Lab Report

ECPE 170 – Computer Systems and Networks – Spring 2021

Name: Kaung Khant Pyae Sone

Lab Topic: Performance Optimization (Memory Hierarchy) (Lab #: 7)

Question #1:
Describe how a two-dimensional array is stored in one-dimensional computer memory.
Answer:
This is the two dimensional array we declared: uint32_t array1[3][5]. It has 3 rows and 5 cols. These are the memory address allocated printed in row major order:

```
0x7fffeffb7be0 0x7fffeffb7be4 0x7fffeffb7be8 0x7fffeffb7bec 0x7fffeffb7bf0
0x7fffeffb7bf4 0x7fffeffb7bf8 0x7fffeffb7bfc 0x7fffeffb7c00 0x7fffeffb7c04
0x7fffeffb7c08 0x7fffeffb7c0c 0x7fffeffb7c10 0x7fffeffb7c14 0x7fffeffb7c18
```

Last 3 characters of memory addresses of row 1 col 4 to row 2 col 2:
Row 1 col 4: `0x7fffeffb7bec`
Row 1 col 5: `0x7fffeffb7bf0`
Row 2 col 1: `0x7fffeffb7bf4`
Row 2 col 2: `0x7fffeffb7bf8`

If we look at the last three characters of the memory addresses, the hexadecimal values are exactly 4 bytes away from each other, meaning that they are adjacent in memory. The row 1 col 5 value and row 2 col 1 memory address are also adjacent to each other, which shows that the two dimensional array is read row wise and stored in one dimensional memory.

Question #2:
Describe how a three-dimensional array is stored in one-dimensional computer memory.
Answer:
This is the three dimensional array we declared: uint32_t array1[2][3][4]. It has 2 blocks, each block has 3 rows and 5 columns. Below are the memory addresses of the two blocks in our array, printed in row major order.

```
Block 1
0x7ffd385635f0 0x7ffd385635f4 0x7ffd385635f8 0x7ffd385635fc 0x7ffd38563600
0x7ffd38563604 0x7ffd38563608 0x7ffd3856360c 0x7ffd38563610 0x7ffd38563614
0x7ffd38563618 0x7ffd3856361c 0x7ffd38563620 0x7ffd38563624 0x7ffd38563628

Block 2
0x7ffd3856362c 0x7ffd38563630 0x7ffd38563634 0x7ffd38563638 0x7ffd3856363c
0x7ffd38563640 0x7ffd38563644 0x7ffd38563648 0x7ffd3856364c 0x7ffd38563650
0x7ffd38563654 0x7ffd38563658 0x7ffd3856365c 0x7ffd38563660 0x7ffd38563664
```

Last 2 characters of memory addresses of block 1 row 3 col 5 to block 2 row 1 col 1:
Block 1 row 3 col 5: `0x7ffd3856362 8`
Block 2 row 1 col 1: `0x7ffd3856362c`

If we look at the last two characters of the memory addresses, we can see that the last row and col of block 1 and the first row and col of block 2 are adjacent to each other in memory. C treats each block of a three dimensional array as a two dimensional array and continuously reads all the blocks and stores them in one dimensional memory.

Question #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 Git repository.
Answer:
Output:

```
Memory address of two dimensional array array1[3][5] printed in row order
             0              1              2              3              4
0     0x7ffd385635b0 0x7ffd385635b4 0x7ffd385635b8 0x7ffd385635bc 0x7ffd385635c0
1     0x7ffd385635c4 0x7ffd385635c8 0x7ffd385635cc 0x7ffd385635d0 0x7ffd385635d4
2     0x7ffd385635d8 0x7ffd385635dc 0x7ffd385635e0 0x7ffd385635e4 0x7ffd385635e8

Memory addresses of row 1 col 4 to row 2 col 2:
Row 1 col 4: 0x7ffd385635bc
Row 1 col 5: 0x7ffd385635c0
Row 2 col 1: 0x7ffd385635c4
Row 2 col 2: 0x7ffd385635c8
Last three hexadecimals show that the memory addresses are adjacent to each other
in memory.
Row 1 col 5 and row 2 col 1 are adjacent in memory.
C reads two dimensional array rowwise and stores it in one dimensional memory

Memory address of three dimensional array array3[2][3][5] printed in row order
Block 1
             0              1              2              3              4
0     0x7ffd385635f0 0x7ffd385635f4 0x7ffd385635f8 0x7ffd385635fc 0x7ffd38563600
1     0x7ffd38563604 0x7ffd38563608 0x7ffd3856360c 0x7ffd38563610 0x7ffd38563614
2     0x7ffd38563618 0x7ffd3856361c 0x7ffd38563620 0x7ffd38563624 0x7ffd38563628

Block 2
             0              1              2              3              4
0     0x7ffd3856362c 0x7ffd38563630 0x7ffd38563634 0x7ffd38563638 0x7ffd3856363c
1     0x7ffd38563640 0x7ffd38563644 0x7ffd38563648 0x7ffd3856364c 0x7ffd38563650
2     0x7ffd38563654 0x7ffd38563658 0x7ffd3856365c 0x7ffd38563660 0x7ffd38563664

Memory addresses of block 1 row 3 col 5 to block 2 row 1 col 1:
Block 1 row 3 col 5: 0x7ffd38563628
Block 2 row 1 col 1: 0x7ffd3856362c
Last two hexadecimals show that they are apart by 4 bytes and adjacent to each
other.
Block 1 row 3 col 5 and block 2 row 1 col 1 are adjacent in memory.
C treats each block like a two dimensional array and continuosly reads all the
blocks.
```

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

Note 1: Address assumes array starts at memory address 0.
Note 2: Table is sorted by ascending memory addresses, not by program access order.

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:
The scalar variables i, j and sum has good temporal locality, as they are continuously accessed throughout the function. They are scalar variables so there is no guarantee of spatial locality.
The array a has poor temporal locality as each of its values are being read only once. However, it has good spatial locality as its contents are being read sequentially.

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

Note 1: Address assumes array starts at memory address 0.
Note 2: Table is sorted by ascending memory addresses, not by program access order.

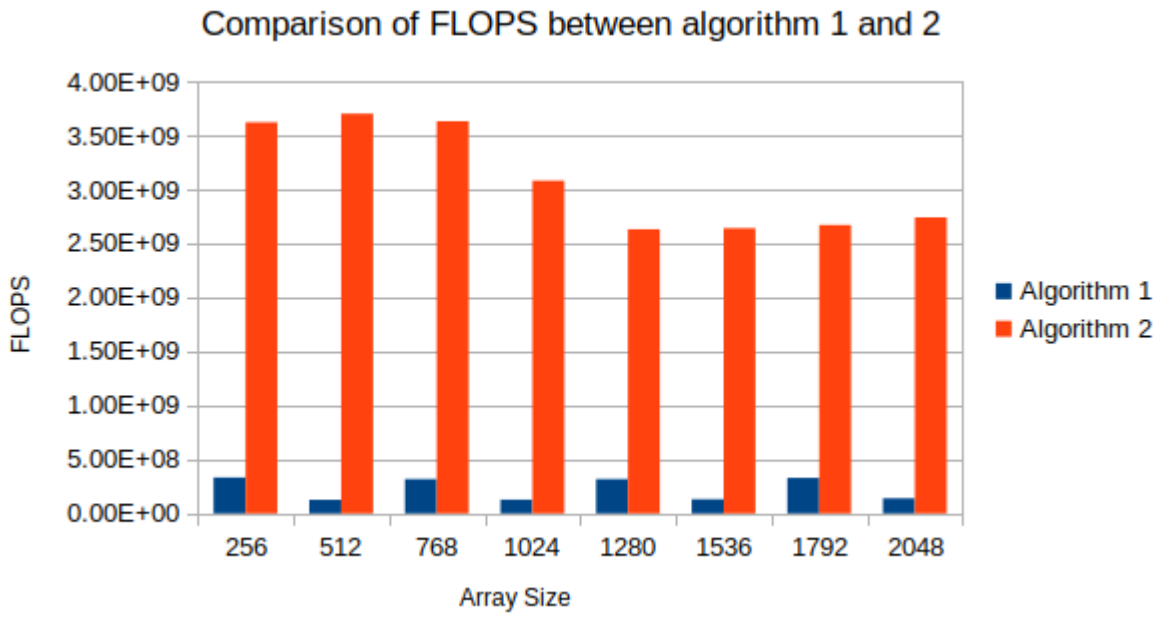
Question #7:
Does sumarraycols() have good temporal or spatial locality?
For your answer to receive full credit, you must discuss the locality of both the array itself, and the scalar variables such as i that are present in the function.
Answer:
The scalar variables i, j and sum has good temporal locality, as they are continuously accessed throughout the function. They are scalar variables so there is no guarantee of spatial locality.
The array a has poor temporal locality as each of its values are being read only once, and also poor spatial locality as it is not reading memory sequentially and instead jumping from memory addresses that are not adjacent to each other.

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:
The dimensions seem to be offset by n, which is the size of the array that the program takes in as an argument.

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:

Algorithm	Array Size	FLOPS
1	256	3.32E+08
	512	1.25E+08
	768	3.17E+08
	1024	1.26E+08
	1280	3.17E+08
	1536	1.30E+08
	1792	3.28E+08
	2048	1.39E+08
2	256	3.62E+09
	512	3.70E+09
	768	3.63E+09
	1024	3.08E+09
	1280	2.63E+09
	1536	2.64E+09
	1792	2.67E+09
	2048	2.74E+09

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.
Answer:



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:

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 8
model name      : AMD Ryzen 5 2600X Six-Core Processor
stepping        : 2
microcode       : 0xffffffff
cpu MHz         : 4000.108
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext
fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl tsc_reliable
nonstop_tsc cpuid extd_apicid pni pclmulqdq ssse3 fma cx16 sse4_1
sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor
lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse 3dnowprefetch
osvw topoext ssbd ibpb vmmcall fsgsbase bmi1 avx2 smep bmi2 rdseed
adx smap clflushopt sha_ni xsaveopt xsavec xgetbv1 xsaves clzero
virt_ssbd arat overflow_recov succor
bugs            : fxsave_leak sysret_ss_attrs null_seg spectre_v1
spectre_v2 spec_store_bypass
bogomips        : 8000.21
TLB size        : 2560 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 45 bits physical, 48 bits virtual
power management:
```

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!
Answer:
(a) L1 instruction cache size: 64 KB
(b) L1 data cache size: 32 KB
(c) L2 cache size: 512 KB
(d) L3 cache size: 8 MB
(e) https://www.cpu-world.com/CPUs/Zen/AMD-Ryzen%205%202600X.html

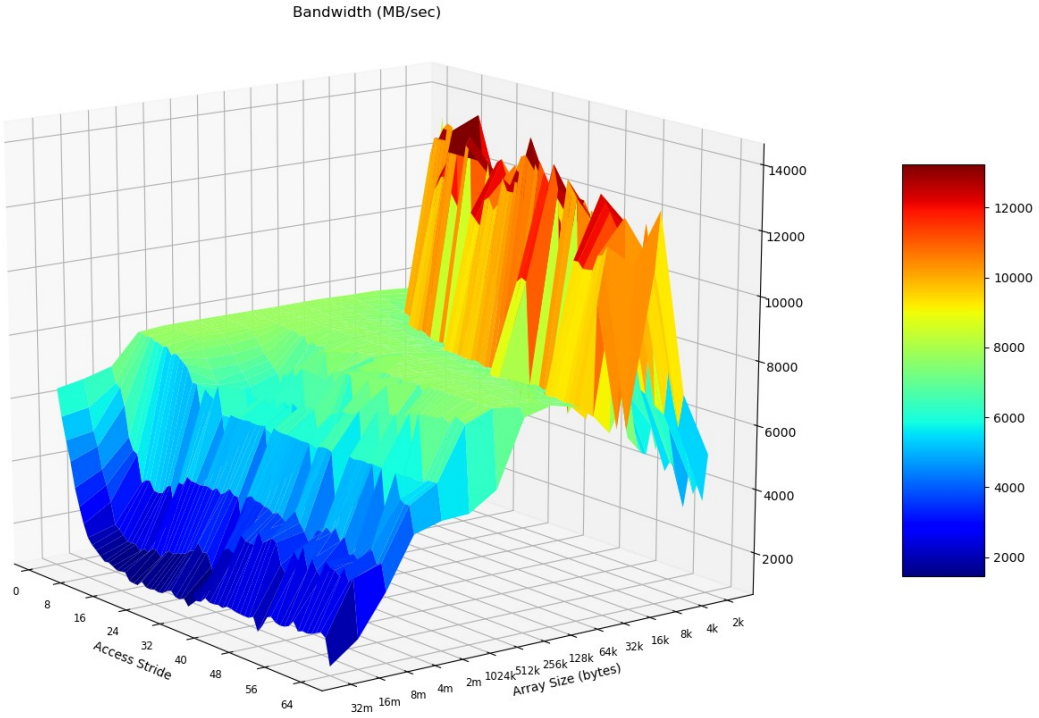
Question #14:
Why is it important to run the test program on an idle computer system?
Explain what would happen if the computer was running several other active programs in the background at the same time, and how that would impact the test results.
Answer:
To ensure that no other programs are using the CPU caches at the time we run our program, which could make our data bad.

Question #15:
What is the size (in bytes) of a data element read from the array in the test?
Answer:
8 bytes

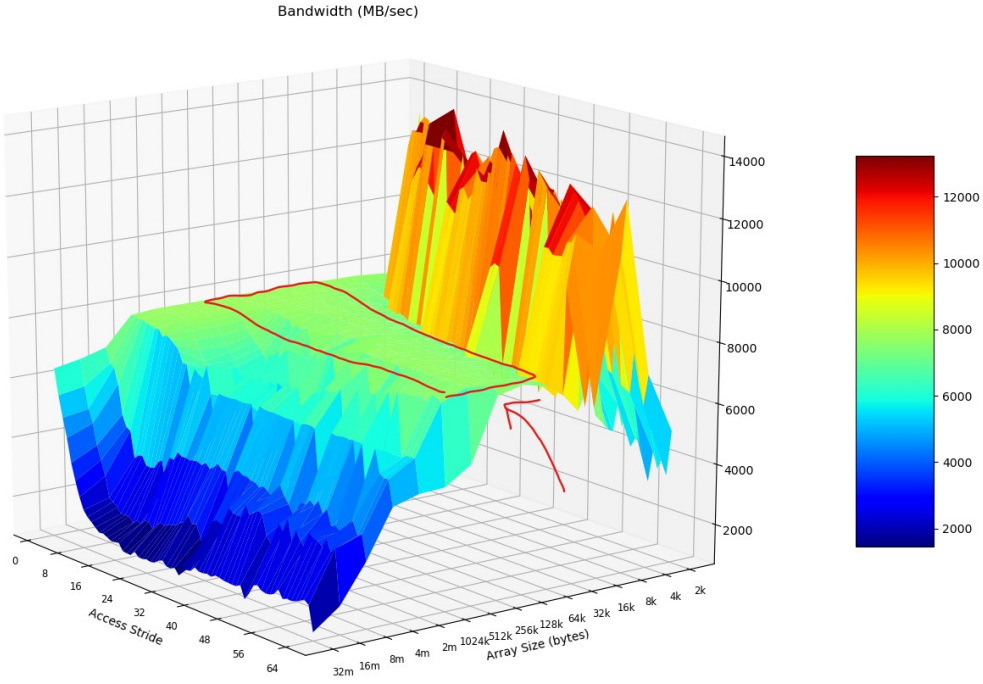
Question #16:
What is the range (min, max) of strides used in the test?
Answer:
0-64

Question #17:
What is the range (min, max) of array sizes used in the test?
Answer:
1 kb – 32 mb

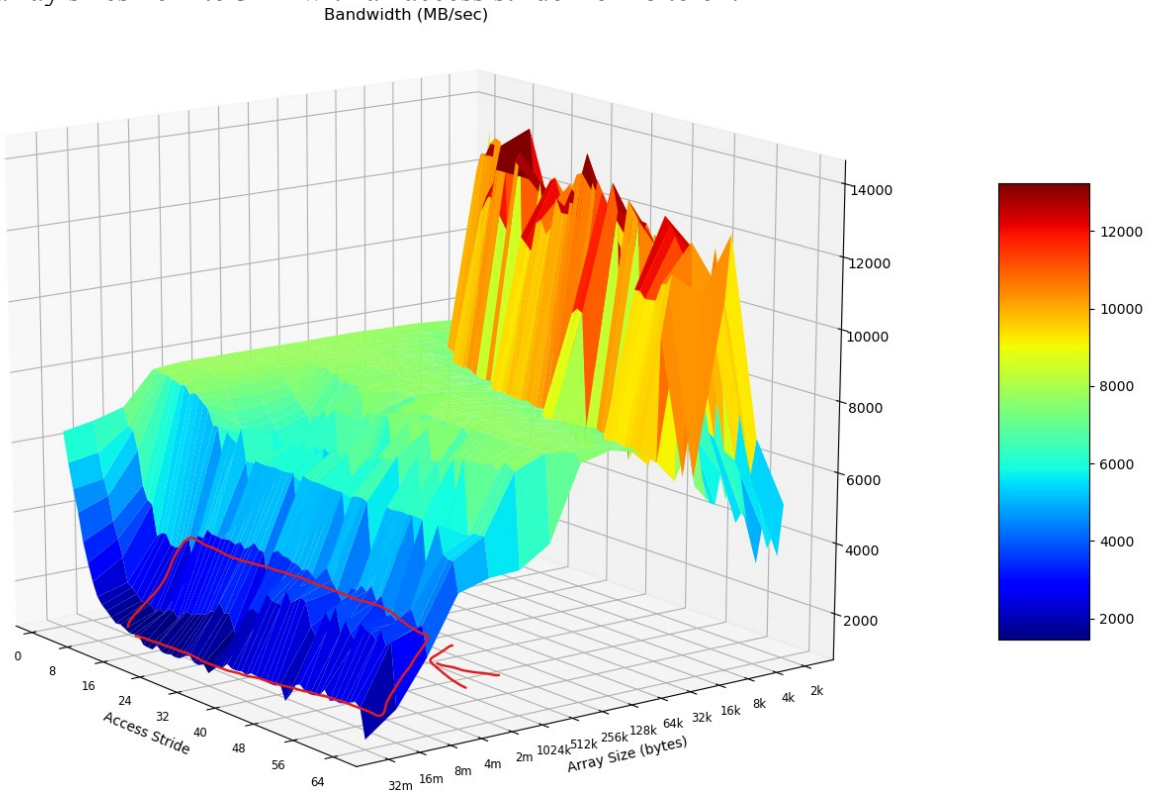
Question #18:
Take a screen capture of the displayed "memory mountain" (maximize the window so it's sufficiently large to read), and place the resulting image in your report
Answer:
This graphs below are using the results.txt obtained from the professor.



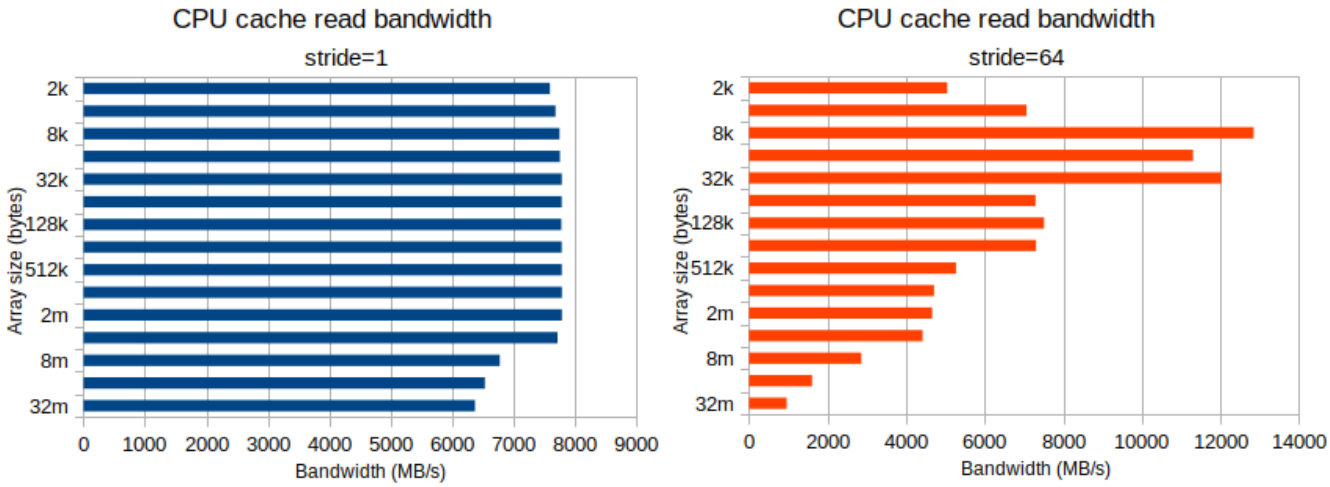
Question #19:
What region (array size, stride) gets the most consistently high performance? (Ignoring spikes in the graph that are noisy results...) What is the read bandwidth reported? Annotate your figure by drawing an arrow on it.
Answer:
Between array sizes 32k to 1024k with an access stride from 0 to 64.



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:
Between array sizes 16m to 32m with an access stride from 8 to 64.



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:
This graphs below are using the results.txt obtained from the professor.



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 reading arrays that are larger than the CPU cache, the CPU fetches data constantly from system memory as the mountain program reads data from the cache. The data being fetched is not being read by the program at the moment, but the CPU thinks that it will be used in the future and fetches it. This is called ‘cache prefetching’, where the CPU fetches data from system memory before it is actually needed, thus speeding up performance.

Question #23:
What is temporal locality? What is spatial locality?
Answer:
Temporal locality is data that is accessed or reused often, such as variables that are repeatedly called. Spatial locality is data that is stored close to other related data, such as an array whose contents are adjacent in memory.

Question #24:
Adjusting the total array size impacts temporal locality - why? Will an increased array size increase or decrease temporal locality?
Answer:
If the values inside an array is read only once, the temporal locality remains poor regardless of the change in array size. However, if the values are read multiple times, the temporal locality may become poorer as the array size increases as the whole array may not fit inside the cache.

Question #25:
Adjusting the read stride impacts spatial locality - why? Will an increased read stride increase or decrease spatial locality?
Answer:
As the read stride increases, the space in memory between the adjacent array values increases. This allows less of the array's data to be stored in the cache, which decreases spatial locality.

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:
Keep array sizes small.
Keep access strides low.
If you are coding for a specific system, then optimize your program to its specifications.