e0

Depth 1:
    36: c991d0e4 37: c991d0e8 38: c991d0ec 39: c991d0f0 40: c991d0f4 41:
c991d0f8 42: c991d0fc
    43: c991d100 44: c991d104 45: c991d108 46: c991d10c 47: c991d110 48:
c991d114 49: c991d118
    50: c991d11c 51: c991d120 52: c991d124 53: c991d128 54: c991d12c 55:
c991d130 56: c991d134
    57: c991d138 58: c991d13c 59: c991d140 60: c991d144 61: c991d148 62:
c991d14c 63: c991d150
    64: c991d154 65: c991d158 66: c991d15c 67: c991d160 68: c991d164 69:
c991d168 70: c991d16c

Depth 2:

71: c991d170 72: c991d174 73: c991d178 74: c991d17c 75: c991d180 76: c991d184 77: c991d188

78: c991d18c 79: c991d190 80: c991d194 81: c991d198 82: c991d19c 83: c991d1a0 84: c991d1a4

85: c991d1a8 86: c991d1ac 87: c991d1b0 88: c991d1b4 89: c991d1b8 90: c991d1bc 91: c991d1c0

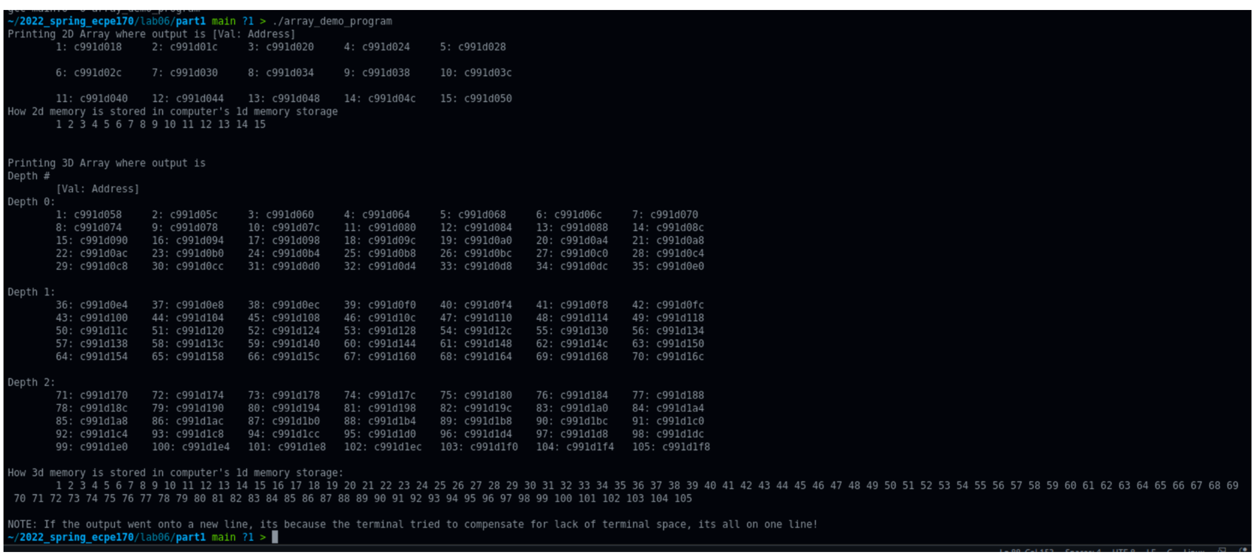
92: c991d1c4 93: c991d1c8 94: c991d1cc 95: c991d1d0 96: c991d1d4 97: c991d1d8 98: c991d1dc

99: c991d1e0 100: c991d1e4 101: c991d1e8 102: c991d1ec 103: c991d1f0 104: c991d1f4 105: c991d1f8

How 3d memory is stored in computer's 1d memory storage:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105

NOTE: If the output went onto a new line, its because the terminal tried to compensate for lack of terminal space, it's all on one line!



```
~/2022_spring_ecpe170/lab06/part1 main ?1 > ./array_demo_program
Printing 2D Array where output is [Val: Address]
        1: c991d018     2: c991d01c     3: c991d020     4: c991d024     5: c991d028

        6: c991d02c     7: c991d030     8: c991d034     9: c991d038     10: c991d03c

        11: c991d040    12: c991d044    13: c991d048    14: c991d04c    15: c991d050
How 2d memory is stored in computer's 1d memory storage
        1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


Printing 3D Array where output is
Depth #
        [Val: Address]
Depth 0:
        1: c991d058     2: c991d05c     3: c991d060     4: c991d064     5: c991d068     6: c991d06c     7: c991d070
        8: c991d074     9: c991d078     10: c991d07c    11: c991d080    12: c991d084    13: c991d088    14: c991d08c
        15: c991d090    16: c991d094    17: c991d098    18: c991d09c    19: c991d0a0    20: c991d0a4    21: c991d0a8
        22: c991d0ac    23: c991d0b0    24: c991d0b4    25: c991d0b8    26: c991d0bc    27: c991d0c0    28: c991d0c4
        29: c991d0c8    30: c991d0cc    31: c991d0d0    32: c991d0d4    33: c991d0d8    34: c991d0dc    35: c991d0e0

Depth 1:
        36: c991d0e4    37: c991d0e8    38: c991d0ec    39: c991d0f0    40: c991d0f4    41: c991d0f8    42: c991d0fc
        43: c991d100    44: c991d104    45: c991d108    46: c991d10c    47: c991d110    48: c991d114    49: c991d118
        50: c991d11c    51: c991d120    52: c991d124    53: c991d128    54: c991d12c    55: c991d130    56: c991d134
        57: c991d138    58: c991d13c    59: c991d140    60: c991d144    61: c991d148    62: c991d14c    63: c991d150
        64: c991d154    65: c991d158    66: c991d15c    67: c991d160    68: c991d164    69: c991d168    70: c991d16c

Depth 2:
        71: c991d170    72: c991d174    73: c991d178    74: c991d17c    75: c991d180    76: c991d184    77: c991d188
        78: c991d18c    79: c991d190    80: c991d194    81: c991d198    82: c991d19c    83: c991d1a0    84: c991d1a4
        85: c991d1a8    86: c991d1ac    87: c991d1b0    88: c991d1b4    89: c991d1b8    90: c991d1bc    91: c991d1c0
        92: c991d1c4    93: c991d1c8    94: c991d1cc    95: c991d1d0    96: c991d1d4    97: c991d1d8    98: c991d1dc
        99: c991d1e0    100: c991d1e4   101: c991d1e8   102: c991d1ec   103: c991d1f0   104: c991d1f4   105: c991d1f8

How 3d memory is stored in computer's 1d memory storage:
        1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105

NOTE: If the output went onto a new line, its because the terminal tried to compensate for lack of terminal space, its all on one line!
~/2022_spring_ecpe170/lab06/part1 main ?1 >
```

Part 2 - Memory Locality

Question #4:

Provide an Access Pattern table for the sumarrayrows() function assuming ROWS=2 and COLS=3.

The table should be sorted by ascending memory addresses, not by program access order.

Answer:

Memory Address	0	4	8	12	16	20
Memory Contents	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Program Access order	1	2	3	4	5	6

Question #5:

Does sumarrayrows() have good temporal or spatial locality?

For your answer to receive full credit, you must discuss the locality of both the array itself, and the scalar variables such as i that are present in the function.

Answer:

sumarrayrows() has good spatial locality since the elements are being read sequentially. The scalar variables: i, j, sum — are accessed sequentially which allows it to have good spatial locality. However, the temporal locality isn't that good since it only accesses one address at a time.

Question #6:

Provide an Access Pattern table for the sumarraycols() function assuming ROWS=2 and COLS=3.

The table should be sorted by ascending memory addresses, not by program access order.

Answer:

Memory Address	0	4	8	12	16	20
Memory Contents	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

Program Access order	1	3	5	2	4	6

Question #7:

Does sumarraycols() have good temporal or spatial locality?

For your answer to receive full credit, you must discuss the locality of both the array itself, and the scalar variables such as i that are present in the function.

Answer:

The spatial locality is bad, since it's trying to access different columns in different row locations. You can think of every row as a cache, so the first time a row is accessed the spatial locality isn't good, but every subsequent access in that row is going to be good. However, since you're accessing different rows every iteration of sumarraycols(), then the benefit that the cache has on spatial locality does not exist.

Part 3 - Performance Measurement

Question #8:

Inspect the provided source code. Describe how the *two*-dimensional arrays are stored in memory, since the code only has one-dimensional array accesses like: a[element #].

Answer:

Since the 2D arrays are stored sequentially, the way that the program accesses two different memory locations is by referencing the row # relative to i. Therefore, the code is accessing multiple locations — which represent different rows — in the 1D memory space by accessing (iterator * rows + j).

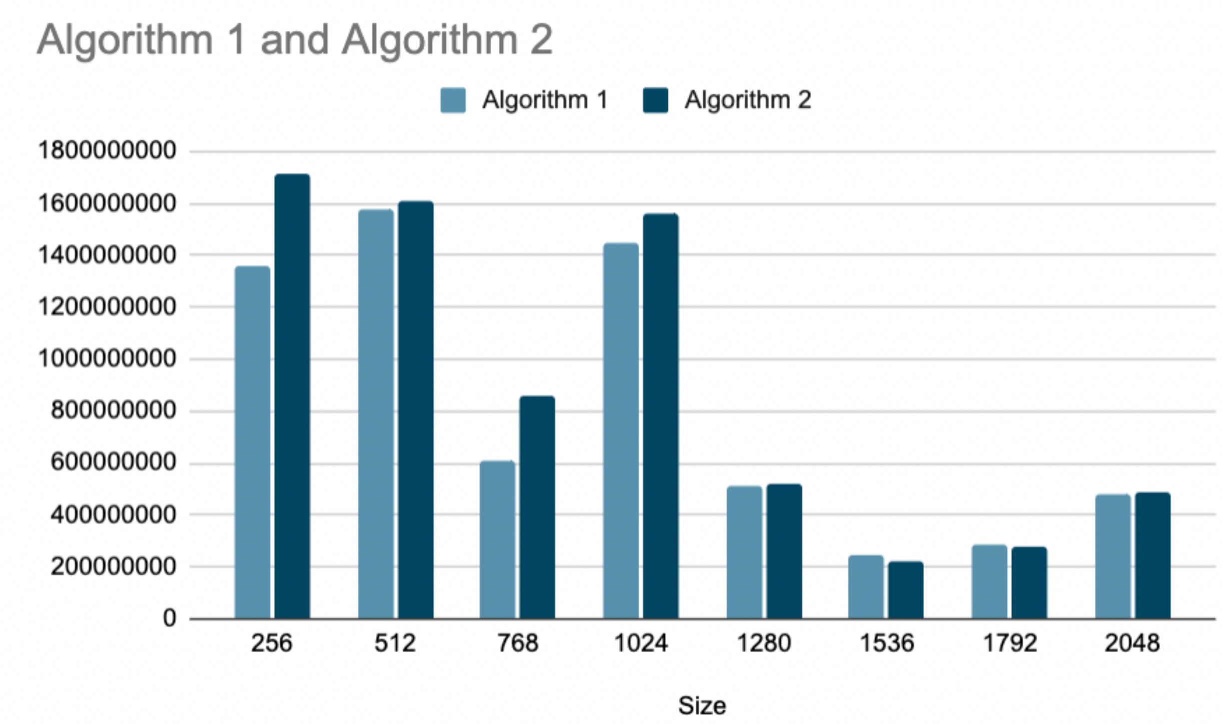
Question #9:

After running your experiment script, create a **table** that shows floating point operations per second for both algorithms at the array sizes listed in Table 2.

Answer:

Size	Algorithm 1	Algorithm 2
256	1360000000	1710000000
512	1580000000	1610000000
768	609000000	858000000
1024	1450000000	1560000000
1280	515000000	517000000
1536	245000000	225000000
1792	285000000	281000000
2048	483000000	490000000

Question #10:
After running your experiment script, create a **graph** that shows floating point operations per second for both algorithms at the array sizes listed in Table 2.
Note: No credit will be given for sloppy graphs that lack X and Y axis labels, a legend, and a title.
Answer:



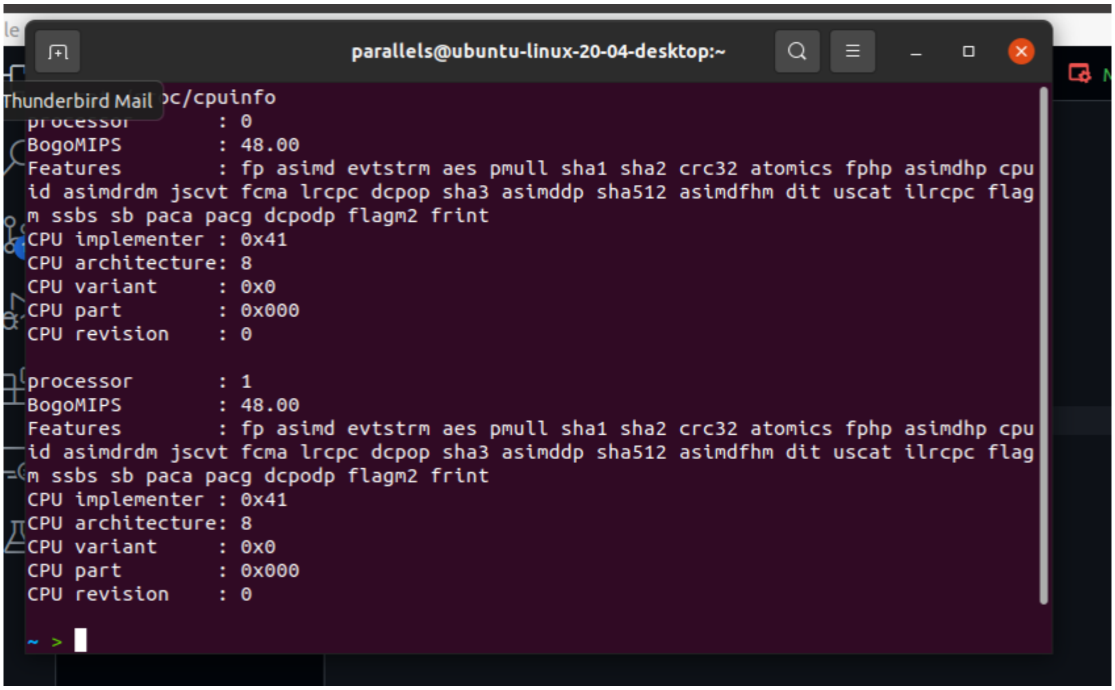
Question #11:
Be sure that the script source code is included in your compressed folder to submit.
Answer:
It is

Part 4 - Memory Mountain

Question #12:

Place the output of /proc/cpuinfo in your report. *(I only need to see one processor core, not all the cores as reported)*

Answer:



Question #13:

Based on the processor type reported, obtain the following specifications for your CPU from cpu-world.com or cpudb.stanford.edu

You might have to settle for a close processor from the same family. Make sure the frequency and L3 cache size match the results from /proc/cpuinfo!

(a) L1 instruction cache size

(b) L1 data cache size

(c) L2 cache size

(d) L3 cache size

(e) What URL did you obtain the above specifications from?

Answer:

Note: I have a MacBook Pro with the new M1 Pro processors

a. 192 KB
b. 128 KB
c. 24 MB for performance cores, 4 MB for efficiency cores
d. 24 MB
e. https://en.wikipedia.org/wiki/Apple_M1_Pro_and_M1_Max

Question #14: NOTE: I USED PROFESSORS RESULTS
Why is it important to run the test program on an idle computer system?
Explain what would happen if the computer was running several other active programs in the background at the same time, and how that would impact the test results.
Answer:
If the test program is run on a computer that isn't idle, i.e. it has a lot of background programs running, then all of the running programs are competing for system resources. This causes the program's run results to be skewed as they are not entirely accurate due to dynamic resource allocation which might interfere with the results

Question #15:
What is the size (in bytes) of a data element read from the array in the test?
Answer:
Since in init_data() in mountain.c uses doubles, each element is a double with size of 8 bytes

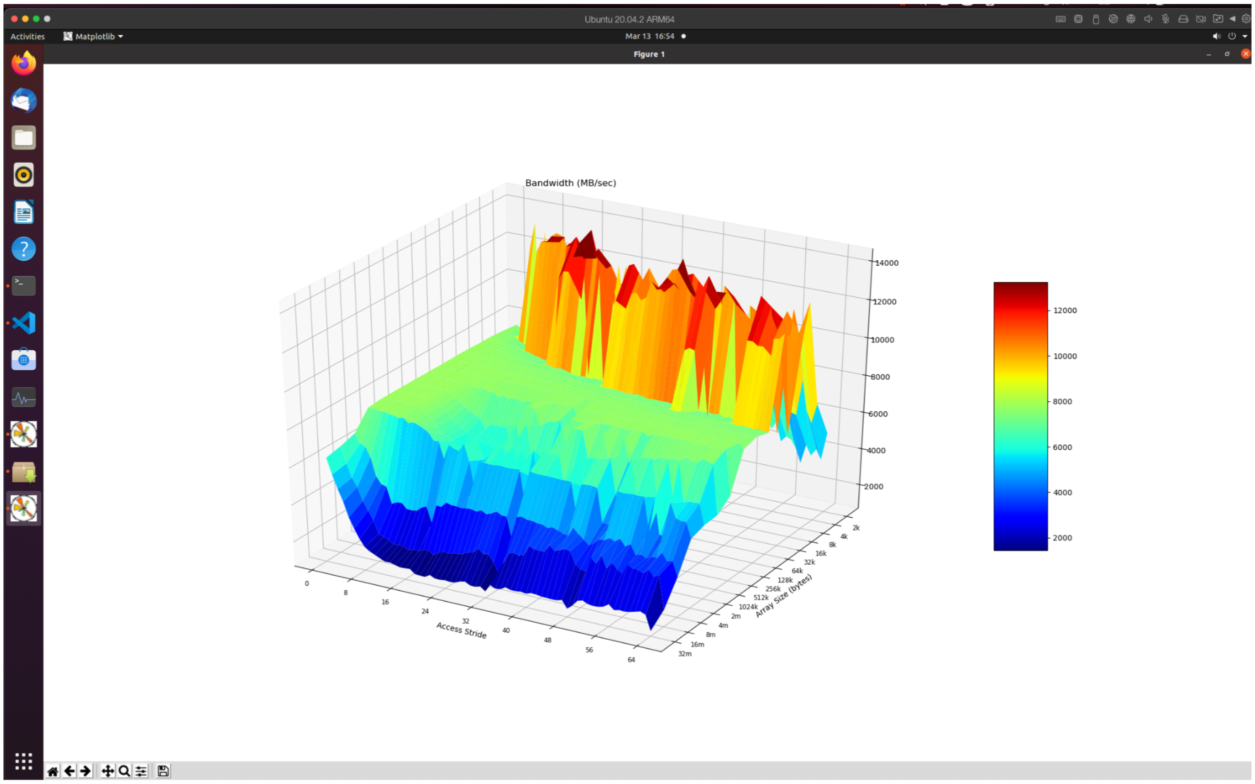
Question #16:
What is the range (min, max) of *strides* used in the test?
Answer:
Range relative to strides used is 0 to 64

Question #17:
What is the range (min, max) of *array sizes* used in the test?
Answer:
Range relative to array size used is 2k to 32m

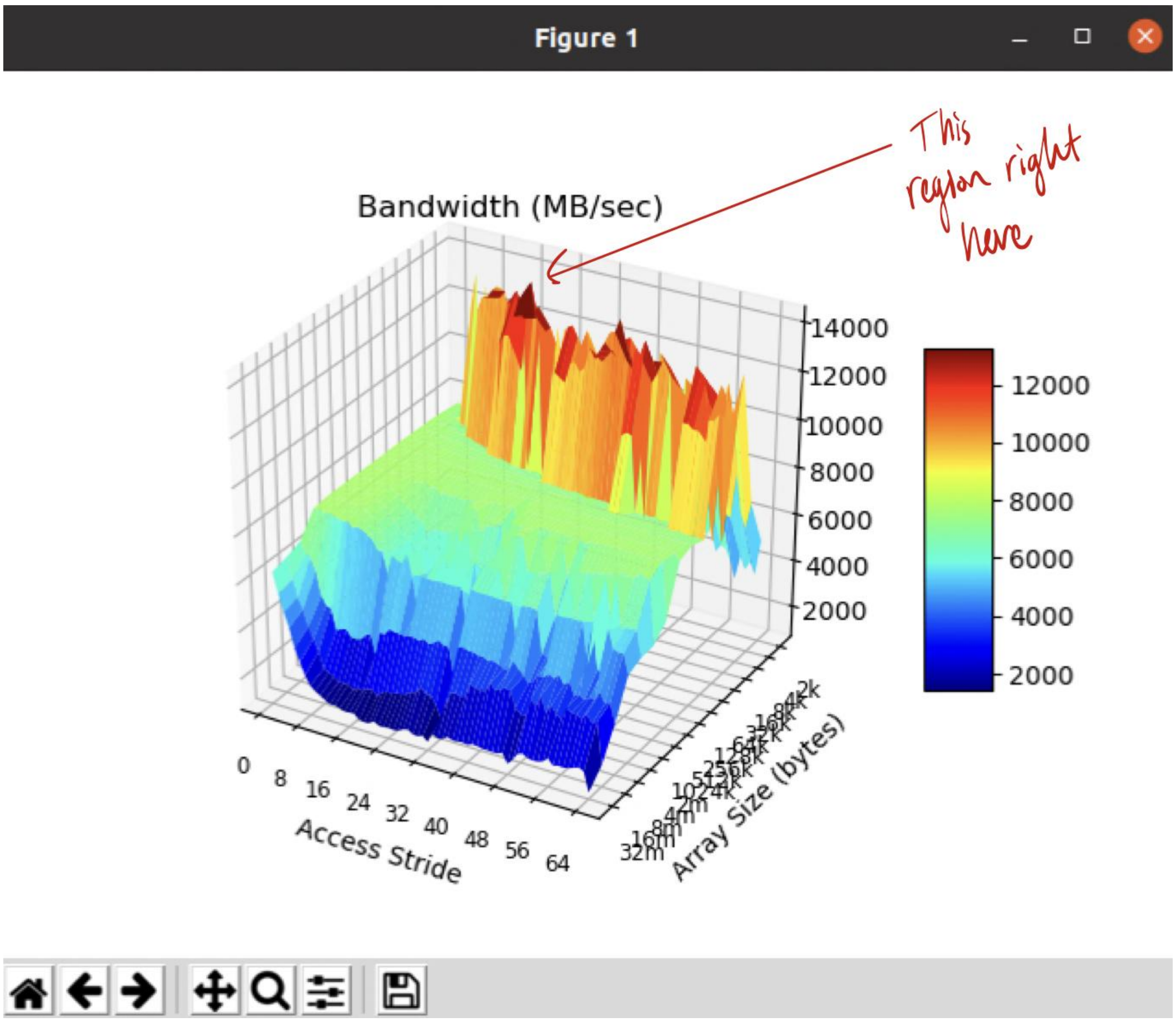
Question #18:

Take a screen capture of the displayed "memory mountain" (maximize the window so it's sufficiently large to read), and place the resulting image in your report

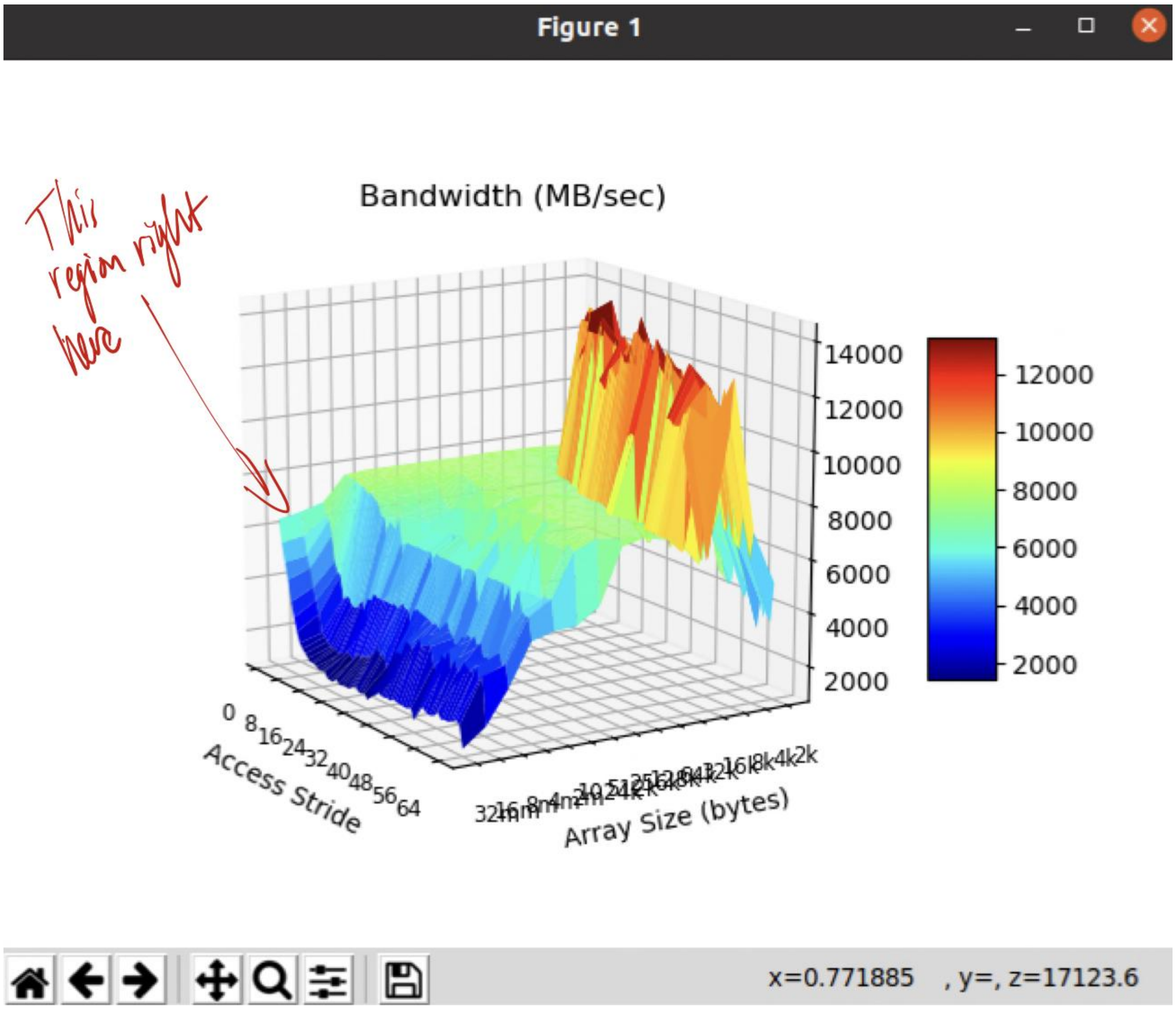
Answer:



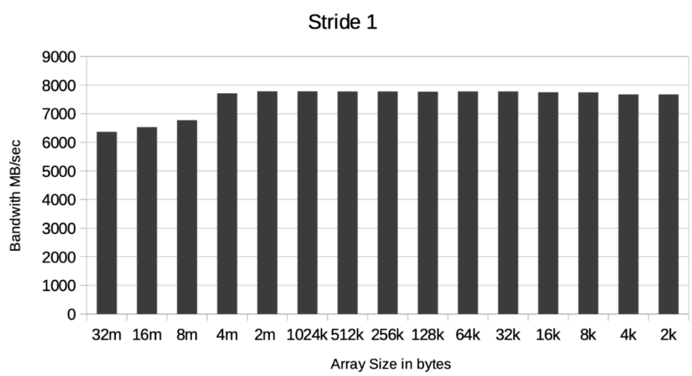
Question #19:
What region (array size, stride) gets the most **consistently** high performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.
Answer:
The region that is getting the most consistent high performance is in the range of array size 2k - 16k, and the stride size of 16 - 24 bytes. Moreover, the bandwidth that is reported is between is between 10000 - 12000 MB/sec

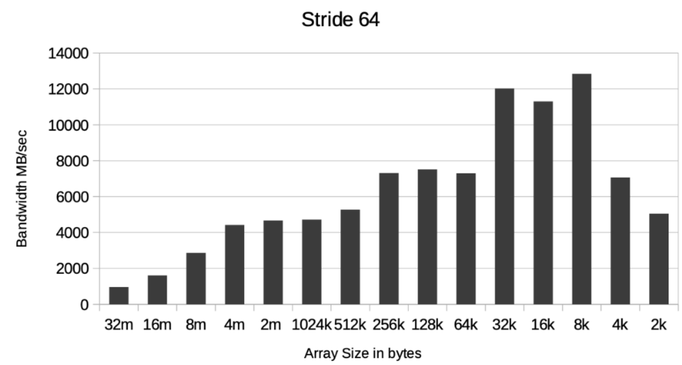


Question #20:
What region (array size, stride) gets the most **consistently** low performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.
Answer:
The region that is getting the most consistent low performance is in the range of array size 8m - 32m, and the stride size of 8 - 40 bytes. Moreover, the bandwidth that is reported is between greater than 2000 MB/sec



Question #21:
Using LibreOffice calc, create two new bar graphs: One for stride=1, and the other for stride=64. Place them side-by-size in the report.
No credit will be given for sloppy graphs that lack X and Y axis labels and a title.
You can obtain the raw data from results.txt. Open it in gedit and turn off *Text Wrapping* in the preferences. (Otherwise, the columns will be a mess)
Answer:





Question #22:
When you look at the graph for stride=1, you (should) see relatively high performance compared to stride=64. This is true even for large array sizes that are much larger than the L3 cache size reported in /proc/cpuinfo.
How is this possible, when the array cannot possibly all fit into the cache? Your explanation should include a brief overview of hardware prefetching as it applies to caches.
Answer:
When the stride is equal to 1, the data undergoes prefetching, which is the process when it is moved from lower level cache to higher level, i.e. it moves the data closer to the CPU. This leads to faster access time.

Question #23:
What is temporal locality? What is spatial locality?
Answer:
Temporal locality - When specific data is reused in a relatively small time duration
Spatial locality - When data elements are relatively close to each other in terms of storage locations

Question #24:
Adjusting the total array size impacts temporal locality - why? Will an increased array size increase or decrease temporal locality?
Answer:
Increasing the size of the array negatively impacts temporal locality, since with an increase in array size it would be less likely that the data gets reused.

Question #25:
Adjusting the read *stride* impacts spatial locality - why? Will an increased read stride increase or decrease spatial locality?
Answer:
Increasing the stride is a two-fold response. If you increase the stride a small amount, it will increase the spatial locality, since it would decrease the amount of time to access data. However, if your stride is large, then it would actually cause poor spatial locality, since you're not really accessing any nearby data in the cache.

Question #26:
As a software designer, describe at least 2 broad "guidelines" to ensure your programs run in the high-performing region of the graph instead of the low-performing region.
Answer:
1. Keep array sizes as small as possible
2. Pick the right size stride to maximize spatial locality, without having it be too large to the point where it would actually negatively affect performance.