1)

Two-dimensional arrays are stored in one-dimensional computer memory by incrementing the last digit of the memory address by the amount of bytes required (in this case, 4 bytes to store an integer) for each stored value. For example, in the output for #3, the first value was stored at memory 0x7fff5136f700 and the next memory is stored 4 bytes thereafter which would be 0x7fff5136f704 and so on and so forth.

2)

Three-dimensional arrays are stored in one-dimensional computer memory by incrementing the last digit of the memory address by the amount of bytes required (just like two-dimensional arrays).

3)

```
TWO-D ARRAY
Value @ x:0, y:0 | 0 |   Memory: 0x7fff5136f700
Value @ x:0, y:1 | 1 |   Memory: 0x7fff5136f704
Value @ x:0, y:2 | 2 |   Memory: 0x7fff5136f708
Value @ x:1, y:0 | 3 |   Memory: 0x7fff5136f70c
Value @ x:1, y:1 | 4 |   Memory: 0x7fff5136f710
Value @ x:1, y:2 | 5 |   Memory: 0x7fff5136f714
Value @ x:2, y:0 | 6 |   Memory: 0x7fff5136f718
Value @ x:2, y:1 | 7 |   Memory: 0x7fff5136f71c
Value @ x:2, y:2 | 8 |   Memory: 0x7fff5136f720
THREE-D ARRAY
Value @ x:0, y:0, z:0 | 0 |     Memory: 0x7fff5136f730
Value @ x:0, y:0, z:1 | 1 |     Memory: 0x7fff5136f734
Value @ x:0, y:0, z:2 | 2 |     Memory: 0x7fff5136f738
Value @ x:0, y:1, z:0 | 3 |     Memory: 0x7fff5136f73c
Value @ x:0, y:1, z:1 | 4 |     Memory: 0x7fff5136f740
Value @ x:0, y:1, z:2 | 5 |     Memory: 0x7fff5136f744
Value @ x:0, y:2, z:0 | 6 |     Memory: 0x7fff5136f748
Value @ x:0, y:2, z:1 | 7 |     Memory: 0x7fff5136f74c
Value @ x:0, y:2, z:2 | 8 |     Memory: 0x7fff5136f750
Value @ x:1, y:0, z:0 | 9 |     Memory: 0x7fff5136f754
Value @ x:1, y:0, z:1 | 10 |    Memory: 0x7fff5136f758
Value @ x:1, y:0, z:2 | 11 |    Memory: 0x7fff5136f75c
Value @ x:1, y:1, z:0 | 12 |    Memory: 0x7fff5136f760
Value @ x:1, y:1, z:1 | 13 |    Memory: 0x7fff5136f764
Value @ x:1, y:1, z:2 | 14 |    Memory: 0x7fff5136f768
Value @ x:1, y:2, z:0 | 15 |    Memory: 0x7fff5136f76c
Value @ x:1, y:2, z:1 | 16 |    Memory: 0x7fff5136f770
Value @ x:1, y:2, z:2 | 17 |    Memory: 0x7fff5136f774
Value @ x:2, y:0, z:0 | 18 |     Memory: 0x7fff5136f778
Value @ x:2, y:0, z:1 | 19 |     Memory: 0x7fff5136f77c
Value @ x:2, y:0, z:2 | 20 |     Memory: 0x7fff5136f780
Value @ x:2, y:1, z:0 | 21 |     Memory: 0x7fff5136f784
Value @ x:2, y:1, z:1 | 22 |     Memory: 0x7fff5136f788
Value @ x:2, y:1, z:2 | 23 |     Memory: 0x7fff5136f78c
Value @ x:2, y:2, z:0 | 24 |     Memory: 0x7fff5136f790
Value @ x:2, y:2, z:1 | 25 |     Memory: 0x7fff5136f794
Value @ x:2, y:2, z:2 | 26 |     Memory: 0x7fff5136f798
```

4)

**Table:** Access Pattern for `sumarrayrows()` Assuming ROWS=2 and COLS=3

| | | | | | | |
|---|---|---|---|---|---|---|
| **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 |

5)

The function sumarrayrows() has good spatial locality. The scalar variables i, j and sum have temporal locality as they are accessed within the function numerous time through the for loop (i++, j++, sum += a[i][j]) However, the array a has spatial locality as each content of the array is only accessed once.

6)

**Table:** Access Pattern for `sumarraycols()` Assuming ROWS=2 and COLS=3

| | | | | | | |
|---|---|---|---|---|---|---|
| **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 |

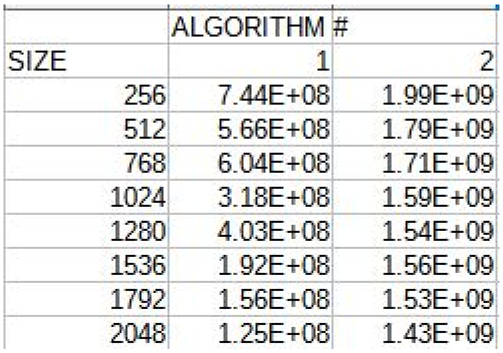
7)

The function sumarraycols() has good spatial locality. The scalar variables i, j and sum have temporal locality as they are accessed within the function numerous time through the for loop (i++, j++, sum += a[i][j]) However, the array a has spatial locality as each content of the array is only accessed once.

8)

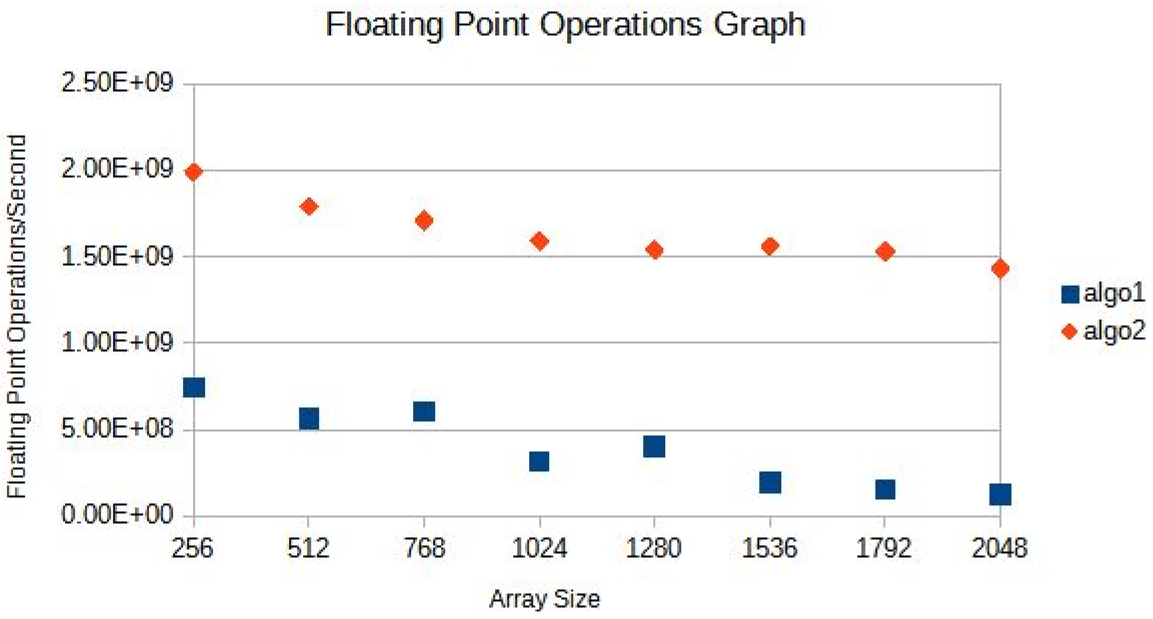
Analyzing the source code, the two-dimensional arrays are stored in memory as a one-dimensional array by mallocing (n*n*8), n being the array size input by the user and 8 being the byte size of a double.

9)



| | ALGORITHM # | |
|---|---|---|
| SIZE | 1 | 2 |
| 256 | 7.44E+08 | 1.99E+09 |
| 512 | 5.66E+08 | 1.79E+09 |
| 768 | 6.04E+08 | 1.71E+09 |
| 1024 | 3.18E+08 | 1.59E+09 |
| 1280 | 4.03E+08 | 1.54E+09 |
| 1536 | 1.92E+08 | 1.56E+09 |
| 1792 | 1.56E+08 | 1.53E+09 |
| 2048 | 1.25E+08 | 1.43E+09 |

10)



12)

```
processor	: 0
vendor_id	: GenuineIntel
cpu family	: 6
model		: 158
model name	: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
stepping	: 10
microcode	: 0xb4
cpu MHz		: 2303.999
cache size	: 8192 KB
physical id	: 0
siblings	: 1
core id		: 0
cpu cores	: 1
apicid		: 0
initial apicid	: 0
fpu		: yes
fpu_exception	: yes
cpuid level	: 22
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 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl
xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2
x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm
3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 avx2
smep bmi2 invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves arat md_clear
flush_l1d arch_capabilities
bugs		: cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
itlb_multihit srbds
bogomips	: 4607.99
clflush size	: 64
cache_alignment	: 64
address sizes	: 45 bits physical, 48 bits virtual
power management:
```

13)

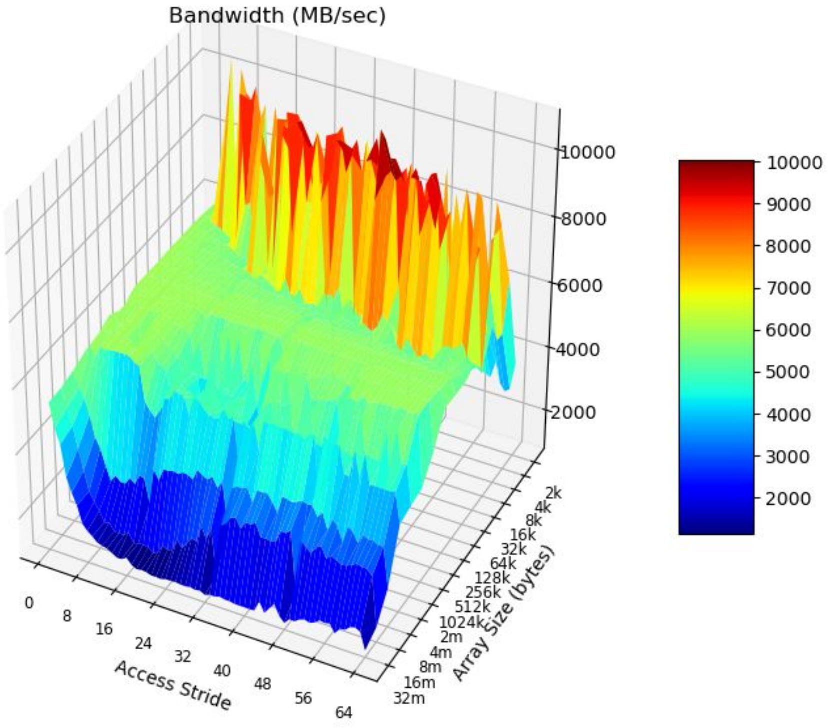
a) 4 x 32 KB 8-way set associative instruction caches
b) 4 x 32 KB 8-way set associative data caches
c) 4 x 256 KB 4-way set associative caches
d) 8 MB 16-way set associative shared cache
e) https://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5%20i5-8300H.html

14) It is important to run the test program on an idle computer system because the program is accessing memory of large sizes which may be slowed by other programs. It may result in skewed results with the mountain rising earlier than it would. (Higher blues, greens, and red)

15) 32m bytes(?) I'm not sure if this is correct

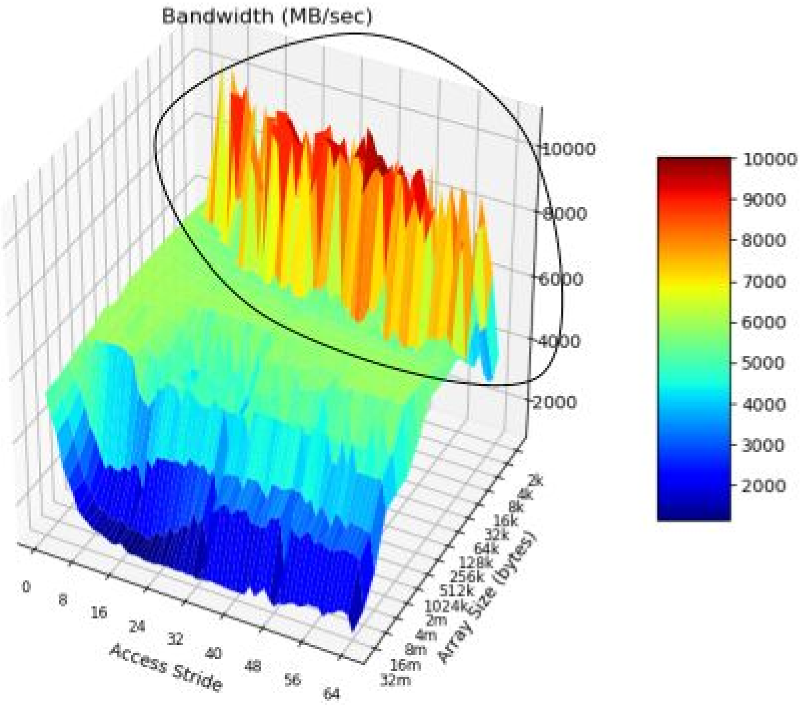
16) Min: 0, Max: 64
17) Min: 2k bytes, Max: 32m bytes

18)



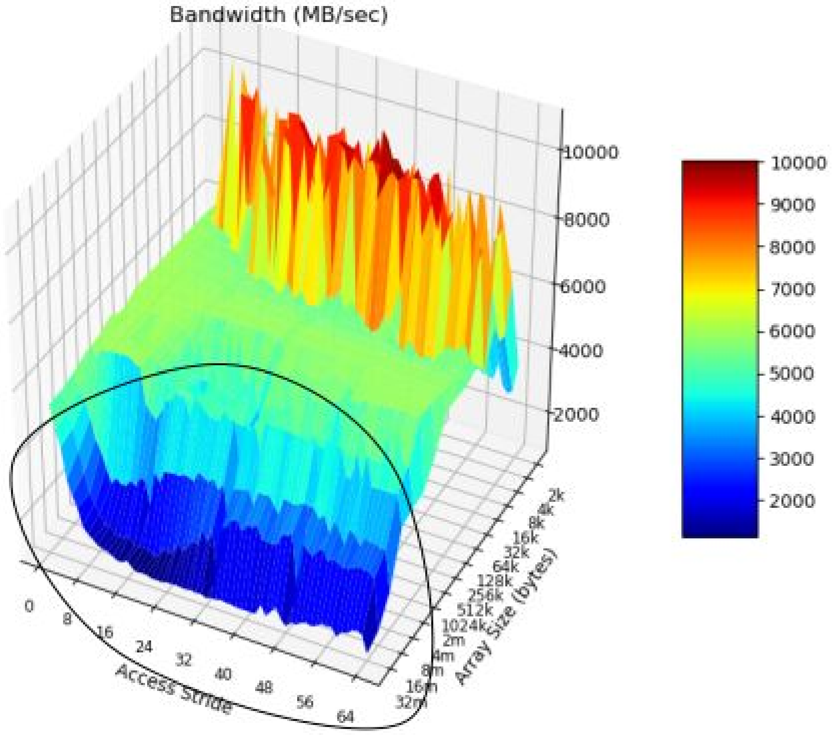
19)

Array Size: 16k to 2k, Access Stride 0 to 64 with a bandwidth of between 7000 to 10000.

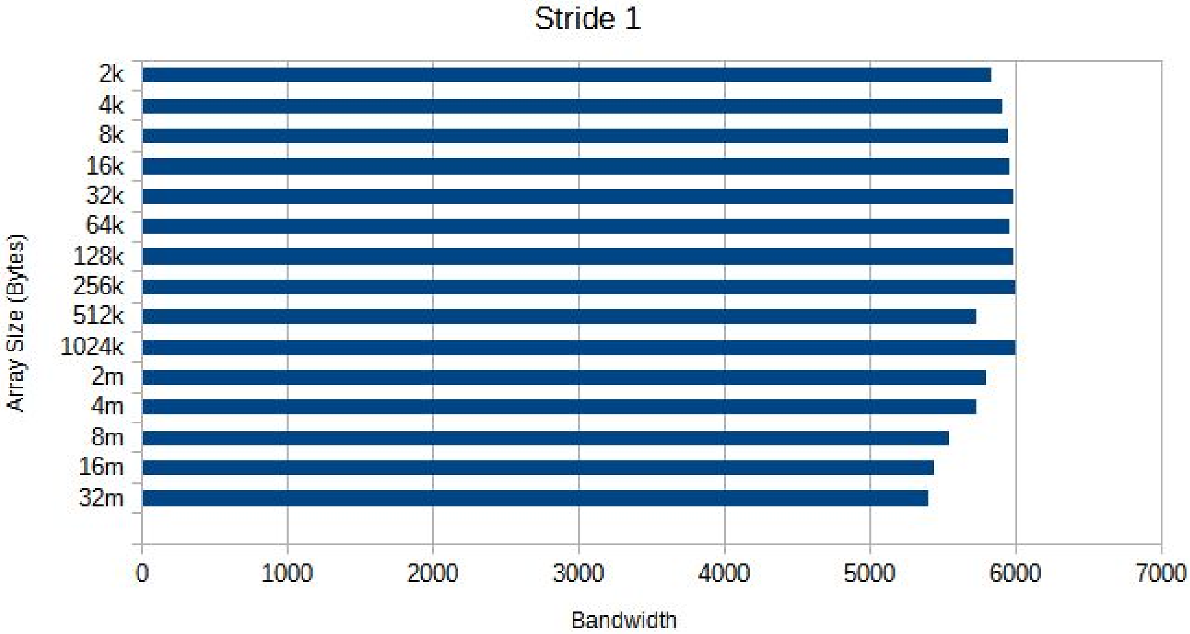


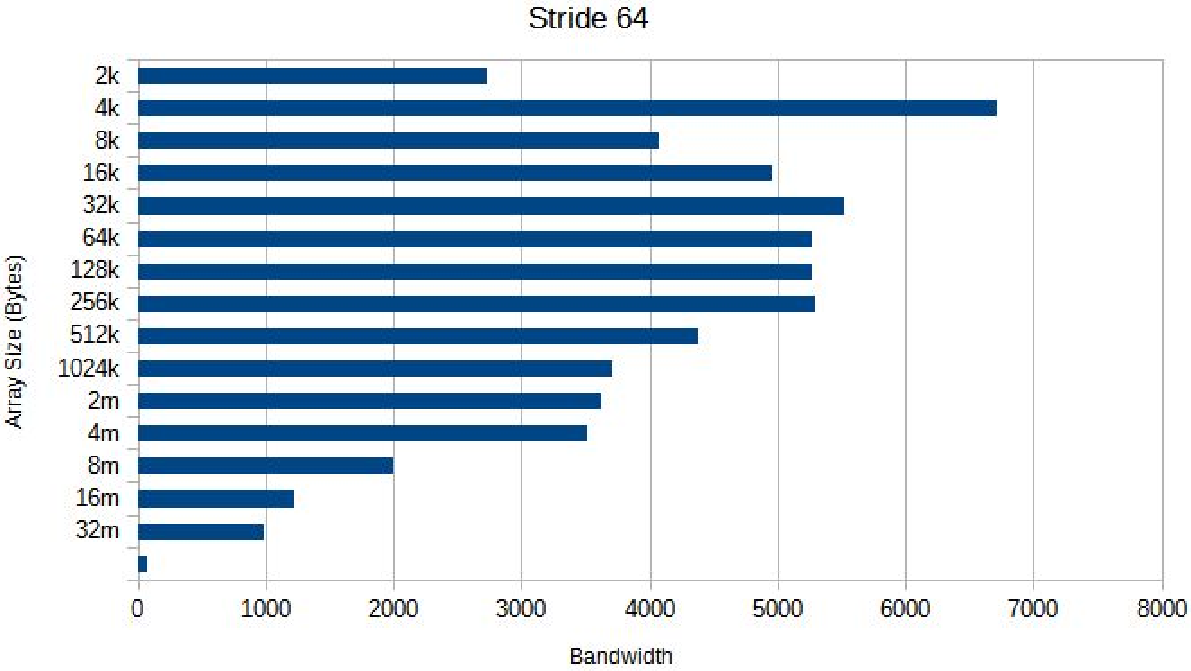
20)

Array Size: 32m to 8m, Access Stride 8 to 64 with a bandwidth of between 0 to 3000.



21)





22) In stride 1, it is reading new instruction resulting in a hefty amount of processing power. As it has read the instruction in stride 1, it puts the data into a cache to be easily retrievable if necessary so by the time it is stride 64, as the data is already in cache, the data is in close range, resulting in a significantly lower amount of processing power.

23) Temporal locality is when you reuse data while spatial locality is when you use data that may have nearby memory which will be used.

24) Adjusting the array size may impact temporal locality because the larger the array size, the more new data you're creating, which will result in reusing the data less. Therefore, an increased array size would decrease temporal locality.

25) Adjusting the read stride may impact spatial locality because the larger the read stride, the more it has already read the data, which will result in reusing the data more. Therefore, an increased read stride would increase spatial locality.

26) In order to run programs in high-performing regions of the graph, it should have low spatial locality and high temporal locality.